



MATPOWER Reference Manual

Release 8.0

Ray D. Zimmerman

May 17, 2024

Contents

1	Introduction	1
2	Functions	2
2.1	MATPOWER Installer	2
2.2	Top-Level Simulation Functions	4
2.3	Other Functions	7
3	Classes	9
3.1	Task Classes	9
3.2	Data Model Classes	30
3.3	Data Model Converter Classes	61
3.4	Network Model Classes	74
3.5	Mathematical Model Classes	124
3.6	Miscellaneous Classes	159
3.7	MATPOWER Extension Classes	173
4	Tests	217
4.1	MATPOWER Tests	217
4.2	MATPOWER Test Data	220
5	Legacy	223
5.1	Legacy Class	223
5.2	Legacy Functions	229
5.3	Legacy Tests	374
6	Previous Versions	387
	Index	388

The purpose of this *Reference Manual* is to provide reference documentation on each class and function in MATPOWER.

This documentation is automatically generated from the corresponding help text in the Matlab source for each function, class, property or method.

The GitHub icon in the upper right of each reference page links to the corresponding source file in the master branch on GitHub.

Currently, this manual includes *only* classes and functions that make up the new **MP-Core** and the **flexible** and **legacy** MATPOWER frameworks, but not the other legacy MATPOWER functions or the included packages [MP-Opt-Model](#), [MIPS](#), [MP-Test](#), or [MOST](#).

2.1 MATPOWER Installer

Note: The `install_matpower()` (page 2) function is generally **not** in your MATLAB/Octave path, unless you change your current working directory to the MATPOWER install directory where it is located.

2.1.1 `install_matpower`

`install_matpower(modify, save_it, verbose, rm_oldpaths)`

`install_matpower()` (page 2) - Assist user in setting path to install MATPOWER.

```
install_matpower
install_matpower(modify)
install_matpower(modify, save_it)
install_matpower(modify, save_it, verbose)
install_matpower(modify, save_it, verbose, rm_oldpaths)
success = install_matpower(...)
```

Assists the user in setting up the proper MATLAB/Octave path to be able to use MATPOWER and run its tests. With no input arguments it prompts interactively to determine how to handle the paths.

Note: This function is generally **not** in your MATLAB/Octave path, unless you change your current working directory to the MATPOWER install directory where it is located.

There are two main approaches for installing MATPOWER.

1. If you have a single version of MATPOWER, select the options to modify and save the path (interactive option 3). This will add MATPOWER to your default MATLAB/Octave path for all future sessions.
2. If you have multiple versions of MATPOWER, select the options to not modify the path (interactive option 1), but to save the `addpath()` commands to a file. Then execute the saved file to use this version of MATPOWER.

All inputs and outputs are optional.

Inputs

- **modify** (*boolean*) – select how to set path
 - 0 (*default*) - generate relevant `addpath()` commands, but don't execute them; MATPOWER is not installed
 - 1 - modify the path by executing the relevant `addpath()` commands; MATPOWER is installed for this session
- **save_it** (*integer or string*) – indicates whether or not to save the results
 - 0 or [] (*default*) - don't save any results
 - if `modify` is 0
 - * `save_it = <string>` : the relevant `addpath()` commands are saved to a file whose name is provided in `save_it`; execute saved file in any session to make MATPOWER available for the session
 - * `save_it = <other true value>` : the relevant `addpath()` commands are saved to a file named `'startup.m'` in the current directory; MATPOWER is available in any session affected by this `'startup.m'` file
 - * *otherwise* : the commands are displayed, but not saved
 - if `modify` is 1
 - * `save_it = <any true value>` : the path will be modified and saved with `savepath()`; MATPOWER is available in this and all future sessions
 - * *otherwise* : the path will be modified but not saved
- **verbose** (*boolean*) – prints the relevant `addpath()` commands if true (*default*), silent otherwise
- **rm_oldpaths** (*boolean*) – remove existing installation
 - 0 (*default*) - do **not** remove existing MATPOWER from path
 - 1 - remove existing MATPOWER paths first

Output

success (*boolean*) – 1 if all commands succeeded, 0 otherwise

Examples:

```
install_matpower          % interactive mode, prompt for options
install_matpower(0);      % print the required addpath() commands
install_matpower(0, 1);    % save the commands to startup.m
install_matpower(1, 1);    % modify my path and save
install_matpower(1, 0, 0); % modify my path temporarily & silently
install_matpower(0, 'matpower8'); % save the commands to matpower8.m
install_matpower(0, 0, 1, 1); % uninstall MATPOWER from path (must
                             % call savepath() separately to make
                             % permanent)
```

See also `addpath`, `savepath`.

2.2 Top-Level Simulation Functions

These are top-level functions intended as user commands for running power flow (PF), continuation power flow (CPF), optimal power flow (OPF) and other custom simulation or optimization tasks.

2.2.1 run_mp

run_mp(*task_class*, *d*, *mpopt*, *varargin*)

[run_mp\(\)](#) (page 4) - Run any MATPOWER simulation.

```
run_mp(task_class, d, mpopt)
run_mp(task_class, d, mpopt, ...)
task = run_mp(...)
```

This is **the** main function in the **flexible framework** for running MATPOWER. It creates the task object, applying any specified extensions, runs the task, and prints or saves the solution, if desired.

It is typically called from one of the wrapper functions such as [run_pf\(\)](#) (page 5), [run_cpf\(\)](#) (page 5), or [run_opf\(\)](#) (page 6).

Inputs

- **task_class** (*function handle*) – handle to constructor of default task class for type of task to be run, e.g. [mp.task_pf](#) (page 21) for power flow, [mp.task_cpf](#) (page 22) for CPF, and [mp.task_opf](#) (page 24) for OPF
- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (mpc)
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct

Additional optional inputs can be provided as *<name>*, *<val>* pairs, with the following options:

- 'print_fname' - file name for saving pretty-printed output
- 'soln_fname' - file name for saving solved case
- 'mpx' - MATPOWER extension or cell array of MATPOWER extensions to apply

Output

task ([mp.task](#) (page 9)) – task object containing the solved run including the data, network, and mathematical model objects.

Solution results are available in the data model, and its elements, contained in the returned task object. For example:

```
task = run_opf('case9');
lam_p = task.dm.elements.bus.tab.lam_p    % nodal price
pg = task.dm.elements.gen.tab.pg          % generator active dispatch
```

See also [run_pf\(\)](#) (page 5), [run_cpf\(\)](#) (page 5), [run_opf\(\)](#) (page 6), [mp.task](#) (page 9).

2.2.2 run_pf

run_pf(varargin)

[run_pf\(\)](#) (page 5) - Run a power flow.

```
run_pf(d, mpopt)
run_pf(d, mpopt, ...)
task = run_pf(...)
```

This is the main function used to run power flow (PF) problems via the **flexible MATPOWER framework**.

This function is a simple wrapper around [run_mp\(\)](#) (page 4), calling it with the first argument set to @mp.task_pf.

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (mpc)
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct

Additional optional inputs can be provided as <name>, <val> pairs, with the following options:

- 'print_fname' - file name for saving pretty-printed output
- 'soln_fname' - file name for saving solved case
- 'mpx' - MATPOWER extension or cell array of MATPOWER extensions to apply

Output

task ([mp.task_pf](#) (page 21)) – task object containing the solved run including the data, network, and mathematical model objects.

Solution results are available in the data model, and its elements, contained in the returned task object. For example:

```
task = run_pf('case9');
va = task.dm.elements.bus.tab.va    % bus voltage angles
pg = task.dm.elements.gen.tab.pg    % generator active dispatch
```

See also [run_mp\(\)](#) (page 4), [mp.task_pf](#) (page 21).

2.2.3 run_cpf

run_cpf(varargin)

[run_cpf\(\)](#) (page 5) Run a continuation power flow.

```
run_cpf(d, mpopt)
run_cpf(d, mpopt, ...)
task = run_cpf(...)
```

This is the main function used to run continuation power flow (CPF) problems via the **flexible MATPOWER framework**.

This function is a simple wrapper around [run_mp\(\)](#) (page 4), calling it with the first argument set to @mp.task_cpf.

Inputs

- **d** – data source specification, currently assumed to be a cell array of two MATPOWER case names or case structs (`mpc`), the first being the base case, the second the target case
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct

Additional optional inputs can be provided as `<name>`, `<val>` pairs, with the following options:

- `'print_fname'` - file name for saving pretty-printed output
- `'soln_fname'` - file name for saving solved case
- `'mpx'` - MATPOWER extension or cell array of MATPOWER extensions to apply

Output

task (`mp.task_cpf` (page 22)) – task object containing the solved run including the data, network, and mathematical model objects.

Solution results are available in the data model, and its elements, contained in the returned task object. For example:

```
task = run_cpf({'case9', 'case9target'});  
vm = task.dm.elements.bus.tab.vm      % bus voltage magnitudes  
pg = task.dm.elements.gen.tab.pg      % generator active dispatch
```

See also `run_mp()` (page 4), `mp.task_cpf` (page 22).

2.2.4 run_opf

run_opf(*varargin*)

`run_opf()` (page 6) Run an optimal power flow.

```
run_opf(d, mpopt)  
run_opf(d, mpopt, ...)  
task = run_opf(...)
```

This is the main function used to run optimal power flow (OPF) problems via the **flexible MATPOWER framework**.

This function is a simple wrapper around `run_mp()` (page 4), calling it with the first argument set to `@mp.task_opf`.

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct

Additional optional inputs can be provided as `<name>`, `<val>` pairs, with the following options:

- `'print_fname'` - file name for saving pretty-printed output
- `'soln_fname'` - file name for saving solved case
- `'mpx'` - MATPOWER extension or cell array of MATPOWER extensions to apply

Output

task ([mp.taskopf](#) (page 24)) – task object containing the solved run including the data, network, and mathematical model objects.

Solution results are available in the data model, and its elements, contained in the returned task object. For example:

```
task = run_opf('case9');
lam_p = task.dm.elements.bus.tab.lam_p % nodal price
pg = task.dm.elements.gen.tab.pg % generator active dispatch
```

See also [run_mp\(\)](#) (page 4), [mp.taskopf](#) (page 24).

2.3 Other Functions

2.3.1 mp.load_dm

`mp.load_dm(d, task_class, mpopt, varargin)`

[mp.load_dm\(\)](#) (page 7) - Load a MATPOWER data model.

```
dm = mp.load_dm(d)
dm = mp.load_dm(d, task_class)
dm = mp.load_dm(d, task_class, mpopt)
dm = mp.load_dm(d, task_class, mpopt, ...)
[dm, task] = mp.load_dm(...)
```

Uses a task object to load a MATPOWER data model object, optionally returning the task object as well.

The resulting data model object can later be passed to [run_pf\(\)](#) (page 5), [run_cpf\(\)](#) (page 5), [run_opf\(\)](#) (page 6), or directly to the [run\(\)](#) (page 12) method of the task object.

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (mpc)
- **task_class** (*function handle*) – (*optional*) handle to constructor of task class, (*default is* [mp.taskopf](#) (page 24))
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct

Additional optional inputs can be provided as <name>, <val> pairs, with the following options:

- 'mpx' - MATPOWER extension or cell array of MATPOWER extensions to apply

Outputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **task** ([mp.task](#) (page 9)) – task object containing the solved run including the data, network, and mathematical model objects.

See also [run_pf\(\)](#) (page 5), [run_cpf\(\)](#) (page 5), [run_opf\(\)](#) (page 6), [mp.task](#) (page 9).

2.3.2 mp_table_class

mp_table_class()

mp_table_class() (page 8) - Returns handle to constructor for `table` or *mp_table* (page 159).

Returns a handle to `table` constructor, if it is available, otherwise to *mp_table* (page 159) constructor. Useful for table-based code that is compatible with both MATLAB (using native tables) and Octave (using *mp_table* (page 159) or the `table` implementation from Tablicious, if available).

```
% Works in MATLAB or Octave, which does not (yet) natively support table().  
table_class = mp_table_class();  
T = table_class(var1, var2, ...);
```

See also `table`, *mp_table* (page 159).

3.1 Task Classes

3.1.1 Core Task Classes

`mp.task`

`class mp.task`

Bases: `handle`

`mp.task` (page 9) - MATPOWER task abstract base class.

Each task type (e.g. power flow, CPF, OPF) will inherit from `mp.task` (page 9).

Provides properties and methods related to the specific problem specification being solved (e.g. power flow, continuation power flow, optimal power flow, etc.). In particular, it coordinates all interactions between the 3 (data, network, mathematical) model layers.

The model objects, and indirectly their elements, as well as the solution success flag and messages from the mathematical model solver, are available in the properties of the task object.

`mp.task` Properties:

- `tag` (page 11) - task tag - e.g. 'PF', 'CPF', 'OPF'
- `name` (page 11) - task name - e.g. 'Power Flow', etc.
- `dmc` (page 11) - data model converter object
- `dm` (page 11) - data model object
- `nm` (page 11) - network model object
- `mm` (page 11) - mathematical model object
- `mm_opt` (page 11) - solve options for mathematical model
- `i_dm` (page 11) - iteration counter for data model loop
- `i_nm` (page 11) - iteration counter for network model loop

- `i_mm` (page 11) - iteration counter for math model loop
- `success` (page 11) - success flag, 1 - math model solved, 0 - didn't solve
- `message` (page 11) - output message
- `et` (page 11) - elapsed time (seconds) for `run()` (page 12) method

mp.task Methods:

- `load_dm()` (page 11) - load the data model
- `run()` (page 12) - execute the task
- `next_mm()` (page 12) - controls iterations over mathematical models
- `next_nm()` (page 12) - controls iterations over network models
- `next_dm()` (page 13) - controls iterations over data models
- `run_pre()` (page 13) - called at beginning of `run()` (page 12) method
- `run_post()` (page 13) - called at end of `run()` (page 12) method
- `print_soln()` (page 14) - display pretty-printed results
- `print_soln_header()` (page 14) - display success/failure, elapsed time
- `save_soln()` (page 14) - save solved case to file
- `dm_converter_class()` (page 14) - get data model converter constructor
- `dm_converter_class_mpc2_default()` (page 14) - get default data model converter constructor
- `dm_converter_create()` (page 15) - create data model converter object
- `data_model_class()` (page 15) - get data model constructor
- `data_model_class_default()` (page 15) - get default data model constructor
- `data_model_create()` (page 16) - create data model object
- `data_model_build()` (page 16) - create and build data model object
- `data_model_build_pre()` (page 16) - called at beginning of `data_model_build()` (page 16)
- `data_model_build_post()` (page 17) - called at end of `data_model_build()` (page 16)
- `network_model_class()` (page 17) - get network model constructor
- `network_model_class_default()` (page 17) - get default network model constructor
- `network_model_create()` (page 17) - create network model object
- `network_model_build()` (page 18) - create and build network model object
- `network_model_build_pre()` (page 18) - called at beginning of `network_model_build()` (page 18)
- `network_model_build_post()` (page 18) - called at end of `network_model_build()` (page 18)
- `network_model_x_soln()` (page 18) - update network model state from math model solution
- `network_model_update()` (page 19) - update net model state/soln from math model soln
- `math_model_class()` (page 19) - get mathematical model constructor
- `math_model_class_default()` (page 19) - get default mathematical model constructor
- `math_model_create()` (page 19) - create mathematical model object

- `math_model_build()` (page 20) - create and build mathematical model object
- `math_model_opt()` (page 20) - get options struct to pass to `mm.solve()`

See the `sec_task` section in the *MATPOWER Developer's Manual* for more information.

See also `mp.data_model` (page 30), `mp.net_model` (page 93), `mp.math_model` (page 124), `mp.dm_converter` (page 62).

Property Summary

tag

(*char array*) task `tag` (page 11) - e.g. 'PF', 'CPF', 'OPF'

name

(*char array*) task `name` (page 11) - e.g. 'Power Flow', etc.

dmc

(`mp.dm_converter` (page 62)) data model converter object

dm

(`mp.data_model` (page 30)) data model object

nm

(`mp.net_model` (page 93)) network model object

mm

(`mp.math_model` (page 124)) mathematical model object

mm_opt

(*struct*) solve options for mathematical model

i_dm

(*integer*) iteration counter for data model loop

i_nm

(*integer*) iteration counter for network model loop

i_mm

(*integer*) iteration counter for math model loop

success

(*integer*) `success` (page 11) flag, 1 - math model solved, 0 - didn't solve

message

(*char array*) output `message` (page 11)

et

(*double*) elapsed time (seconds) for `run()` (page 12) method

Method Summary

load_dm(d, mpopt, mpx)

Load the data model.

```
task.load_dm(d)
task.load_dm(d, mpopt)
task.load_dm(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of *mp.extension* (page 174)) – MATPOWER Extensions

Output

task (*mp.task* (page 9)) – task object containing the newly loaded data model converter and data model objects, in `task.dmc` and `task.dm`, respectively.

Create the data model converter and the data model object.

See the `sec_task` section in the *MATPOWER Developer's Manual* for more information.

run(*d, mpopt, mpx*)

Execute the task.

```
task.run(d)
task.run(d, mpopt)
task.run(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`) **or** data model object
- **mpopt** (*struct*) – (*optional*) MATPOWER options struct
- **mpx** (cell array of *mp.extension* (page 174)) – (*optional*) MATPOWER Extensions

Output

task (*mp.task* (page 9)) – task object containing the solved `run()` (page 12) including the data, network, and mathematical model objects.

Execute the task, creating the data model converter and the data, network and mathematical model objects, solving the math model and propagating the solution back to the data model.

See the `sec_task` section in the *MATPOWER Developer's Manual* for more information.

next_mm(*mm, nm, dm, mpopt, mpx*)

Controls iterations over mathematical models.

```
[mm, nm, dm] = task.next_mm(mm, nm, dm, mpopt, mpx)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of *mp.extension* (page 174)) – MATPOWER Extensions

Output

- **mm** (*mp.math_model* (page 124)) – new or updated mathematical model object, or empty matrix
- **nm** (*mp.net_model* (page 93)) – potentially updated network model object
- **dm** (*mp.data_model* (page 30)) – potentially updated data model object

Called automatically by `run()` (page 12) method. Subclasses can override this method to return a new or updated math model object for use in the next iteration or an empty matrix (the default) if finished.

next_nm(*mm, nm, dm, mpopt, mpx*)

Controls iterations over network models.

```
[nm, dm] = task.next_nm(mm, nm, dm, mpopt, mpx)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of *mp.extension* (page 174)) – MATPOWER Extensions

Output

- **nm** (*mp.net_model* (page 93)) – new or updated network model object, or empty matrix
- **dm** (*mp.data_model* (page 30)) – potentially updated data model object

Called automatically by *run()* (page 12) method. Subclasses can override this method to return a new or updated network model object for use in the next iteration or an empty matrix (the default) if finished.

next_dm(*mm, nm, dm, mpopt, mpx*)

Controls iterations over data models.

```
dm = task.next_dm(mm, nm, dm, mpopt, mpx)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of *mp.extension* (page 174)) – MATPOWER Extensions

Output

dm (*mp.data_model* (page 30)) – new or updated data model object, or empty matrix

Called automatically by *run()* (page 12) method. Subclasses can override this method to return a new or updated data model object for use in the next iteration or an empty matrix (the default) if finished.

run_pre(*d, mpopt*)

Called at beginning of *run()* (page 12) method.

```
[d, mpopt] = task.run_pre(d, mpopt)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (*mpc*), or data model object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **d** – updated value of corresponding input
- **mpopt** (*struct*) – updated value of corresponding input

Subclasses can override this method to update the input data or options before beginning the run.

run_post(*mm, nm, dm, mpopt*)

Called at end of *run()* (page 12) method.

```
task.run_post(mm, nm, dm, mpopt)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task (*mp.task* (page 9)) – task object

Subclasses can override this method to do any final processing after the run is complete.

print_soln(*mpopt, fname*)

Display the pretty-printed results.

```
task.print_soln(mpop)  
task.print_soln(mpop, fname)
```

Inputs

- **mpopt** (*struct*) – MATPOWER options struct
- **fname** (*char array*) – file name for saving pretty-printed output

Display to standard output and/or save to a file the pretty-printed solved case.

print_soln_header(*mpopt, fd*)

Display solution header information.

```
task.print_soln_header(mpop, fd)
```

Inputs

- **mpopt** (*struct*) – MATPOWER options struct
- **fd** (*integer*) – file identifier (1 for standard output)

Called by [print_soln\(\)](#) (page 14) to print success/failure, elapsed time, etc. to a file identifier.

save_soln(*fname*)

Save the solved case to a file.

```
task.save_soln(fname)
```

Input

fname (*char array*) – file name for saving solved case

dm_converter_class(*d, mpopt, mpx*)

Get data model converter constructor.

```
dmc_class = task.dm_converter_class(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (*mpc*)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dmc_class (*function handle*) – handle to the constructor to be used to instantiate the data model converter object

Called by [dm_converter_create\(\)](#) (page 15) to determine the class to use for the data model converter object. Handles any modifications specified by MATPOWER options or extensions.

dm_converter_class_mpc2_default()

Get default data model converter constructor.

```
dmc_class = task.dm_converter_class_mpc2_default()
```

Output

dmc_class (*function handle*) – handle to default constructor to be used to instantiate the data model converter object

Called by [dm_converter_class\(\)](#) (page 14) to determine the default class to use for the data model converter object when the input is a version 2 MATPOWER case struct.

dm_converter_create(*d, mpopt, mpx*)

Create data model converter object.

```
dmc = task.dm_converter_create(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dmc ([mp.dm_converter](#) (page 62)) – data model converter object, ready to build

Called by [dm_converter_build\(\)](#) (page 15) method to instantiate the data model converter object. Handles any modifications to data model converter elements specified by MATPOWER options or extensions.

dm_converter_build(*d, mpopt, mpx*)

Create and build data model converter object.

```
dmc = task.dm_converter_build(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dmc ([mp.dm_converter](#) (page 62)) – data model converter object, ready for use

Called by [run\(\)](#) (page 12) method to instantiate and build the data model converter object, including any modifications specified by MATPOWER options or extensions.

data_model_class(*d, mpopt, mpx*)

Get data model constructor.

```
dm_class = task.data_model_class(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dm_class (*function handle*) – handle to the constructor to be used to instantiate the data model object

Called by [data_model_create\(\)](#) (page 16) to determine the class to use for the data model object. Handles any modifications specified by MATPOWER options or extensions.

data_model_class_default()

Get default data model constructor.

```
dm_class = task.data_model_class_default()
```

Output

dm_class (*function handle*) – handle to default constructor to be used to instantiate the data model object

Called by [data_model_class\(\)](#) (page 15) to determine the default class to use for the data model object.

data_model_create(d, mpopt, mpx)

Create data model object.

```
dm = task.data_model_create(d, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (*mpc*)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dm ([mp.data_model](#) (page 30)) – data model object, ready to build

Called by [data_model_build\(\)](#) (page 16) to instantiate the data model object. Handles any modifications to data model elements specified by MATPOWER options or extensions.

data_model_build(d, dmc, mpopt, mpx)

Create and build data model object.

```
dm = task.data_model_build(d, dmc, mpopt, mpx)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (*mpc*)
- **dmc** ([mp.dm_converter](#) (page 62)) – data model converter object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

dm ([mp.data_model](#) (page 30)) – data model object, ready for use

Called by [run\(\)](#) (page 12) method to instantiate and build the data model object, including any modifications specified by MATPOWER options or extensions.

data_model_build_pre(dm, d, dmc, mpopt)

Called at beginning of [data_model_build\(\)](#) (page 16).

```
[dm, d] = task.data_model_build_pre(dm, d, dmc, mpopt)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (*mpc*)
- **dmc** ([mp.dm_converter](#) (page 62)) – data model converter object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **dm** ([mp.data_model](#) (page 30)) – updated data model object
- **d** – updated value of corresponding input

Called just *before* calling the data model's build() method. In this base class, this method does nothing.

data_model_build_post(dm, dmc, mpopt)

Called at end of [data_model_build\(\)](#) (page 16).

```
dm = task.data_model_build_post(dm, dmc, mpopt)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **dmc** ([mp.dm_converter](#) (page 62)) – data model converter object
- **mpopt** (*struct*) – MATPOWER options struct

Output

dm ([mp.data_model](#) (page 30)) – updated data model object

Called just *after* calling the data model's build() method. In this base class, this method does nothing.

network_model_class(dm, mpopt, mpx)

Get network model constructor.

```
nm_class = task.network_model_class(dm, mpopt, mpx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

nm_class (*function handle*) – handle to the constructor to be used to instantiate the network model object

Called by [network_model_create\(\)](#) (page 17) to determine the class to use for the network model object. Handles any modifications specified by MATPOWER options or extensions.

network_model_class_default(dm, mpopt)

Get default network model constructor.

```
nm_class = task.network_model_class_default(dm, mpopt)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

nm_class (*function handle*) – handle to default constructor to be used to instantiate the network model object

Called by [network_model_class\(\)](#) (page 17) to determine the default class to use for the network model object.

Note: This is an abstract method that must be implemented by a subclass.

network_model_create(dm, mpopt, mpx)

Create network model object.

```
nm = task.network_model_create(dm, mpopt, mpx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

nm ([mp.net_model](#) (page 93)) – network model object, ready to build

Called by [network_model_build\(\)](#) (page 18) to instantiate the network model object. Handles any modifications to network model elements specified by MATPOWER options or extensions.

network_model_build(dm, mpopt, mpx)

Create and build network model object.

```
nm = task.network_model_build(dm, mpopt, mpx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

nm ([mp.net_model](#) (page 93)) – network model object, ready for use

Called by [run\(\)](#) (page 12) method to instantiate and build the network model object, including any modifications specified by MATPOWER options or extensions.

network_model_build_pre(nm, dm, mpopt)

Called at beginning of [network_model_build\(\)](#) (page 18).

```
nm = task.network_model_build_pre(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

nm ([mp.net_model](#) (page 93)) – updated network model object

Called just *before* calling the network model's `build()` method. In this base class, this method does nothing.

network_model_build_post(nm, dm, mpopt)

Called at end of [network_model_build\(\)](#) (page 18).

```
nm = task.network_model_build_post(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

nm ([mp.net_model](#) (page 93)) – updated network model object

Called just *after* calling the network model's `build()` method. In this base class, this method does nothing.

network_model_x_soln(mm, nm)

Update network model state from math model solution.

```
nm = task.network_model_x_soln(mm, nm)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object

Output

nm ([mp.net_model](#) (page 93)) – updated network model object

Called by [network_model_update\(\)](#) (page 19).

network_model_update(*mm, nm*)

Update network model state, solution values from math model solution.

```
nm = task.network_model_update(mm, nm)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object

Output

nm ([mp.net_model](#) (page 93)) – updated network model object

Called by [run\(\)](#) (page 12) method.

math_model_class(*nm, dm, mpopt, mpx*)

Get mathematical model constructor.

```
mm_class = task.math_model_class(nm, dm, mpopt, mpx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output

mm_class (*function handle*) – handle to the constructor to be used to instantiate the mathematical model object

Called by [math_model_create\(\)](#) (page 19) to determine the class to use for the mathematical model object. Handles any modifications specified by MATPOWER options or extensions.

math_model_class_default(*nm, dm, mpopt*)

Get default mathematical model constructor.

```
mm_class = task.math_model_class_default(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

mm_class (*function handle*) – handle to the constructor to be used to instantiate the mathematical model object

Called by [math_model_class\(\)](#) (page 19) to determine the default class to use for the mathematical model object.

Note: This is an abstract method that must be implemented by a subclass.

math_model_create(*nm, dm, mpopt, mpx*)

Create mathematical model object.

```
mm = task.math_model_create(nm, dm, mpopt, mpx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output**mm** ([mp.math_model](#) (page 124)) – mathematical model object, ready to build

Called by [math_model_build\(\)](#) (page 20) to instantiate the mathematical model object. Handles any modifications to mathematical model elements specified by MATPOWER options or extensions.

math_model_build(*nm, dm, mpopt, mpx*)

Create and build mathematical model object.

```
mm = task.math_model_build(nm, dm, mpopt, mpx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of [mp.extension](#) (page 174)) – MATPOWER Extensions

Output**mm** ([mp.math_model](#) (page 124)) – mathematical model object, ready for use

Called by [run\(\)](#) (page 12) method to instantiate and build the mathematical model object, including any modifications specified by MATPOWER options or extensions.

math_model_opt(*mm, nm, dm, mpopt*)Get the options struct to pass to `mm.solve()`.

```
opt = task.math_model_opt(mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output**opt** (*struct*) – options struct for mathematical model `solve()` method

Called by [run\(\)](#) (page 12) method.

mp.task_pf**class mp.task_pf**

Bases: [mp.task](#) (page 9)

[mp.task_pf](#) (page 21) - MATPOWER task for power flow (PF).

Provides task implementation for the power flow problem.

This includes the handling of iterative runs to enforce generator reactive power limits, if requested.

mp.task_pf Properties:

- [tag](#) (page 21) - task tag 'PF'
- [name](#) (page 21) - task name 'Power Flow'
- [dc](#) (page 21) - `true` if using DC network model
- [iterations](#) (page 21) - total number of power flow iterations
- [ref](#) (page 21) - current ref node indices
- [ref0](#) (page 21) - initial ref node indices
- [va_ref0](#) (page 21) - initial ref node voltage angles
- [fixed_q_idx](#) (page 22) - indices of fixed Q gens
- [fixed_q_qty](#) (page 22) - Q output of fixed Q gens

mp.task_pf Methods:

- [run_pre\(\)](#) (page 22) - set dc property
- [next_dm\(\)](#) (page 22) - optionally iterate to enforce generator reactive limits
- [enforce_q_lims\(\)](#) (page 22) - implementation of generator reactive limits
- [network_model_class_default\(\)](#) (page 22) - select default network model constructor
- [network_model_build_post\(\)](#) (page 22) - initialize properties for reactive limits
- [network_model_x_soln\(\)](#) (page 22) - correct the voltage angles if necessary
- [math_model_class_default\(\)](#) (page 22) - select default math model constructor

See also [mp.task](#) (page 9).

Property Summary

tag = 'PF'

name = 'Power Flow'

dc

`true` if using DC network model (from `mpopt.model`, cached in [run_pre\(\)](#) (page 22))

iterations

(*integer*) total number of power flow [iterations](#) (page 21)

ref

(*integer*) current [ref](#) (page 21) node indices

ref0

(*integer*) initial ref node indices

va_ref0

(double) initial ref node voltage angles

fixed_q_idx

(integer) indices of fixed Q gens

fixed_q_qty

(double) Q output of fixed Q gens

Method Summary

run_pre(*d*, *mpopt*)

Set dc property after calling superclass [run_pre\(\)](#) (page 13).

next_dm(*nm*, *nm*, *dm*, *mpopt*, *mpx*)

Implement optional iterations to enforce generator reactive limits.

enforce_q_lims(*nm*, *dm*, *mpopt*)

Used by [next_dm\(\)](#) (page 22) to implement enforcement of generator reactive limits.

network_model_class_default(*dm*, *mpopt*)

Implement selector for default network model constructor depending on `mpopt.model` and `mpopt.pf.v_cartesian`.

network_model_build_post(*nm*, *dm*, *mpopt*)

Initialize [mp.task_pf](#) (page 21) properties, if non-empty AC case with generator reactive limits enforced.

network_model_x_soln(*nm*, *nm*)

Call superclass [network_model_x_soln\(\)](#) (page 18) then correct the voltage angle if the ref node has been changed.

math_model_class_default(*nm*, *dm*, *mpopt*)

Implement selector for default mathematical model constructor depending on `mpopt.model`, `mpopt.pf.v_cartesian`, and `mpopt.pf.current_balance`.

mp.task_cpf

class mp.task_cpf

Bases: [mp.task_pf](#) (page 21)

[mp.task_cpf](#) (page 22) - MATPOWER task for continuation power flow (CPF).

Provides task implementation for the continuation power flow problem.

This includes the iterative solving of the mathematical model (using warm restarts) after updating the problem data, e.g. when enforcing certain limits.

mp.task_cpf Properties:

- [warmstart](#) (page 23) - warm start data

mp.task_cpf Methods:

- [task_cpf\(\)](#) (page 23) - constructor, inherits from [mp.task_pf](#) (page 21) constructor
- [run_pre\(\)](#) (page 23) - call superclass [run_pre\(\)](#) (page 22) for base and target inputs

- `next_mm()` (page 23) - handle warm start of continuation iterations
- `dm_converter_class()` (page 23) - select data model converter class
- `data_model_class_default()` (page 23) - select default data model constructor
- `data_model_build()` (page 23) - build base and target data models
- `network_model_build()` (page 23) - build base and target network models
- `network_model_x_soln()` (page 23) - update network model solution
- `network_model_update()` (page 23) - evaluate port injection solution
- `math_model_class_default()` (page 23) - select default math model constructor
- `math_model_opt()` (page 24) - add warmstart parameters to math model solve options

See also `mp.task` (page 9), `mp.task_pf` (page 21).

Constructor Summary

`task_cpf()`

Constructor, inherits from `mp.task_pf` (page 21) constructor.

Property Summary

`warmstart`

(*struct*) warm start data, with fields:

- `clam` - corrector parameter lambda
- `plam` - predictor parameter lambda
- `cV` - corrector complex voltage vector
- `pV` - predictor complex voltage vector

Method Summary

`run_pre(d, mpopt)`

Call superclass `run_pre()` (page 22) for base and target inputs.

`next_mm(mm, nm, dm, mpopt, mpx)`

Handle warm start of continuation iterations, after problem data update.

`dm_converter_class(d, mpopt, mpx)`

Implement selector for data model converter class based on superclass constructor.

`data_model_class_default()`

Implement selector for default data model constructor.

`data_model_build(d, dmc, mpopt, mpx)`

Call superclass `data_model_build()` for base and target models.

`network_model_build(dm, mpopt, mpx)`

Call superclass `network_model_build()` for base and target models.

`network_model_x_soln(mm, nm)`

Call superclass `network_model_x_soln()` (page 22) then update solution in target network model.

`network_model_update(mm, nm)`

Call superclass `network_model_update()` then update port injection solution by interpolating with parameter lambda.

math_model_class_default(*nm, dm, mpopt*)

Implement selector for default mathematical model constructor depending on `mpopt.pf.v_cartesian` and `mpopt.pf.current_balance`.

math_model_opt(*mm, nm, dm, mpopt*)

Call superclass `math_model_opt()` then add warmstart parameters, if available.

mp.task_opf

class mp.task_opf

Bases: [mp.task](#) (page 9)

[mp.task_opf](#) (page 24) - MATPOWER task for optimal power flow (OPF).

Provides task implementation for the optimal power flow problem.

mp.task_opf Properties:

- `tag` - task tag 'OPF'
- `name` - task name 'Optimal Power Flow'
- `dc` (page 24) - true if using DC network model

mp.task_opf Methods:

- [run_pre\(\)](#) (page 24) - set dc property
- [print_soln_header\(\)](#) (page 24) - add printout of objective function value
- [data_model_class_default\(\)](#) (page 24) - select default data model constructor
- [data_model_build_post\(\)](#) (page 24) - adjust bus voltage limits, if requested
- [network_model_class_default\(\)](#) (page 25) - select default network model constructor
- [math_model_class_default\(\)](#) (page 25) - select default math model constructor

See also [mp.task](#) (page 9).

Property Summary

dc

true if using DC network model (from `mpopt.model`, cached in [run_pre\(\)](#) (page 24))

Method Summary

run_pre(*d, mpopt*)

Set dc property after calling superclass [run_pre\(\)](#) (page 13), then check for unsupported AC OPF solver selection.

print_soln_header(*mpopt, fd*)

Call superclass [print_soln_header\(\)](#) (page 14) the print out the objective function value.

data_model_class_default()

Implement selector for default data model constructor.

data_model_build_post(*dm, dmc, mpopt*)

Call superclass [data_model_build_post\(\)](#) (page 17) then adjust bus voltage magnitude limits based on generator `vm_setpoint`, if requested.

network_model_class_default(*dm, mpopt*)

Implement selector for default network model constructor depending on `mpopt.model` and `mpopt.opf.v_cartesian`.

math_model_class_default(*nm, dm, mpopt*)

Implement selector for default mathematical model constructor depending on `mpopt.model`, `mpopt.opf.v_cartesian`, and `mpopt.opf.current_balance`.

3.1.2 Legacy Task Classes

Used by MP-Core when called by the *legacy MATPOWER framework*.

mp.task_pf_legacy

class mp.task_pf_legacy

Bases: [mp.task_pf](#) (page 21), [mp.task_shared_legacy](#) (page 29)

[mp.task_pf_legacy](#) (page 25) - MATPOWER task for legacy power flow (PF).

Adds functionality needed by the *legacy MATPOWER framework* to the task implementation for the power flow problem. This consists of pre-processing some input data and exporting and packaging result data.

mp.task_pf Methods:

- [run_pre\(\)](#) (page 25) - pre-process inputs that are for legacy framework only
- [run_post\(\)](#) (page 25) - export results back to data model source
- [legacy_post_run\(\)](#) (page 26) - post-process *legacy framework* outputs

See also [mp.task_pf](#) (page 21), [mp.task](#) (page 9), [mp.task_shared_legacy](#) (page 29).

Method Summary

run_pre(*d, mpopt*)

Pre-process inputs that are for *legacy framework* only.

```
[d, mpopt] = task.run_pre(d, mpopt)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`), **or** data model object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **d** – updated value of corresponding input
- **mpopt** (*struct*) – updated value of corresponding input

Call [run_pre_legacy\(\)](#) (page 29) method before calling parent.

run_post(*mm, nm, dm, mpopt*)

Export results back to data model source.

```
task.run_post(mm, nm, dm, mpopt)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task (*mp.task* (page 9)) – task object

Calls *mp.dm_converter.export()* (page 63) and saves the result in the data model source property.

legacy_post_run(*mpopt*)

Post-process *legacy framework* outputs.

```
[results, success] = task.legacy_post_run(mpop)
```

Input

mpopt (*struct*) – MATPOWER options struct

Outputs

- **results** (*struct*) – results struct for *legacy MATPOWER framework*, see Table 4.1 in *legacy MATPOWER User's Manual*.
- **success** (*integer*) – 1 - succeeded, 0 - failed

Extract results and success and save the task object in `results.task` before returning.

mp.task_cpf_legacy

class mp.task_cpf_legacy

Bases: *mp.task_cpf* (page 22), *mp.task_shared_legacy* (page 29)

mp.task_cpf (page 22) - MATPOWER task for legacy continuation power flow (CPF).

Adds functionality needed by the *legacy MATPOWER framework* to the task implementation for the continuation power flow problem. This consists of pre-processing some input data and exporting and packaging result data.

mp.task_pf Methods:

- *run_pre()* (page 26) - pre-process inputs that are for legacy framework only
- *run_post()* (page 27) - export results back to data model source
- *legacy_post_run()* (page 27) - post-process *legacy framework* outputs

See also *mp.task_cpf* (page 22), *mp.task* (page 9), *mp.task_shared_legacy* (page 29).

Method Summary

run_pre(*d, mpopt*)

Pre-process inputs that are for *legacy framework* only.

```
[d, mpopt] = task.run_pre(d, mpopt)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`), **or** data model object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **d** – updated value of corresponding input
- **mpopt** (*struct*) – updated value of corresponding input

Call [run_pre_legacy\(\)](#) (page 29) method for both input cases before calling parent.

run_post(*mm, nm, dm, mpopt*)

Export results back to data model source.

```
task.run_post(mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task ([mp.task](#) (page 9)) – task object

Calls [mp.dm_converter.export\(\)](#) (page 63) and saves the result in the data model source property.

legacy_post_run(*mpopt*)

Post-process *legacy framework* outputs.

```
[results, success] = task.legacy_post_run(mpop)
```

Input

mpopt (*struct*) – MATPOWER options struct

Outputs

- **results** (*struct*) – results struct for *legacy MATPOWER framework*, see Table 5.1 in [legacy MATPOWER User's Manual](#).
- **success** (*integer*) – 1 - succeeded, 0 - failed

Extract results and success and save the task object in `results.task` before returning.

mp.task_opf_legacy

class mp.task_opf_legacy

Bases: [mp.task_opf](#) (page 24), [mp.task_shared_legacy](#) (page 29)

[mp.task_opf](#) (page 24) - MATPOWER task for legacy optimal power flow (OPF).

Adds functionality needed by the *legacy MATPOWER framework* to the task implementation for the optimal power flow problem. This consists of pre-processing some input data and exporting and packaging result data, as well as using some legacy specific model sub-classes.

mp.task_pf Methods:

- [run_pre\(\)](#) (page 28) - pre-process inputs that are for legacy framework only
- [run_post\(\)](#) (page 28) - export results back to data model source

- `dm_converter_class_mpc2_default()` (page 28) - set to `mp.dm_converter_mpc2_legacy` (page 65)
- `data_model_build_post()` (page 28) - get data model converter to do more input pre-processing
- `math_model_class_default()` (page 28) - use legacy math model subclasses
- `legacy_post_run()` (page 28) - post-process *legacy framework* outputs

See also `mp.taskopf` (page 24), `mp.task` (page 9), `mp.task_shared_legacy` (page 29).

Method Summary

`run_pre(d, mpopt)`

Pre-process inputs that are for *legacy framework* only.

```
[d, mpopt] = task.run_pre(d, mpopt)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`), **or** data model object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **d** – updated value of corresponding input
- **mpopt** (*struct*) – updated value of corresponding input

Call `run_pre_legacy()` (page 29) method before calling parent.

`run_post(mm, nm, dm, mpopt)`

Export results back to data model source.

```
task.run_post(mm, nm, dm, mpopt)
```

Inputs

- **mm** (`mp.math_model` (page 124)) – mathematical model object
- **nm** (`mp.net_model` (page 93)) – network model object
- **dm** (`mp.data_model` (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task (`mp.task` (page 9)) – task object

Calls `mp.dm_converter.export()` (page 63) and saves the result in the data model source property.

`dm_converter_class_mpc2_default()`

Set to `mp.dm_converter_mpc2_legacy` (page 65).

```
dmc_class = task.dm_converter_class_mpc2_default()
```

`data_model_build_post(dm, dmc, mpopt)`

Get data model converter to do more input pre-processing after calling superclass `data_model_build_post()` (page 24).

`math_model_class_default(nm, dm, mpopt)`

Use legacy math model subclasses to support legacy costs and callbacks.

Uses math model variations that inherit from `mp.mmm_shared_opf_legacy` (page 145) (compatible with the legacy `opf_model` (page 223)), in order to support legacy cost functions and callback functions that expect to find the MATPOWER case struct in `mm.mpc`.

legacy_post_run(*mpopt*)

Post-process *legacy framework* outputs.

```
[results, success, raw] = task.legacy_post_run(mpop)
```

Input

mpopt (*struct*) – MATPOWER options struct

Outputs

- **results** (*struct*) – results struct for *legacy MATPOWER framework*, see Table 6.1 in [legacy MATPOWER User's Manual](#).
- **success** (*integer*) – 1 - succeeded, 0 - failed
- **raw** (*struct*) – see raw field in Table 6.1 in [legacy MATPOWER User's Manual](#).

Extract results and success and save the task object in `results.task` before returning. This method also creates and populates numerous other fields expected in the legacy OPF results struct, such as `f`, `x`, `om`, `mu`, `g`, `dg`, `raw`, `var`, `nle`, `nli`, `lin`, and `cost`. Based on code from the legacy functions [opf_execute\(\)](#) (page 295), [dcopf_solver\(\)](#) (page 289), and [nlpopf_solver\(\)](#) (page 290).

mp.task_shared_legacy**class mp.task_shared_legacy**

Bases: `handle`

[mp.task_shared_legacy](#) (page 29) - Shared legacy task functionality.

Provides legacy task functionality shared across different tasks (e.g. PF, CPF, OPF), specifically, the pre-processing of input data for the experimental system-wide ZIP load data.

mp.task_pf Methods:

- [run_pre_legacy\(\)](#) (page 29) - handle experimental system-wide ZIP load inputs

See also [mp.task](#) (page 9).

Method Summary**run_pre_legacy(*d*, *mpopt*)**

Handle experimental system-wide ZIP load inputs.

```
[d, mpopt] = task.run_pre_legacy(d, mpopt)
```

Inputs

- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`), **or** data model object
- **mpopt** (*struct*) – MATPOWER options struct

Outputs

- **d** – updated value of corresponding input
- **mpopt** (*struct*) – updated value of corresponding input

Moves the legacy experimental system-wide ZIP load data from `mpopt.exp.sys_wide_zip_loads` to `d.sys_wide_zip_loads` to make it available to the data model converter ([mp.dmce_load_mpc2](#) (page 73)).

Called by [run_pre\(\)](#) (page 13).

3.2 Data Model Classes

3.2.1 Containers

`mp.data_model`

`class mp.data_model`

Bases: `mp.element_container` (page 168)

`mp.data_model` (page 30) - Base class for MATPOWER **data model** objects.

The data model object encapsulates the input data provided by the user for the problem of interest and the output data presented back to the user upon completion. It corresponds roughly to the `mpc` (MATPOWER case) and `results` structs used throughout the legacy MATPOWER implementation, but encapsulated in an object with additional functionality. It includes tables of data for each type of element in the system.

A data model object is primarily a container for data model element (`mp.dm_element` (page 38)) objects. Concrete data model classes may be specific to the task.

By convention, data model variables are named `dm` and data model class names begin with `mp.data_model`.

`mp.data_model` Properties:

- `base_mva` (page 31) - system per unit MVA base
- `base_kva` (page 31) - system per unit kVA base
- `source` (page 31) - source of data, e.g. `mpc` (MATPOWER case struct)
- `userdata` (page 31) - arbitrary user data

`mp.data_model` Methods:

- `data_model()` (page 31) - constructor, assign default data model element classes
- `copy()` (page 31) - make duplicate of object
- `build()` (page 31) - create, add, and build element objects
- `count()` (page 32) - count instances of each element and remove if count is zero
- `initialize()` (page 32) - initialize (online/offline) status of each element
- `update_status()` (page 32) - update (online/offline) status based on connectivity, etc
- `build_params()` (page 32) - extract/convert/calculate parameters for online elements
- `rebuild()` (page 32) - rebuild object, calling `count()` (page 32), `initialize()` (page 32), `update_status()` (page 32), `build_params()` (page 32)
- `online()` (page 33) - get number of online elements of named type
- `display()` (page 33) - display the data model object
- `pretty_print()` (page 33) - pretty print data model to console or file
- `pp_flags()` (page 33) - from options, build flags to control pretty printed output
- `pp_section_label()` (page 34) - construct section header lines for output
- `pp_section_list()` (page 34) - return list of section tags
- `pp_have_section()` (page 34) - return true if section exists for object

- [pp_section\(\)](#) (page 35) - pretty print the given section
- [pp_get_headers\(\)](#) (page 35) - construct pretty printed lines for section headers
- [pp_get_headers_cnt\(\)](#) (page 35) - construct pretty printed lines for **cnt** section headers
- [pp_get_headers_ext\(\)](#) (page 35) - construct pretty printed lines for **ext** section headers
- [pp_data\(\)](#) (page 36) - pretty print the data for the given section
- [set_bus_v_lims_via_vg\(\)](#) (page 36) - set gen bus voltage limits based on gen voltage setpoints

See the `sec_data_model` section in the *MATPOWER Developer's Manual* for more information.

See also [mp.task](#) (page 9), [mp.net_model](#) (page 93), [mp.math_model](#) (page 124), [mp.dm_converter](#) (page 62).

Constructor Summary

data_model()

Constructor, assign default data model element classes.

```
dm = mp.data_model()
```

Property Summary

base_mva

(*double*) system per unit MVA base, for balanced single-phase systems/sections, must be provided if system includes any 'bus' elements

base_kva

(*double*) system per unit kVA base, for unbalanced 3-phase systems/sections, must be provided if system includes any 'bus3p' elements

source

[source](#) (page 31) of data, e.g. `mpc` (MATPOWER case struct)

userdata = struct()

(*struct*) arbitrary user data

Method Summary

copy()

Create a duplicate of the data model object, calling the [copy\(\)](#) (page 43) method on each element.

```
new_dm = dm.copy()
```

build(d, dmc)

Create and add data model element objects.

```
dm.build(d, dmc)
```

Inputs

- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))
- **dmc** ([mp.dm_converter](#) (page 62)) – data model converter

Create the data model element objects by instantiating each class in the [element_classes](#) (page 168) property and adding the resulting object to the [elements](#) (page 168) property. Then proceed through the following additional [build\(\)](#) (page 31) stages for each element.

- Import
- Count
- Initialize
- Update status
- Build parameters

See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

count()

Count instances of each element and remove if `count()` (page 32) is zero.

```
dm.count()
```

Call each element's `count()` (page 43) method to determine the number of instances of that element in the data, and remove the element type from `elements` (page 168) if the count is 0.

Called by `build()` (page 31) to perform its **count** stage. See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

initialize()

Initialize (online/offline) status of each element.

```
dm.initialize()
```

Call each element's `initialize()` (page 43) method to `initialize()` (page 32) statuses and create ID to row index mappings.

Called by `build()` (page 31) to perform its **initialize** stage. See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

update_status()

Update (online/offline) status based on connectivity, etc.

```
dm.update_status()
```

Call each element's `update_status()` (page 44) method to update statuses based on connectivity or other criteria and define element properties containing number and row indices of online elements, indices of offline elements, and mapping of row indices to indices in online and offline element lists.

Called by `build()` (page 31) to perform its **update status** stage. See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

build_params()

Extract/convert/calculate parameters for online elements.

```
dm.build_params()
```

Call each element's `build_params()` (page 44) method to build parameters as necessary for online elements from the original data tables (e.g. p.u. conversion, initial state, etc.) and store them in element-specific properties.

Called by `build()` (page 31) to perform its **build parameters** stage. See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* more information.

rebuild()

Rebuild object, calling `count()` (page 32), `initialize()` (page 32), `update_status()` (page 32), `build_params()` (page 32).

```
dm.rebuild()
```

Typically used after modifying data in the main tables of one or more of the elements.

online(*name*)

Get number of online elements of named type.

```
n = dm.online(name)
```

Input

name (*char array*) – name of element type (e.g. 'bus', 'gen') as returned by the element's [name\(\)](#) (page 40) method

Output

n (*integer*) – number of online elements

display()

Display the data model object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the data model elements.

pretty_print(*mpopt*, *fd*)

Pretty print data model to console or file.

```
dm.pretty_print(mpop)
dm.pretty_print(mpop, fd)
[dm, out] = dm.pretty_print(mpop, fd)
```

Inputs

- **mpopt** (*struct*) – MATPOWER options struct
- **fd** (*integer*) – (*optional, default = 1*) file identifier to use for printing, (1 for standard output, 2 for standard error)

Outputs

- **dm** ([mp.data_model](#) (page 30)) – the data model object
- **out** (*struct*) – struct of output control flags

Displays the model parameters to a pretty-printed text format. The result can be output either to the console or to a file.

The output is organized into sections and each element type controls its own output for each section.

The default sections are:

- **cnt** - counts, number of online, offline, and total elements of this type
- **sum** - summary, e.g. total amount of capacity, load, line loss, etc.
- **ext** - extremes, e.g. min and max voltages, nodal prices, etc.
- **det** - details, table of detailed data, e.g. voltages, prices for buses, dispatch, limits for generators, etc.

pp_flags(*mpopt*)

From options, build flags to control pretty printed output.

```
[out, add] = dm.pp_flags(mpop)
```

Input

mpopt (*struct*) – MATPOWER options struct

Outputs

- **out** (*struct*) – struct of output control flags

```

out
  .all      (-1, 0 or 1)
  .any      (0 or 1)
  .sec
    .cnt
      .all      (-1, 0 or 1)
      .any      (0 or 1)
      .sum      (same as cnt)
      .ext      (same as cnt)
      .det
        .all      (-1, 0 or 1)
        .any      (0 or 1)
        .elm
          .<name>    (0 or 1)

```

where <name> is the name of the corresponding element type.

- **add** (*struct*) – additional data for subclasses to use

```

add
  .s0
    .<name> = 0
  .s1
    .<name> = 1
  .suppress      (-1, 0 or 1)
  .names         (cell array of element names)
  .ne            (number of element names)

```

See also [pretty_print\(\)](#) (page 33).

pp_section_label(*label, blank_line*)

Construct pretty printed lines for section label.

```
h = dm.pp_section_label(label, blank_line)
```

Inputs

- **label** (*char array*) – label for the section header
- **blank_line** (*boolean*) – include a blank line before the section label if true

Output

h (*cell array of char arrays*) – individual lines of section label

See also [pretty_print\(\)](#) (page 33).

pp_section_list(*out*)

Return list of section tags.

```
sections = dm.pp_section_list(out)
```

Input

out (*struct*) – struct of output control flags (see [pp_flags\(\)](#) (page 33) for details)

Output

sections (*cell array of char arrays*) – e.g. {'cnt', 'sum', 'ext', 'det'}

See also [pretty_print\(\)](#) (page 33).

pp_have_section(*section*, *mpopt*)

Return true if section exists for object with given options.

```
TorF = dm.pp_have_section(section, mpopt)
```

Inputs

- **section** (*char array*) – e.g. 'cnt', 'sum', 'ext', or 'det'
- **mpopt** (*struct*) – MATPOWER options struct

Output

TorF (*boolean*) – true if section exists

See also [pretty_print\(\)](#) (page 33).

pp_section(*section*, *out_s*, *mpopt*, *fd*)

Pretty print the given section.

```
dm.pp_section(section, out_s, mpopt, fd)
```

Inputs

- **section** (*char array*) – e.g. 'cnt', 'sum', 'ext', or 'det'
- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec(section)`
- **mpopt** (*struct*) – MATPOWER options struct
- **fd** (*integer*) – (*optional, default = 1*) file identifier to use for printing, (1 for standard output, 2 for standard error)

See also [pretty_print\(\)](#) (page 33).

pp_get_headers(*section*, *out_s*, *mpopt*)

Construct pretty printed lines for section headers.

```
h = dm.pp_get_headers(section, out_s, mpopt)
```

Inputs

- **section** (*char array*) – e.g. 'cnt', 'sum', 'ext', or 'det'
- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec(section)`
- **mpopt** (*struct*) – MATPOWER options struct

Output

h (*cell array of char arrays*) – individual lines of section headers

See also [pretty_print\(\)](#) (page 33).

pp_get_headers_cnt(*out_s*, *mpopt*)

Construct pretty printed lines for **cnt** section headers.

```
h = dm.pp_get_headers_cnt(out_s, mpopt)
```

Inputs

- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec(section)`
- **mpopt** (*struct*) – MATPOWER options struct

Output

h (*cell array of char arrays*) – individual lines of **cnt** section headers

See also [pretty_print\(\)](#) (page 33), [pp_get_headers\(\)](#) (page 35).

pp_get_headers_ext(*out_s*, *mpopt*)

Construct pretty printed lines for **ext** section headers.

```
h = dm.pp_get_headers_cnt(out_s, mpopt)
```

Inputs

- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec.(section)`
- **mpopt** (*struct*) – MATPOWER options struct

Output

h (*cell array of char arrays*) – individual lines of **ext** section headers

See also [pretty_print\(\)](#) (page 33), [pp_get_headers\(\)](#) (page 35).

pp_get_headers_other(*section, out_s, mpopt*)

Construct pretty printed lines for other section headers.

Returns nothing in base class, but subclasses can implement other section types (e.g. 'lim' for OPF).

```
h = dm.pp_get_headers_other(section, out_s, mpopt)
```

Inputs

- **section** (*char array*) – e.g. 'cnt', 'sum', 'ext', or 'det'
- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec.(section)`
- **mpopt** (*struct*) – MATPOWER options struct

Output

h (*cell array of char arrays*) – individual lines of **ext** section headers

See also [pretty_print\(\)](#) (page 33), [pp_get_headers\(\)](#) (page 35).

pp_data(*section, out_s, mpopt, fd*)

Pretty print the data for the given section.

```
dm.pp_data(section, out_s, mpopt, fd)
```

Inputs

- **section** (*char array*) – e.g. 'cnt', 'sum', 'ext', or 'det'
- **out_s** (*struct*) – output control flags for the section, `out_s = out.sec.(section)`
- **mpopt** (*struct*) – MATPOWER options struct
- **fd** (*integer*) – (*optional, default = 1*) file identifier to use for printing, (1 for standard output, 2 for standard error)

See also [pretty_print\(\)](#) (page 33), [pp_section\(\)](#) (page 35).

set_bus_v_lims_via_vg(*use_vg*)

Set gen bus voltage limits based on gen voltage setpoints.

```
dm.set_bus_v_lims_via_vg(use_vg)
```

Input

use_vg (*double*) – 1 if voltage setpoint should be used, 0 for original bus voltage bounds, or fractional value between 0 and 1 for bounds interpolated between the two.

mp.data_model_cpf

class mp.data_model_cpf

Bases: [mp.data_model](#) (page 30)

[mp.data_model_cpf](#) (page 37) - MATPOWER **data model** for CPF tasks.

The purpose of this class is to include CPF-specific subclasses for the load and shunt elements, which need to be able to provide versions of their model parameters that are parameterized by the continuation parameter λ .

data_model_cpf Methods:

- [data_model_cpf\(\)](#) (page 37) - constructor, assign default data model element classes

See also [mp.data_model](#) (page 30).

Constructor Summary

data_model_cpf()

Constructor, assign default data model element classes.

Create an empty data model object and assign the default data model element classes, which are the same as those defined by the base class, except for loads and shunts.

```
dm = mp.data_model_cpf()
```

mp.data_model_opf

class mp.data_model_opf

Bases: [mp.data_model](#) (page 30)

[mp.data_model_opf](#) (page 37) - MATPOWER **data model** for OPF tasks.

The purpose of this class is to include OPF-specific subclasses for its elements and to handle pretty-printing output for **lim** sections.

mp.data_model_opf Methods:

- [data_model_opf\(\)](#) (page 37) - constructor, assign default data model element classes
- [pp_flags\(\)](#) (page 37) - add flags for **lim** sections
- [pp_section_list\(\)](#) (page 38) - append 'lim' tag for **lim** sections to default list
- [pp_get_headers_other\(\)](#) (page 38) - construct headers for **lim** section headers

See also [mp.data_model](#) (page 30).

Constructor Summary

data_model_opf()

Constructor, assign default data model element classes.

Create an empty data model object and assign the default data model element classes, each specific to OPF.

```
dm = mp.data_model_opf()
```

Method Summary

pp_flags(*mpopt*)

Add flags for **lim** sections.

See [mp.data_model.pp_flags\(\)](#) (page 33).

pp_section_list(*out*)

Append 'lim' tag for **lim** section to default list.

See [mp.data_model.pp_section_list\(\)](#) (page 34).

pp_get_headers_other(*section, out_s, mpo**pt*)

Construct pretty printed lines for **lim** section headers.

See [mp.data_model.pp_get_headers_other\(\)](#) (page 36).

3.2.2 Elements

mp.dm_element

class `mp.dm_element`

Bases: `handle`

[mp.dm_element](#) (page 38) - Abstract base class for MATPOWER **data model element** objects.

A data model element object encapsulates all of the input and output data for a particular element type. All data model element classes inherit from [mp.dm_element](#) (page 38) and each element type typically implements its own subclass. A given data model element object contains the data for all instances of that element type, stored in one or more table data structures.

Defines the following columns in the main data table, which are inherited by all subclasses:

Name	Type	Description
<code>uid</code>	<i>integer</i>	unique ID
<code>name</code>	<i>char</i> <i>array</i>	element name
<code>status</code>	<i>boolean</i>	true = online, false = offline
<code>source_uid</code>	<i>unde-</i> <i>fined</i>	intended for any info required to link back to element instance in source data

By convention, data model element variables are named `dme` and data model element class names begin with `mp.dme`.

In addition to being containers for the data itself, data model elements are responsible for handling the on/off status of each element, preparation of parameters needed by network and mathematical models, definition of connections with other elements, defining solution data to be updated when exporting, and pretty-printing of data to the console or file.

Elements that create nodes (e.g. buses) are called **junction** elements. Elements that define ports (e.g. generators, branches, loads) can connect the ports of a particular instance to the nodes of a particular instance of a junction element by specifying two pieces of information for each port:

- the **type** of junction element it connects to

- the **index** of the specific junction element

mp.dm_element Properties:

- *tab* (page 40) - main data table
- *nr* (page 40) - total number of rows in table
- *n* (page 40) - number of online elements
- *ID2i* (page 40) - max(ID) x 1 vector, maps IDs to row indices
- *on* (page 40) - n x 1 vector of row indices of online elements
- *off* (page 40) - (nr-n) x 1 vector of row indices of offline elements
- *i2on* (page 40) - nr x 1 vector mapping row index to index in on/off respectively

mp.dm_element Methods:

- *name()* (page 40) - get name of element type, e.g. 'bus', 'gen'
- *label()* (page 41) - get singular label for element type, e.g. 'Bus', 'Generator'
- *labels()* (page 41) - get plural label for element type, e.g. 'Buses', 'Generators'
- *cxn_type()* (page 41) - type(s) of junction element(s) to which this element connects
- *cxn_idx_prop()* (page 41) - name(s) of property(ies) containing indices of junction elements
- *cxn_type_prop()* (page 41) - name(s) of property(ies) containing types of junction elements
- *table_exists()* (page 42) - check for existence of data in main data table
- *main_table_var_names()* (page 42) - names of variables (columns) in main data table
- *export_vars()* (page 42) - names of variables to be exported by DMCE to data source
- *export_vars_offline_val()* (page 42) - values of export variables for offline elements
- *dm_converter_element()* (page 43) - get corresponding data model converter element
- *copy()* (page 43) - create a duplicate of the data model element object
- *count()* (page 43) - determine number of instances of this element in the data
- *initialize()* (page 43) - initialize (online/offline) status of each element
- *ID()* (page 44) - return unique ID's for all or indexed rows
- *init_status()* (page 44) - initialize status column
- *update_status()* (page 44) - update (online/offline) status based on connectivity, etc
- *build_params()* (page 44) - extract/convert/calculate parameters for online elements
- *rebuild()* (page 45) - rebuild object, calling *count()* (page 43), *initialize()* (page 43), *update_status()* (page 44), *build_params()* (page 44)
- *display()* (page 45) - display the data model element object
- *pretty_print()* (page 45) - pretty-print data model element to console or file
- *pp_have_section()* (page 45) - true if pretty-printing for element has specified section
- *pp_rows()* (page 45) - indices of rows to include in pretty-printed output
- *pp_get_headers()* (page 46) - get pretty-printed headers for this element/section
- *pp_get_footers()* (page 46) - get pretty-printed footers for this element/section

- `pp_data()` (page 46) - pretty-print the data for this element/section
- `pp_have_section_cnt()` (page 46) - true if pretty-printing for element has **counts** section
- `pp_data_cnt()` (page 47) - pretty-print the **counts** data for this element
- `pp_have_section_sum()` (page 47) - true if pretty-printing for element has **summary** section
- `pp_data_sum()` (page 47) - pretty-print the **summary** data for this element
- `pp_have_section_ext()` (page 47) - true if pretty-printing for element has **extremes** section
- `pp_data_ext()` (page 47) - pretty-print the **extremes** data for this element
- `pp_have_section_det()` (page 47) - true if pretty-printing for element has **details** section
- `pp_get_title_det()` (page 47) - get title of **details** section for this element
- `pp_get_headers_det()` (page 48) - get pretty-printed **details** headers for this element
- `pp_get_footers_det()` (page 48) - get pretty-printed **details** footers for this element
- `pp_data_det()` (page 48) - pretty-print the **details** data for this element
- `pp_data_row_det()` (page 48) - get pretty-printed row of **details** data for this element

See the `sec_dm_element` section in the *MATPOWER Developer's Manual* for more information.

See also `mp.data_model` (page 30).

Property Summary

tab

(*table*) main data table

nr

(*integer*) total number of rows in table

n

(*integer*) number of online elements

ID2i

(*integer*) max(ID) x 1 vector, maps IDs to row indices

on

(*integer*) n x 1 vector of row indices of online elements

off

(*integer*) (nr-n) x 1 vector of row indices of offline elements

i2on

(*integer*) nr x 1 vector mapping row index to index in on/off respectively

Method Summary

name()

Get name of element type, e.g. 'bus', 'gen'.

```
name = dme.name()
```

Output

name (*char array*) – name of element type, must be a valid struct field name

Implementation provided by an element type specific subclass.

label()

Get singular label for element type, e.g. 'Bus', 'Generator'.

```
label = dme.label()
```

Output

label (*char array*) – user-visible label for element type, when singular

Implementation provided by an element type specific subclass.

labels()

Get plural label for element type, e.g. 'Buses', 'Generators'.

```
label = dme.labels()
```

Output

label (*char array*) – user-visible label for element type, when plural

Implementation provided by an element type specific subclass.

cxn_type()

Type(s) of junction element(s) to which this element connects.

```
name = dme.cxn_type()
```

Output

name (*char array or cell array of char arrays*) – name(s) of type(s) of junction elements, i.e. node-creating elements (e.g. 'bus'), to which this element connects

Assuming an element with *nc* connections, there are three options for the return value:

1. Single char array with one type that applies to all connections, [cxn_type_prop\(\)](#) (page 41) returns *empty*.
2. Cell array with *nc* elements, one for each connection, [cxn_type_prop\(\)](#) (page 41) returns *empty*.
3. Cell array of valid junction element types, [cxn_type_prop\(\)](#) (page 41) return value *not empty*.

See the `sec_dm_element_cxn` section in the *MATPOWER Developer's Manual* for more information.

Implementation provided by an element type specific subclass.

See also [cxn_idx_prop\(\)](#) (page 41), [cxn_type_prop\(\)](#) (page 41).

cxn_idx_prop()

Name(s) of property(ies) containing indices of junction elements.

```
name = dme.cxn_idx_prop()
```

Output

name (*char array or cell array of char arrays*) – name(s) of property(ies) containing indices of junction elements that define connections (e.g. {'fbus', 'tbus'})

See the `sec_dm_element_cxn` section in the *MATPOWER Developer's Manual* for more information.

Implementation provided by an element type specific subclass.

See also [cxn_type\(\)](#) (page 41), [cxn_type_prop\(\)](#) (page 41).

cxn_type_prop()

Name(s) of property(ies) containing types of junction elements.

```
name = dme.cxn_type_prop()
```

Output

name (*char array or cell array of char arrays*) – name(s) of properties containing type of junction elements for each connection

Note: If not empty, dimension must match [cxn_idx_prop\(\)](#) (page 41)

This is only used if the junction element type can vary by individual element, e.g. some elements of this type connect to one kind of bus, some to another kind. Otherwise, it returns an empty string and the junction element types for the connections are determined solely by [cxn_type\(\)](#) (page 41).

See the `sec_dm_element_cxn` section in the *MATPOWER Developer's Manual* for more information.

Implementation provided by an element type specific subclass.

See also [cxn_type\(\)](#) (page 41), [cxn_idx_prop\(\)](#) (page 41).

table_exists()

Check for existence of data in main data table.

```
TorF = dme.table_exists()
```

Output

TorF (*boolean*) – true if main data table is not empty

main_table_var_names()

Names of variables (columns) in main data table.

```
names = dme.main_table_var_names()
```

Output

names (*cell array of char arrays*) – names of variables (columns) in main table

This base class includes the following variables {'uid', 'name', 'status', 'source_uid'} which are common to all element types and should therefore be included in all subclasses. That is, subclass methods should append their additional fields to those returned by this parent method. For example, a subclass method would like something like the following:

```
function names = main_table_var_names(obj)
    names = horzcat( main_table_var_names@mp.dm_element(obj), ...
        {'subclass_var1', 'subclass_var2'} );
end
```

export_vars()

Names of variables to be exported by DMCE to data source.

```
vars = dme.export_vars()
```

Output

vars (*cell array of char arrays*) – names of variables to export

Return the names of the variables the data model converter element needs to export to the data source. This is typically the list of variables updated by the solution process, e.g. bus voltages, line flows, etc.

export_vars_offline_val()

Values of export variables for offline elements.

```
s = dme.export_vars_offline_val()
```

Output

s (*struct*) – keys are export variable names, values are the corresponding values to assign to these variables for offline elements.

Returns a struct defining the values of export variables for offline elements. Called by `mp.mm_element.data_model_update()` (page 148) to define how to set export variables for offline elements.

Export variables not found in the struct are not modified.

For example, `s = struct('va', 0, 'vm', 1)` would assign the value 0 to the `va` variable and 1 to the `vm` variable for any offline elements.

See also `export_vars()` (page 42).

dm_converter_element(dmc, name)

Get corresponding data model converter element.

```
dmce = dme.dm_converter_element(dmc)
dmce = dme.dm_converter_element(dmc, name)
```

Inputs

- **dmc** (`mp.dm_converter` (page 62)) – data model converter object
- **name** (*char array*) – (*optional*) name of element type (*default is name of this object*)

Output

dmce (`mp.dmc_element` (page 65)) – data model converter element object

copy()

Create a duplicate of the data model element object.

```
new_dme = dme.copy()
```

Output

new_dme (`mp.dm_element` (page 38)) – `copy()` (page 43) of data model element object

count(dm)

Determine number of instances of this element in the data.

Store the count in the `nr` property.

```
nr = dme.count(dm);
```

Input

dm (`mp.data_model` (page 30)) – data model

Output

nr (*integer*) – number of instances (rows of data)

Called for each element by the `count()` (page 32) method of `mp.data_model` (page 30) during the `count` stage of a data model build.

See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

initialize(dm)

Initialize a newly created data model element object.

```
dme.initialize(dm)
```

Input

dm (`mp.data_model` (page 30)) – data model

Initialize the (online/offline) status of each element and create a mapping of ID to row index in the ID2i element property, then call `init_status()` (page 44).

Called for each element by the `initialize()` (page 32) method of `mp.data_model` (page 30) during the **initialize** stage of a data model build.

See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

ID(*idx*)

Return unique ID's for all or indexed rows.

```
uid = dme.ID()
uid = dme.ID(idx)
```

Input

idx (*integer*) – (*optional*) row index vector

Return an *nr* x 1 vector of unique IDs for all rows, i.e. a map of row index to unique ID or, if a row index vector is provided just the ID's of the indexed rows.

init_status(*dm*)

Initialize status column.

```
dme.init_status(dm)
```

Input

dm (`mp.data_model` (page 30)) – data model

Called by `initialize()` (page 43). Does nothing in the base class.

update_status(*dm*)

Update (online/offline) status based on connectivity, etc.

```
dme.update_status(dm)
```

Input

dm (`mp.data_model` (page 30)) – data model

Update status of each element based on connectivity or other criteria and define element properties containing number and row indices of online elements (*n* and *on*), indices of offline elements (*off*), and mapping (*i2on*) of row indices to corresponding entries in *on* or *off*.

Called for each element by the `update_status()` (page 32) method of `mp.data_model` (page 30) during the **update status** stage of a data model build.

See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

build_params(*dm*)

Extract/convert/calculate parameters for online elements.

```
dme.build_params(dm)
```

Input

dm (`mp.data_model` (page 30)) – data model

Extract/convert/calculate parameters as necessary for online elements from the original data tables (e.g. p.u. conversion, initial state, etc.) and store them in element-specific properties.

Called for each element by the `build_params()` (page 32) method of `mp.data_model` (page 30) during the **build parameters** stage of a data model build.

See the `sec_building_data_model` section in the *MATPOWER Developer's Manual* for more information.

Does nothing in the base class.

rebuild(dm)

Rebuild object, calling `count()` (page 43), `initialize()` (page 43), `update_status()` (page 44), `build_params()` (page 44).

```
dme.rebuild(dm)
```

Input

dm (*mp.data_model* (page 30)) – data model

Typically used after modifying data in the main table.

display()

Display the data model element object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the elements, including total number of rows, number of online elements, and the main data table.

pretty_print(dm, section, out_e, mpopt, fd, pp_args)

Pretty print data model element to console or file.

```
dme.pretty_print(dm, section, out_e, mpopt, fd, pp_args)
```

Inputs

- **dm** (*mp.data_model* (page 30)) – data model
- **section** (*char array*) – section identifier, e.g. 'cnt', 'sum', 'ext', or 'det', for **counts**, **summary**, **extremes**, or **details** sections, respectively
- **out_e** (*boolean*) – output control flag for this element/section
- **mpopt** (*struct*) – MATPOWER options struct
- **fd** (*integer*) – (*optional, default = 1*) file identifier to use for printing, (1 for standard output, 2 for standard error)
- **pp_args** (*struct*) – arbitrary struct of additional pretty printing arguments passed to all sub-methods, allowing a single sub-method to be used for multiple output portions (e.g. for active and reactive power) by passing in a different argument; by convention, arguments for a branch element, for example, are passed in `pp_args.branch`, etc.

pp_have_section(section, mpopt, pp_args)

True if pretty-printing for element has specified section.

```
TorF = dme.pp_have_section(section, mpopt, pp_args)
```

Inputs

see `pretty_print()` (page 45) for details

Output

TorF (*boolean*) – true if output includes specified section

Implementation handled by section-specific `pp_have_section` methods or `pp_have_section_other()` (page 61).

See also `pp_have_section_cnt()` (page 46), `pp_have_section_sum()` (page 47), `pp_have_section_ext()` (page 47), `pp_have_section_det()` (page 47).

pp_rows(*dm, section, out_e, mpopt, pp_args*)

Indices of rows to include in pretty-printed output.

```
rows = dme.pp_rows(dm, section, out_e, mpopt, pp_args)
```

Inputs

see [pretty_print\(\)](#) (page 45) for details

Output

rows (*integer*) – index vector of rows to be included in output

- 0 = no rows
- -1 = all rows

Includes all rows by default.

pp_get_headers(*dm, section, out_e, mpopt, pp_args*)

Get pretty-printed headers for this element/section.

```
h = dme.pp_get_headers(dm, section, out_e, mpopt, pp_args)
```

Inputs

see [pretty_print\(\)](#) (page 45) for details

Output

h (*cell array of char arrays*) – lines of pretty printed header output for this element/section

Empty by default for counts, summary and extremes sections, and handled by [pp_get_headers_det\(\)](#) (page 48) for details section.

pp_get_footers(*dm, section, out_e, mpopt, pp_args*)

Get pretty-printed footers for this element/section.

```
f = dme.pp_get_footers(dm, section, out_e, mpopt, pp_args)
```

Inputs

see [pretty_print\(\)](#) (page 45) for details

Output

f (*cell array of char arrays*) – lines of pretty printed footer output for this element/section

Empty by default for counts, summary and extremes sections, and handled by [pp_get_headers_det\(\)](#) (page 48) for details section.

pp_data(*dm, section, rows, out_e, mpopt, fd, pp_args*)

Pretty-print the data for this element/section.

```
dme.pp_data(dm, section, rows, out_e, mpopt, fd, pp_args)
```

Inputs

- **rows** (*integer*) – indices of rows to include, from [pp_rows\(\)](#) (page 45)
- ... – see [pretty_print\(\)](#) (page 45) for details of other inputs

Implementation handled by section-specific *pp_data* methods or [pp_data_other\(\)](#) (page 61).

See also [pp_data_cnt\(\)](#) (page 47), [pp_data_sum\(\)](#) (page 47), [pp_data_ext\(\)](#) (page 47), [pp_data_det\(\)](#) (page 48).

pp_have_section_cnt(*mpopt, pp_args*)

True if pretty-printing for element has **counts** section.

```
TorF = dme.pp_have_section_cnt(mpopt, pp_args)
```

Default is **true**.

See also [pp_have_section\(\)](#) (page 45).

pp_data_cnt(*dm, rows, out_e, mpopt, fd, pp_args*)

Pretty-print the **counts** data for this element.

```
dme.pp_data_cnt(dm, rows, out_e, mpopt, fd, pp_args)
```

See also [pp_data\(\)](#) (page 46).

pp_have_section_sum(*mpopt, pp_args*)

True if pretty-printing for element has **summary** section.

```
TorF = dme.pp_have_section_sum(mpopt, pp_args)
```

Default is **false**.

See also [pp_have_section\(\)](#) (page 45).

pp_data_sum(*dm, rows, out_e, mpopt, fd, pp_args*)

Pretty-print the **summary** data for this element.

```
dme.pp_data_sum(dm, rows, out_e, mpopt, fd, pp_args)
```

Does nothing by default.

See also [pp_data\(\)](#) (page 46).

pp_have_section_ext(*mpopt, pp_args*)

True if pretty-printing for element has **extremes** section.

```
TorF = dme.pp_have_section_ext(mpopt, pp_args)
```

Default is **false**.

See also [pp_have_section\(\)](#) (page 45).

pp_data_ext(*dm, rows, out_e, mpopt, fd, pp_args*)

Pretty-print the **extremes** data for this element.

```
dme.pp_data_ext(dm, rows, out_e, mpopt, fd, pp_args)
```

Does nothing by default.

See also [pp_data\(\)](#) (page 46).

pp_have_section_det(*mpopt, pp_args*)

True if pretty-printing for element has **details** section.

```
TorF = dme.pp_have_section_det(mpopt, pp_args)
```

Default is **false**.

See also [pp_have_section\(\)](#) (page 45).

pp_get_title_det(*mpopt*, *pp_args*)

Get title of **details** section for this element.

```
str = dme.pp_get_title_det(mpop, pp_args)
```

Inputs

see [pretty_print\(\)](#) (page 45) for details

Output

str (*char array*) – title of details section, e.g. 'Bus Data', 'Generator Data', etc.

Called by [pp_get_headers_det\(\)](#) (page 48) to insert title into detail section header.

pp_get_headers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

Get pretty-printed **details** headers for this element.

```
h = dme.pp_get_headers_det(dm, out_e, mpopt, pp_args)
```

See also [pp_get_headers\(\)](#) (page 46).

pp_get_footers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

Get pretty-printed **details** footers for this element.

```
f = dme.pp_get_footers_det(dm, out_e, mpopt, pp_args)
```

Empty by default.

See also [pp_get_footers\(\)](#) (page 46).

pp_data_det(*dm*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)

Pretty-print the **details** data for this element.

```
dme.pp_data_det(dm, rows, out_e, mpopt, fd, pp_args)
```

Calls [pp_data_row_det\(\)](#) (page 48) for each row.

See also [pp_data\(\)](#) (page 46), [pp_data_row_det\(\)](#) (page 48).

pp_data_row_det(*dm*, *k*, *out_e*, *mpopt*, *fd*, *pp_args*)

Get pretty-printed row of **details** data for this element.

```
str = dme.pp_data_row_det(dm, k, out_e, mpopt, fd, pp_args)
```

Inputs

- **k** (*integer*) – index of row to print
- ... – see [pretty_print\(\)](#) (page 45) for details of other inputs

Output

str (*char array*) – row of data (*without newline*)

Called by [pp_data_det\(\)](#) (page 48) for each row.

mp.dme_branch

class mp.dme_branch

Bases: [mp.dm_element](#) (page 38)

[mp.dme_branch](#) (page 49) - Data model element for branch.

Implements the data element model for branch elements, including transmission lines and transformers.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>bus_fr</code>	<i>integer</i>	bus ID (uid) of “from” bus
<code>bus_to</code>	<i>integer</i>	bus ID (uid) of “to” bus
<code>r</code>	<i>double</i>	per unit series resistance
<code>x</code>	<i>double</i>	per unit series reactance
<code>g_fr</code>	<i>double</i>	per unit shunt conductance at “from” end
<code>b_fr</code>	<i>double</i>	per unit shunt susceptance at “from” end
<code>g_to</code>	<i>double</i>	per unit shunt conductance at “to” end
<code>b_to</code>	<i>double</i>	per unit shunt susceptance at “to” end
<code>sm_ub_a</code>	<i>double</i>	long term apparent power rating (MVA)
<code>sm_ub_b</code>	<i>double</i>	short term apparent power rating (MVA)
<code>sm_ub_c</code>	<i>double</i>	emergency apparent power rating (MVA)
<code>cm_ub_a</code>	<i>double</i>	long term current magnitude rating (MVA equivalent at 1 p.u. voltage)
<code>cm_ub_b</code>	<i>double</i>	short term current magnitude rating (MVA equivalent at 1 p.u. voltage)
<code>cm_ub_c</code>	<i>double</i>	emergency current magnitude rating (MVA equivalent at 1 p.u. voltage)
<code>vad_lb</code>	<i>double</i>	voltage angle difference lower bound
<code>vad_ub</code>	<i>double</i>	voltage angle difference upper bound
<code>tm</code>	<i>double</i>	transformer off-nominal turns ratio
<code>ta</code>	<i>double</i>	transformer phase-shift angle (degrees)
<code>pl_fr</code>	<i>double</i>	active power injection at “from” end
<code>ql_fr</code>	<i>double</i>	reactive power injection at “from” end
<code>pl_to</code>	<i>double</i>	active power injection at “to” end
<code>ql_to</code>	<i>double</i>	reactive power injection at “to” end
<code>psh_fr</code>	<i>double</i>	active power shunt losses at “from” end
<code>qsh_fr</code>	<i>double</i>	reactive power shunt losses at “from” end
<code>psh_to</code>	<i>double</i>	active power shunt losses at “to” end
<code>qsh_to</code>	<i>double</i>	reactive power shunt losses at “to” end

Property Summary

fbus

bus index vector for “from” port (port 1) (all branches)

tbus

bus index vector for “to” port (port 2) (all branches)

r

series resistance (p.u.) for branches that are on

x

series reactance (p.u.) for branches that are on

g_fr

shunt conductance (p.u.) at “from” end for branches that are on

g_to
shunt conductance (p.u.) at “to” end for branches that are on

b_fr
shunt susceptance (p.u.) at “from” end for branches that are on

b_to
shunt susceptance (p.u.) at “to” end for branches that are on

tm
transformer off-nominal turns ratio for branches that are on

ta
transformer phase-shift angle (radians) for branches that are on

rate_a
long term flow limit (p.u.) for branches that are on

loss_tol = 1e-4
loss values < this are displayed as -

Method Summary

name()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

export_vars()

export_vars_offline_val()

initialize(dm)

update_status(dm)

build_params(dm)

pp_data_cnt(dm, rows, out_e, mpopt, fd, pp_args)

pp_have_section_sum(mpop, pp_args)

pp_data_sum(dm, rows, out_e, mpopt, fd, pp_args)

pp_get_headers_det(dm, out_e, mpopt, pp_args)

pp_get_footers_det(dm, out_e, mpopt, pp_args)

pp_have_section_det(mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpopt, fd, pp_args)

mp.dme_branch_opf

class mp.dme_branch_opf

Bases: [mp.dme_branch](#) (page 49), [mp.dme_shared_opf](#) (page 61)

[mp.dme_branch_opf](#) (page 51) - Data model element for branch for OPF.

To parent class [mp.dme_branch](#) (page 49), adds shadow prices on flow and angle difference limits, and pretty-printing for **lim** sections.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>mu_flow_fr_u</code>	<i>double</i>	shadow price on flow constraint at “from” end (u/MVA) ¹
<code>mu_flow_to_u</code>	<i>double</i>	shadow price on flow constraint at “to” end (u/MVA) ¹
<code>mu_vad_lb</code>	<i>double</i>	shadow price on lower bound of voltage angle difference constraint ($u/degree$) ¹
<code>mu_vad_ub</code>	<i>double</i>	shadow price on upper bound of voltage angle difference constraint ($u/degree$) ¹

Method Summary

`main_table_var_names()`

`export_vars()`

`export_vars_offline_val()`

`pretty_print(dm, section, out_e, mpopt, fd, pp_args)`

`pp_have_section_lim(mpop, pp_args)`

`pp_binding_rows_lim(dm, out_e, mpopt, pp_args)`

`pp_get_title_lim(mpop, pp_args)`

`pp_get_headers_lim(dm, out_e, mpopt, pp_args)`

`pp_data_row_lim(dm, k, out_e, mpopt, fd, pp_args)`

¹ Here u denotes the units of the objective function, e.g. USD.

mp.dme_bus

class mp.dme_bus

Bases: [mp.dm_element](#) (page 38)

[mp.dme_bus](#) (page 52) - Data model element for bus.

Implements the data element model for bus elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
base_kv	<i>double</i>	base voltage (<i>kV</i>)
type	<i>integer</i>	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
area	<i>integer</i>	area number
zone	<i>integer</i>	loss zone
vm_lb	<i>double</i>	voltage magnitude lower bound (<i>p.u.</i>)
vm_ub	<i>double</i>	voltage magnitude upper bound (<i>p.u.</i>)
va	<i>double</i>	voltage angle (<i>degrees</i>)
vm	<i>double</i>	voltage magnitude (<i>p.u.</i>)

Property Summary

type

node [type](#) (page 52) vector for buses that are on

vm_start

initial voltage magnitudes (*p.u.*) for buses that are on

va_start

initial voltage angles (radians) for buses that are on

vm_lb

voltage magnitude lower bounds for buses that are on

vm_ub

voltage magnitude upper bounds for buses that are on

vm_control

true if voltage is controlled, for buses that are on

Method Summary

name()

label()

labels()

main_table_var_names()

export_vars()

export_vars_offline_val()

init_status(dm)

```

update_status(dm)

build_params(dm)

pp_data_cnt(dm, rows, out_e, mpopt, fd, pp_args)

pp_have_section_ext(mpop, pp_args)

pp_data_ext(dm, rows, out_e, mpopt, fd, pp_args)

pp_have_section_det(mpop, pp_args)

pp_get_headers_det(dm, out_e, mpopt, pp_args)

pp_data_row_det(dm, k, out_e, mpopt, fd, pp_args)

set_bus_type_ref(dm, idx)

set_bus_type_pv(dm, idx)

set_bus_type_pq(dm, idx)

```

mp.dme_bus_opf

class mp.dme_bus_opf

Bases: [mp.dme_bus](#) (page 52), [mp.dme_shared_opf](#) (page 61)

[mp.dme_bus_opf](#) (page 53) - Data model element for bus for OPF.

To parent class [mp.dme_bus](#) (page 52), adds shadow prices on power balance and voltage magnitude limits, and pretty-printing for **lim** sections.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
lam_p	<i>dou-ble</i>	active power nodal price, i.e. shadow price on active power balance constraint (u/MW) ¹
lam_q	<i>dou-ble</i>	reactive power nodal price, i.e. shadow price on reactive power balance constraint ($u/MVAr$) ¹
mu_vm_ll	<i>dou-ble</i>	shadow price on voltage magnitude lower bound ($u/p.u.$) ¹
mu_vm_ul	<i>dou-ble</i>	shadow price on voltage magnitude upper bound ($u/p.u.$) ¹

Method Summary

```

main_table_var_names()

export_vars()

export_vars_offline_val()

```

¹ Here u denotes the units of the objective function, e.g. USD.


```
pp_data_ext(dm, rows, out_e, mpopt, fd, pp_args)
pp_get_headers_det(dm, out_e, mpopt, pp_args)
pp_data_row_det(dm, k, out_e, mpopt, fd, pp_args)
pp_have_section_lim(mpopt, pp_args)
pp_binding_rows_lim(dm, out_e, mpopt, pp_args)
pp_get_headers_lim(dm, out_e, mpopt, pp_args)
pp_data_row_lim(dm, k, out_e, mpopt, fd, pp_args)
```

mp.dme_gen

class mp.dme_gen

Bases: [mp.dm_element](#) (page 38)

[mp.dme_gen](#) (page 54) - Data model element for generator.

Implements the data element model for generator elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus	<i>integer</i>	bus ID (uid)
vm_setpoint	<i>double</i>	voltage magnitude setpoint (<i>p.u.</i>)
pg_lb	<i>double</i>	active power output lower bound (<i>MW</i>)
pg_ub	<i>double</i>	active power output upper bound (<i>MW</i>)
qg_lb	<i>double</i>	reactive power output lower bound (<i>MVar</i>)
qg_ub	<i>double</i>	reactive power output upper bound (<i>MVar</i>)
pg	<i>double</i>	active power output (<i>MW</i>)
qg	<i>double</i>	reactive power output (<i>MVar</i>)
startup_cost_cold	<i>double</i>	cold startup cost (<i>USD</i>)
pc1	<i>double</i>	lower active power output of PQ capability curve (<i>MW</i>)
pc2	<i>double</i>	upper active power output of PQ capability curve (<i>MW</i>)
qc1_lb	<i>double</i>	lower bound on reactive power output at pc1 (<i>MVar</i>)
qc1_ub	<i>double</i>	upper bound on reactive power output at pc1 (<i>MVar</i>)
qc2_lb	<i>double</i>	lower bound on reactive power output at pc2 (<i>MVar</i>)
qc2_ub	<i>double</i>	upper bound on reactive power output at pc2 (<i>MVar</i>)

Property Summary

bus

[bus](#) (page 54) index vector (all gens)

bus_on

vector of indices into online buses for gens that are on

pg_start

initial active power (p.u.) for gens that are on

qg_start

initial reactive power (p.u.) for gens that are on

vm_setpoint

generator voltage setpoint for gens that are on

pg_lb

active power lower bound (p.u.) for gens that are on

pg_ub

active power upper bound (p.u.) for gens that are on

qg_lb

reactive power lower bound (p.u.) for gens that are on

qg_ub

reactive power upper bound (p.u.) for gens that are on

Method Summary

name()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

export_vars()

export_vars_offline_val()

have_cost()

initialize(*dm*)

update_status(*dm*)

apply_vm_setpoint(*dm*)

build_params(*dm*)

violated_q_lims(*dm*, *mpopt*)

isload(*idx*)

pp_have_section_sum(*mpopt*, *pp_args*)

pp_data_sum(*dm*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)

pp_have_section_det(*mpopt*, *pp_args*)

pp_get_headers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_get_footers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_data_row_det(*dm*, *k*, *out_e*, *mpopt*, *fd*, *pp_args*)

mp.dme_gen_opf

class mp.dme_gen_opf

Bases: [mp.dme_gen](#) (page 54), [mp.dme_shared_opf](#) (page 61)

[mp.dme_gen_opf](#) (page 56) - Data model element for generator for OPF.

To parent class [mp.dme_gen](#) (page 54), adds costs, shadow prices on active and reactive generation limits, and pretty-printing for **lim** sections.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>cost_pg</code>	mp.cost_table	active power cost (u/MW) ¹
<code>cost_qg</code>	mp.cost_table	reactive power cost ($u/MVAr$) ¹
<code>mu_pg_lb</code>	<i>double</i>	shadow price on active power output lower bound (u/MW) ¹
<code>mu_pg_ub</code>	<i>double</i>	shadow price on active power output upper bound (u/MW) ¹
<code>mu_qg_lb</code>	<i>double</i>	shadow price on reactive power output lower bound ($u/MVAr$) ¹
<code>mu_qg_ub</code>	<i>double</i>	shadow price on reactive power output upper bound ($u/MVAr$) ¹

The cost tables `cost_pg` and `cost_qg` are defined as tables with the following columns:

See also [mp.cost_table](#) (page 164).

Method Summary

```
main_table_var_names()
export_vars()
export_vars_offline_val()
have_cost()
build_cost_params(dm, dc)
max_pwl_gencost()
pretty_print(dm, section, out_e, mpopt, fd, pp_args)
pp_have_section_lim(mpop, pp_args)
pp_binding_rows_lim(dm, out_e, mpopt, pp_args)
pp_get_headers_lim(dm, out_e, mpopt, pp_args)
pp_data_row_lim(dm, k, out_e, mpopt, fd, pp_args)
```

¹ Here u denotes the units of the objective function, e.g. USD.

mp.dme_load**class mp.dme_load**

Bases: [mp.dm_element](#) (page 38)

[mp.dme_load](#) (page 57) - Data model element for load.

Implements the data element model for load elements, using a ZIP load model.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>bus</code>	<i>integer</i>	bus ID (<code>uid</code>)
<code>pd</code>	<i>double</i>	p_p , active constant power demand (<i>MW</i>)
<code>qd</code>	<i>double</i>	q_p , reactive constant power demand (<i>MVar</i>)
<code>pd_i</code>	<i>double</i>	p_i , active nominal ¹ constant current demand (<i>MW</i>)
<code>qd_i</code>	<i>double</i>	q_i , reactive nominal ¹ constant current demand (<i>MVar</i>)
<code>pd_z</code>	<i>double</i>	p_z , active nominal ¹ constant impedance demand (<i>MW</i>)
<code>qd_z</code>	<i>double</i>	q_z , reactive nominal ¹ constant impedance demand (<i>MVar</i>)
<code>p</code>	<i>double</i>	p , total active demand (<i>MW</i>)
<code>q</code>	<i>double</i>	q , total reactive demand (<i>MVar</i>)

Implements a ZIP load model, where each load has three components, and total demand for the load i is given by

$$\begin{aligned} s &= s_p + s_i|v| + s_z|v|^2 \\ p + jq &= (p_p + jq_p) + (p_i + jq_i)|v| + (p_z + jq_z)|v|^2 \end{aligned} \quad (3.1)$$

Property Summary**bus**

[bus](#) (page 57) index vector (all loads)

pd

active power demand (p.u.) for constant power loads that are on

qd

reactive power demand (p.u.) for constant power loads that are on

pd_i

active power demand (p.u.) for constant current loads that are on

qd_i

reactive power demand (p.u.) for constant current loads that are on

pd_z

active power demand (p.u.) for constant impedance loads that are on

qd_z

reactive power demand (p.u.) for constant impedance loads that are on

Method Summary**name()**

¹ *Nominal* means for a voltage of 1 p.u.

```
label()  
labels()  
cxn_type()  
cxn_idx_prop()  
main_table_var_names()  
initialize(dm)  
update_status(dm)  
build_params(dm)  
pp_have_section_sum(mpop, pp_args)  
pp_data_sum(dm, rows, out_e, mpop, fd, pp_args)  
pp_have_section_det(mpop, pp_args)  
pp_get_headers_det(dm, out_e, mpop, pp_args)  
pp_get_footers_det(dm, out_e, mpop, pp_args)  
pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)
```

mp.dme_load_cpf

class `mp.dme_load_cpf`

Bases: [mp.dme_load](#) (page 57)

[mp.dme_load_cpf](#) (page 58) - Data model element for load for CPF.

To parent class [mp.dme_load](#) (page 57), adds method for adjusting model parameters based on value of continuation parameter λ , and overrides `export_vars` to export these updated parameter values.

Method Summary

```
export_vars()  
parameterized(dm, dmb, dmt, lam)
```

mp.dme_load_opf

class `mp.dme_load_opf`

Bases: [mp.dme_load](#) (page 57), [mp.dme_shared_opf](#) (page 61)

[mp.dme_load_opf](#) (page 58) - Data model element for load for OPF.

To parent class [mp.dme_load](#) (page 57), adds pretty-printing for **lim** sections.

mp.dme_shunt_cpf

class mp.dme_shunt_cpf

Bases: [mp.dme_shunt](#) (page 59)

[mp.dme_shunt_cpf](#) (page 59) - Data model element for shunt for CPF.

To parent class [mp.dme_shunt](#) (page 59), adds method for adjusting model parameters based on value of continuation parameter λ , and overrides [export_vars\(\)](#) (page 59) to export these updated parameter values.

Method Summary

export_vars()

parameterized(*dm, dmb, dmt, lam*)

mp.dme_shunt

class mp.dme_shunt

Bases: [mp.dm_element](#) (page 38)

[mp.dme_shunt](#) (page 59) - Data model element for shunt.

Implements the data element model for shunt elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus	<i>integer</i>	bus ID (uid)
gs	<i>double</i>	g_s , shunt conductance, specified as nominal ¹ active power demand (<i>MW</i>)
bs	<i>double</i>	b_s , shunt susceptance, specified as nominal ¹ reactive power injection (<i>MVar</i>)
p	<i>double</i>	p , total active power absorbed (<i>MW</i>)
q	<i>double</i>	q , total reactive power absorbed (<i>MVar</i>)

Property Summary

bus

[bus](#) (page 59) index vector (all shunts)

gs

shunt conductance (p.u. active power demanded at

¹ *Nominal* means for a voltage of 1 p.u.

bs

V = 1.0 p.u.) for shunts that are on

Method Summary

name()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

initialize(*dm*)

update_status(*dm*)

build_params(*dm*)

pp_have_section_sum(*mpopt*, *pp_args*)

pp_data_sum(*dm*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)

pp_have_section_det(*mpopt*, *pp_args*)

pp_get_headers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_get_footers_det(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_data_row_det(*dm*, *k*, *out_e*, *mpopt*, *fd*, *pp_args*)

mp.dme_shunt_opf

class mp.dme_shunt_opf

Bases: [mp.dme_shunt](#) (page 59), [mp.dme_shared_opf](#) (page 61)

[mp.dme_shunt_opf](#) (page 60) - Data model element for shunt for OPF.

To parent class [mp.dme_shunt](#) (page 59), adds pretty-printing for **lim** sections.

3.2.3 Element Mixins

`mp.dme_shared_opf`

`class mp.dme_shared_opf`

Bases: `handle`

[`mp.dme_shared_opf`](#) (page 61) - Mixin class for OPF **data model element** objects.

For all elements of [`mp.data_model_opf`](#) (page 37), adds shared functionality for pretty-printing of **lim** sections.

Property Summary

ctol

constraint violation tolerance

ptol

shadow price tolerance

Method Summary

pp_set_tols_lim(*mpopt*)

pp_have_section_other(*section*, *mpopt*, *pp_args*)

pp_rows_other(*dm*, *section*, *out_e*, *mpopt*, *pp_args*)

pp_get_headers_other(*dm*, *section*, *out_e*, *mpopt*, *pp_args*)

pp_get_footers_other(*dm*, *section*, *out_e*, *mpopt*, *pp_args*)

pp_data_other(*dm*, *section*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)

pp_have_section_lim(*mpopt*, *pp_args*)

pp_rows_lim(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_binding_rows_lim(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_get_title_lim(*mpopt*, *pp_args*)

pp_get_headers_lim(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_get_footers_lim(*dm*, *out_e*, *mpopt*, *pp_args*)

pp_data_lim(*dm*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)

pp_data_row_lim(*dm*, *k*, *out_e*, *mpopt*, *fd*, *pp_args*)

3.3 Data Model Converter Classes

3.3.1 Containers

mp.dm_converter

class mp.dm_converter

Bases: [mp.element_container](#) (page 168)

[mp.dm_converter](#) (page 62) - Abstract base class for MATPOWER **data model converter** objects.

A data model converter provides the ability to convert data between a data model and a specific data source or format, such as the PSS/E RAW format or version 2 of the MATPOWER case format. It is used, for example, during the import stage of the data model build process.

A data model converter object is primarily a container for data model converter element ([mp.dmc_element](#) (page 65)) objects. Concrete data model converter classes are specific to the type or format of the data source.

By convention, data model converter variables are named `dmc` and data model converter class names begin with `mp.dm_converter`.

mp.dm_converter Methods:

- [format_tag\(\)](#) (page 62) - return char array identifier for data source/format
- [copy\(\)](#) (page 62) - make duplicate of object
- [build\(\)](#) (page 62) - create and add element objects
- [import\(\)](#) (page 62) - import data from a data source into a data model
- [export\(\)](#) (page 63) - export data from a data model to a data source
- [init_export\(\)](#) (page 63) - initialize a data source for export
- [save\(\)](#) (page 63) - save data source to a file
- [display\(\)](#) (page 63) - display the data model converter object

See the `sec_dm_converter` section in the *MATPOWER Developer's Manual* for more information.

See also [mp.data_model](#) (page 30), [mp.task](#) (page 9).

Method Summary

format_tag()

Return a short char array identifier for data source/format.

```
tag = dmc.format_tag()
```

E.g. the subclass for the MATPOWER case format returns `'mpc2'`.

Note: This is an abstract method that must be implemented by a subclass.

copy()

Create a duplicate of the data model converter object, calling the `copy()` method on each element.

```
new_dmc = dmc.copy()
```

build()

Create and add data model converter element objects.

```
dmc.build()
```

Create the data model converter element objects by instantiating each class in the [element_classes](#) (page 168) property and adding the resulting object to the [elements](#) (page 168) property.

import(dm, d)

Import data from a data source into a data model.

```
dm = dmc.import(dm, d)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))

Output

dm ([mp.data_model](#) (page 30)) – updated data model

Calls the [import\(\)](#) (page 62) method for each data model converter element and its corresponding data model element.

export(dm, d)

Export data from a data model to a data source.

```
d = dmc.export(dm, d)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))

Output

d – updated data source

Calls the [export\(\)](#) (page 63) method for each data model converter element and its corresponding data model element.

init_export(dm)

Initialize a data source for export.

```
d = dmc.export(dm)
```

Input

dm ([mp.data_model](#) (page 30)) – data model

Output

d – new empty data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))

Creates a new data source of the appropriate type in preparation for calling [export\(\)](#) (page 63).

save(fname, d)

Save data source to a file.

```
fname_out = dmc.save(fname, d)
```

Inputs

- **fname** (*char array*)
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))

Output

fname_out (*char array*) – final file name after saving, possibly modified from input (e.g. extension added)

Note: This is an abstract method that must be implemented by a subclass.

display()

Display the data model converter object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the data model converter elements.

mp.dm_converter_mpc2**class mp.dm_converter_mpc2**

Bases: [mp.dm_converter](#) (page 62)

[mp.dm_converter_mpc2](#) (page 64) - MATPOWER **data model converter** for MATPOWER case v2.

This class implements importing/exporting of data models for version 2 of the classic MATPOWER case format. That is, the *data source* **d** for this class is expected to be a MATPOWER case struct.

mp.dm_converter_mpc2 Methods:

- [dm_converter_mpc2\(\)](#) (page 64) - constructor
- [format_tag\(\)](#) (page 64) - return char array identifier for data source/format ('mpc2')
- [import\(\)](#) (page 64) - import data from a MATPOWER case struct into a data model
- [export\(\)](#) - export data from a data model to a MATPOWER case struct
- [save\(\)](#) (page 64) - save MATPOWER case struct to a file

See also [mp.dm_converter](#) (page 62).

Constructor Summary**dm_converter_mpc2()**

Specify the element classes for handling MATPOWER case format.

Method Summary**format_tag()**

Return identifier tag 'mpc2' for version 2 MATPOWER case format.

import(dm, d)

Import data from a version 2 MATPOWER case struct into a data model.

init_export(dm)

Initialize a MATPOWER case struct for export.

save(fname, d)

Save a MATPOWER case struct to a file.

mp.dm_converter_mpc2_legacy

class mp.dm_converter_mpc2_legacy

Bases: [mp.dm_converter_mpc2](#) (page 64)

[mp.dm_converter_mpc2_legacy](#) (page 65) - Legacy MATPOWER **data model converter** for MATPOWER case v2.

Adds to [mp.dm_converter_mpc2](#) (page 64) the ability to handle legacy user customization.

mp.dm_converter_mpc2_legacy Methods:

- [legacy_user_mod_inputs\(\)](#) (page 65) - pre-process legacy inputs for use-defined customization
- [legacy_user_nln_constraints\(\)](#) (page 65) - pre-process legacy inputs for user-defined nonlinear constraints

See also [mp.dm_converter](#) (page 62), [mp.dm_converter_mpc2](#) (page 64), [mp.taskopf_legacy](#) (page 27).

Method Summary

legacy_user_mod_inputs(dm, mpopt, dc)

Handle pre-processing of inputs related to legacy user-defined variables, costs, and constraints. This includes optional mpc fields A, l, u, N, fparam, H1, Cw, z0, z1, zu and user_constraints.

legacy_user_nln_constraints(dm, mpopt)

Handle pre-processing of inputs related to legacy user-defined non-linear constraints, specifically optional mpc fields user_constraints.nle and user_constraints.nli.

Called by [legacy_user_mod_inputs\(\)](#) (page 65) method.

3.3.2 Elements

mp.dmc_element

class mp.dmc_element

Bases: handle

[mp.dmc_element](#) (page 65)- Abstract base class for **data model converter element** objects.

A data model converter element object implements the functionality needed to import and export a particular element type from and to a given data format. All data model converter element classes inherit from [mp.dmc_element](#) (page 65) and each element type typically implements its own subclass.

By convention, data model converter element variables are named dmce and data model converter element class names begin with mp.dmce.

Typically, much of the import/export functionality for a particular concrete subclass can be defined simply by implementing the [table_var_map\(\)](#) (page 68) method.

mp.dmc_element Methods:

- [name\(\)](#) (page 66) - get name of element type, e.g. 'bus', 'gen'
- [data_model_element\(\)](#) (page 66) - get corresponding data model element
- [data_field\(\)](#) (page 66) - get name of field in data source corresponding to default data table

- `data_subs()` (page 67) - get subscript reference struct for accessing data source
- `data_exists()` (page 67) - check if default field exists in data source
- `get_import_spec()` (page 67) - get import specification
- `get_export_spec()` (page 67) - get export specification
- `get_import_size()` (page 68) - get dimensions of data to be imported
- `get_export_size()` (page 68) - get dimensions of data to be exported
- `table_var_map()` (page 68) - get variable map for import/export
- `import()` (page 68) - import data from data source into data model element
- `import_table_values()` (page 69) - import table values for given import specification
- `get_input_table_values()` (page 69) - get values to insert in data model element table
- `import_col()` (page 69) - extract and optionally modify values from data source column
- `export()` (page 70) - export data from data model element to data source
- `export_table_values()` (page 70) - export table values for given import specification
- `init_export_data()` (page 70) - initialize data source for export from data model element
- `default_export_data_table()` (page 71) - create default (empty) data table for data source
- `default_export_data_nrows()` (page 71) - get number of rows `default_export_data_table()` (page 71)
- `export_col()` (page 71) - export a variable (table column) to the data source

See the `sec_dmc_element` section in the *MATPOWER Developer's Manual* for more information.

See also `mp.dm_converter` (page 62).

Method Summary

`name()`

Get name of element type, e.g. 'bus', 'gen'.

```
name = dmce.name()
```

Output

name (*char array*) – name of element type, must be a valid struct field name

Implementation provided by an element type specific subclass.

`data_model_element(dm, name)`

Get the corresponding data model element.

```
dme = dmce.data_model_element(dm)
dme = dmce.data_model_element(dm, name)
```

Inputs

- **dm** (`mp.data_model` (page 30)) – data model object
- **name** (*char array*) – (optional) name of element type (default is name of this object)

Output

dme (`mp.dm_element` (page 38)) – data model element object

data_field()

Get name of field in data source corresponding to default data table.

```
df = dmce.data_field()
```

Output

df (*char array*) – field name

data_subs()

Get subscript reference struct for accessing data source.

```
s = dmce.data_subs()
```

Output

s (*struct*) – same as the **s** input argument to the built-in `subsref()`, to access this element's data in data source, with fields:

- **type** – character vector or string containing '()', '{}', or '.' specifying the subscript type
- **subs** – cell array, character vector, or string containing the actual subscripts

The default implementation in this base class uses the return value of the `data_field()` (page 66) method to access a field of the data source struct. That is:

```
s = struct('type', '.', 'subs', dmce.data_field());
```

data_exists(d)

Check if default field exists in data source.

```
TorF = dmce.data_exists(d)
```

Input

d – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2` (page 64))

Output

TorF (*boolean*) – true if field exists

Check if value returned by `data_field()` (page 66) exists as a field in **d**.

get_import_spec(dme, d)

Get import specification.

```
spec = dmce.get_import_spec(dme, d)
```

Inputs

- **dme** (`mp.dm_element` (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2` (page 64))

Output

spec (*struct*) – import specification, with keys:

- 'subs' - subscript reference struct for accessing data source, as returned by `data_subs()` (page 67)
- 'nr', 'nc', 'r' - number of rows, number of columns, row index vector, as returned by `get_import_size()` (page 68)
- 'vmap' - variable map, as returned by `table_var_map()` (page 68)

See also `get_export_spec()` (page 67).

get_export_spec(*dme*, *d*)

Get export specification.

```
spec = dmce.get_export_spec(dme, d)
```

Inputs

- **dme** (*mp.dm_element* (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))

Output

spec (*struct*) – export specification, see *get_import_spec()* (page 67)

See also *get_import_spec()* (page 67).

get_import_size(*d*)

Get dimensions of data to be imported.

```
[nr, nc, r] = dmce.get_import_size(d)
```

Input

d – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))

Outputs

- **nr** (*integer*) – number of rows of data
- **nc** (*integer*) – number of columns of data
- **r** (*integer*) – optional index vector (*empty by default*) of rows in data source field that correspond to data to be imported

get_export_size(*dme*)

Get dimensions of data to be exported.

```
[nr, nc, r] = dmce.get_export_size(dme)
```

Input

dme (*mp.dm_element* (page 38)) – data model element object

Outputs

- **nr** (*integer*) – number of rows of data
- **nc** (*integer*) – number of columns of data
- **r** (*integer*) – optional index vector (*empty by default*) of rows in main table of *dme* that correspond to data to be exported

table_var_map(*dme*, *d*)

Get variable map for import/export.

```
vmap = dmce.table_var_map(dme, d)
```

Inputs

- **dme** (*mp.dm_element* (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))

Output

vmap (*struct*) – variable map, see *tab_var_map* in the *MATPOWER Developer's Manual* for details

This method initializes each entry to { 'col', [] } by default, so subclasses only need to assign *vmap.vn*{2} for columns that map directly from a column of the data source.

import(*dme*, *d*, *var_names*, *ridx*)

Import data from data source into data model element.

```
dme = dmce.import(dme, d, var_names, ridx)
```

Inputs

- **dme** (*mp.dm_element* (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))
- **var_names** (*cell array*) – (optional) list of names of variables (columns of main table) to import (*default is all variables*)
- **ridx** (*integer*) – (optional) vector of row indices of data to import (*default is all rows*)

Output

dme (*mp.dm_element* (page 38)) – updated data model element object

See also [export\(\)](#) (page 70).

import_table_values(*dme*, *d*, *spec*, *var_names*, *ridx*)

Import table values for given import specification.

```
dme = dmce.import_table_values(dme, d, spec, var_names, ridx)
```

Inputs

- **dme** (*mp.dm_element* (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))
- **spec** (*struct*) – import specification, see [get_import_spec\(\)](#) (page 67)
- **var_names** (*cell array*) – (optional) list of names of variables (columns of main table) to import (*default is all variables*)
- **ridx** (*integer*) – (optional) vector of row indices of data to import (*default is all rows*)

Output

dme (*mp.dm_element* (page 38)) – updated data model element object

Called by [import\(\)](#) (page 68).

get_input_table_values(*d*, *spec*, *var_names*, *ridx*)

Get values to insert in data model element table.

```
vals = dmce.get_input_table_values(d, spec, var_names, ridx)
```

Inputs

- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))
- **spec** (*struct*) – import specification, see [get_import_spec\(\)](#) (page 67)
- **var_names** (*cell array*) – (optional) list of names of variables (columns of main table) to import (*default is all variables*)
- **ridx** (*integer*) – (optional) vector of row indices of data to import (*default is all rows*)

Output

vals (*cell array*) – values to assign to table columns in data model element

Called by [import_table_values\(\)](#) (page 69).

import_col(*d*, *spec*, *vn*, *c*, *sf*)

Extract and optionally modify values from data source column.

```
vals = dmce.import_col(d, spec, vn, c, sf)
```


Inputs

- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))
- **spec** (*struct*) – import specification, see [get_import_spec\(\)](#) (page 67)
- **vn** (*char array*) – variable name
- **c** (*integer*) – column index for data in data source
- **sf** (*double or function handle*) – (*optional*) scale factor, function is called as *sf(dmce, vn)*

Output

vals (*cell array*) – values to assign to table columns in data model element

Called by [get_input_table_values\(\)](#) (page 69).

export(*dme, d, var_names, ridx*)

Export data from data model element to data source.

```
d = dmce.export(dme, d, var_names, ridx)
```

Inputs

- **dme** ([mp.dm_element](#) (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))
- **var_names** (*cell array*) – (*optional*) list of names of variables (columns of main table) to export (*default is all variables*)
- **ridx** (*integer*) – (*optional*) vector of row indices of data to export (*default is all rows*)

Output

d – updated data source

See also [import\(\)](#) (page 68).

export_table_values(*dme, d, spec, var_names, ridx*)

Export table values for given import specification.

```
d = dmce.export_table_values(dme, d, spec, var_names, ridx)
```

Inputs

- **dme** ([mp.dm_element](#) (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))
- **spec** (*struct*) – export specification, see [get_export_spec\(\)](#) (page 67)
- **var_names** (*cell array*) – (*optional*) list of names of variables (columns of main table) to export (*default is all variables*)
- **ridx** (*integer*) – (*optional*) vector of row indices of data to export (*default is all rows*)

Output

d – updated data source

Called by [export\(\)](#) (page 70).

init_export_data(*dme, d, spec*)

Initialize data source for export from data model element.

```
d = dmce.init_export_data(dme, d, spec)
```

Inputs

- **dme** ([mp.dm_element](#) (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for [mp.dm_converter_mpc2](#) (page 64))
- **spec** (*struct*) – export specification, see [get_export_spec\(\)](#) (page 67)

Output

d – updated data source

Called by [export_table_values\(\)](#) (page 70).

default_export_data_table(spec)

Create default (empty) data table for data source.

```
dt = dmce.default_export_data_table(spec)
```

Input

spec (*struct*) – export specification, see [get_export_spec\(\)](#) (page 67)

Output

dt – data table for data source, type depends on implementing subclass

Called by [init_export_data\(\)](#) (page 70).

default_export_data_nrows(spec)

Get number of rows for [default_export_data_table\(\)](#) (page 71).

```
nr = default_export_data_nrows(spec)
```

Input

spec (*struct*) – export specification, see [get_export_spec\(\)](#) (page 67)

Output

nr (*integer*) – number of rows

Called by [default_export_data_table\(\)](#) (page 71).

export_col(dme, d, spec, vn, ridx, c, sf)

Export a variable (table column) to the data source.

```
d = dmce.export_col(dme, d, spec, vn, ridx, c, sf)
```

Inputs

- **dme** (*mp.dm_element* (page 38)) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for *mp.dm_converter_mpc2* (page 64))
- **spec** (*struct*) – export specification, see [get_export_spec\(\)](#) (page 67)
- **vn** (*char array*) – variable name
- **ridx** (*integer*) – (*optional*) vector of row indices of data to export (*default is all rows*)
- **c** (*integer*) – column index for data in data source
- **sf** (*double or function handle*) – (*optional*) scale factor, function is called as *sf(dmce, vn)*

Output

d – updated data source

Called by [export_table_values\(\)](#) (page 70).

mp.dmce_branch_mpc2

class mp.dmce_branch_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_branch_mpc2](#) (page 72) - Data model converter element for branch for MATPOWER case v2.

Method Summary

name()
data_field()
table_var_map(*dme, mpc*)
default_export_data_table(*spec*)

mp.dmce_bus_mpc2

class mp.dmce_bus_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_bus_mpc2](#) (page 72) - Data model converter element for bus for MATPOWER case v2.

Method Summary

name()
data_field()
table_var_map(*dme, mpc*)
init_export_data(*dme, d, spec*)
default_export_data_table(*spec*)
bus_name_import(*mpc, spec, vn, c*)
bus_name_export(*dme, mpc, spec, vn, ridx, c*)
bus_status_import(*mpc, spec, vn, c*)

mp.dmce_gen_mpc2

class mp.dmce_gen_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_gen_mpc2](#) (page 72) - Data model converter element for generator for MATPOWER case v2.

Property Summary

pwl1indices of single-block piecewise linear costs, all gens (*automatically converted to linear cost*)**Method Summary****name()****data_field()****table_var_map**(*dme, mpc*)**default_export_data_table**(*spec*)**start_cost_import**(*mpc, spec, vn*)**start_cost_export**(*dme, mpc, spec, vn, ridx*)**gen_cost_import**(*mpc, spec, vn, p_or_q*)**gen_cost_export**(*dme, mpc, spec, vn, p_or_q, ridx*)**static gencost2cost_table**(*gencost*)**static cost_table2gencost**(*gencost0, cost, ridx*)**mp.dmce_load_mpc2****class mp.dmce_load_mpc2**Bases: *mp.dmc_element* (page 65)*mp.dmce_load_mpc2* (page 73) - Data model converter element for load for MATPOWER case v2.**Property Summary****bus****Method Summary****name()****data_field()****get_import_size**(*mpc*)**get_export_size**(*dme*)**table_var_map**(*dme, mpc*)**scale_factor_fcn**(*vn, zip_sf*)**sys_wide_zip_loads**(*mpc*)

`mp.dmce_shunt_mpc2`

class `mp.dmce_shunt_mpc2`

Bases: `mp.dmc_element` (page 65)

`mp.dmce_shunt_mpc2` (page 74) - Data model converter element for shunt for MATPOWER case v2.

Property Summary

bus

Method Summary

name()

data_field()

get_import_size(*mpc*)

get_export_size(*dme*)

table_var_map(*dme, mpc*)

3.4 Network Model Classes

3.4.1 Containers

`mp.form`

class `mp.form`

Bases: `handle`

`mp.form` (page 74) - Abstract base class for MATPOWER **formulation**.

Used as a mix-in class for all **network model element** classes. That is, each concrete network model element class must inherit, at least indirectly, from both `mp.nm_element` (page 110) and `mp.form` (page 74).

`mp.form` (page 74) provides properties and methods that are specific to the network model formulation (e.g. DC version, AC polar power version, etc.).

For more details, see the `sec_net_model_formulations` section in the *MATPOWER Developer's Manual* and the derivations in *MATPOWER Technical Note 5*.

mp.form Properties:

subclasses provide properties for model parameters

mp.form Methods:

- `form_name()` (page 75) - get char array w/name of formulation
- `form_tag()` (page 75) - get char array w/short label of formulation
- `model_params()` (page 75) - get cell array of names of model parameters
- `model_vvars()` (page 75) - get cell array of names of voltage state variables
- `model_zvars()` (page 75) - get cell array of names of non-voltage state variables

- `get_params()` (page 76) - get network model element parameters
- `find_form_class()` (page 76) - get name of network element object's formulation subclass

See also `mp.nm_element` (page 110).

Method Summary

`form_name()`

Get user-readable name of formulation, e.g. 'DC', 'AC-cartesian', 'AC-polar'.

```
name = nme.form_name()
```

Output

name (*char array*) – name of formulation

Note: This is an abstract method that must be implemented by a subclass.

`form_tag()`

Get short label of formulation, e.g. 'dc', 'acc', 'acp'.

```
tag = nme.form_tag()
```

Output

tag (*char array*) – short label of formulation

Note: This is an abstract method that must be implemented by a subclass.

`model_params()`

Get cell array of names of model parameters.

```
params = nme.model_params()
```

Output

params (*cell array of char arrays*) – names of object properties for model parameters

Note: This is an abstract method that must be implemented by a subclass.

`model_vvars()`

Get cell array of names of voltage state variables.

```
vtypes = nme.model_vvars()
```

Output

vtypes (*cell array of char arrays*) – names of network object properties for voltage state variables

The network model object, which inherits from `mp_idx_manager`, uses these values as set types for tracking its voltage state variables.

Note: This is an abstract method that must be implemented by a subclass.

`model_zvars()`

Get cell array of names of non-voltage state variables.

```
vtypes = nme.model_zvars()
```

Output

vtypes (*cell array of char arrays*) – names of network object properties for voltage state variables

The network model object, which inherits from `mp_idx_manager`, uses these values as set types for tracking its non-voltage state variables.

Note: This is an abstract method that must be implemented by a subclass.

`get_params(idx, names)`

Get network model element parameters.

```
[p1, p2, ..., pN] = nme.get_params(idx)
pA = nme.get_params(idx, nameA)
[pA, pB, ...] = nme.get_params(idx, {nameA, nameB, ...})
```

Inputs

- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns parameters corresponding to all ports
- **names** (*char array or cell array of char arrays*) – (*optional*) name(s) of parameters to return

Outputs

- **p1, p2, ..., pN** – full set of parameters in canonical order
- **pA, pB** – parameters specified by **names**

If a particular parameter in the object is empty, this method returns a sparse zero matrix or vector of the appropriate size.

`find_form_class()`

Get name of network element object's formulation subclass.

```
form_class = nme.find_form_class()
```

Output

form_class (*char array*)

Selects from this network model elements parent classes, the `mp.form` (page 74) subclass, that is not a subclass of `mp.nm_element` (page 110), with the longest inheritance path back to `mp.form` (page 74).

`mp.form_ac`

`class mp.form_ac`

Bases: `mp.form` (page 74)

`mp.form_ac` (page 76) - Abstract base class for MATPOWER AC **formulations**.

Used as a mix-in class for all **network model element** classes with an AC network model formulation. That is, each concrete network model element class with an AC formulation must inherit, at least indirectly, from both `mp.nm_element` (page 110) and `mp.form_ac` (page 76).

`mp.form_ac` (page 76) defines the complex port injections as functions of the state variables **x**, that is, the complex voltages **v** and non-voltage states **z**. They are defined in terms of 3 components, the linear current injection and linear power injection components,

$$\begin{aligned} \mathbf{i}^{lin}(\mathbf{x}) &= \begin{bmatrix} \mathbf{Y} & \mathbf{L} \end{bmatrix} \mathbf{x} + \mathbf{i} \\ &= \mathbf{Y}\mathbf{v} + \mathbf{L}\mathbf{z} + \mathbf{i} \end{aligned} \quad (3.2)$$

$$\begin{aligned} \mathbf{s}^{lin}(\mathbf{x}) &= \begin{bmatrix} \mathbf{M} & \mathbf{N} \end{bmatrix} \mathbf{x} + \mathbf{s} \\ &= \mathbf{M}\mathbf{v} + \mathbf{N}\mathbf{z} + \mathbf{s}, \end{aligned} \quad (3.3)$$

and an arbitrary nonlinear injection component represented by $s^{nln}(\mathbf{x})$ or $i^{nln}(\mathbf{x})$. The full complex power and current port injection functions implemented by `mp.form_ac` (page 76), are respectively

$$\begin{aligned} \mathbf{g}^S(\mathbf{x}) &= [\mathbf{v}_\setminus] (\mathbf{i}^{lin}(\mathbf{x}))^* + \mathbf{s}^{lin}(\mathbf{x}) + \mathbf{s}^{nln}(\mathbf{x}) \\ &= [\mathbf{v}_\setminus] (\mathbf{Y}\mathbf{v} + \mathbf{L}\mathbf{z} + \mathbf{i})^* + \mathbf{M}\mathbf{v} + \mathbf{N}\mathbf{z} + \mathbf{s} + \mathbf{s}^{nln}(\mathbf{x}) \end{aligned} \quad (3.4)$$

$$\begin{aligned} \mathbf{g}^I(\mathbf{x}) &= \mathbf{i}^{lin}(\mathbf{x}) + [\mathbf{s}^{lin}(\mathbf{x})_\setminus]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x}) \\ &= \mathbf{Y}\mathbf{v} + \mathbf{L}\mathbf{z} + \mathbf{i} + [\mathbf{M}\mathbf{v} + \mathbf{N}\mathbf{z} + \mathbf{s}_\setminus]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x}) \end{aligned} \quad (3.5)$$

where \mathbf{Y} , \mathbf{L} , \mathbf{M} , \mathbf{N} , \mathbf{i} , and \mathbf{s} , along with $s^{nln}(\mathbf{x})$ or $i^{nln}(\mathbf{x})$, are the model parameters.

For more details, see the `sec_nm_formulations_ac` section in the *MATPOWER Developer's Manual* and the derivations in *MATPOWER Technical Note 5*.

mp.form_dc Properties:

- Y (page 78) - $n_p n_k \times n_n$ matrix \mathbf{Y} of model parameters
- L (page 78) - $n_p n_k \times n_z$ matrix \mathbf{L} of model parameters
- M (page 78) - $n_p n_k \times n_n$ matrix \mathbf{M} of model parameters
- N (page 78) - $n_p n_k \times n_z$ matrix \mathbf{N} of model parameters
- i (page 78) - $n_p n_k \times 1$ vector \mathbf{i} of model parameters
- s (page 78) - $n_p n_k \times 1$ vector \mathbf{s} of model parameters
- `params_ncols` - specify number of columns for each parameter
- `inln` (page 78) - function to compute $i^{nln}(\mathbf{x})$
- `snln` (page 78) - function to compute $s^{nln}(\mathbf{x})$
- `inln_hess` (page 78) - function to compute Hessian of $i^{nln}(\mathbf{x})$
- `snln_hess` (page 78) - function to compute Hessian of $s^{nln}(\mathbf{x})$

mp.form_dc Methods:

- `model_params()` (page 79) - get network model element parameters (`{'Y', 'L', 'M', 'N', 'i', 's'}`)
- `model_zvars()` (page 79) - get cell array of names of non-voltage state variables (`{'zr', 'zi'}`)
- `port_inj_current()` (page 79) - compute port current injections from network state
- `port_inj_power()` (page 79) - compute port power injections from network state
- `port_inj_current_hess()` (page 80) - compute Hessian of port current injections
- `port_inj_power_hess()` (page 81) - compute Hessian of port power injections
- `port_inj_current_jac()` (page 81) - abstract method to compute voltage-related Jacobian terms
- `port_inj_current_hess_v()` (page 81) - abstract method to compute voltage-related Hessian terms
- `port_inj_current_hess_vz()` (page 81) - abstract method to compute voltage-related Hessian terms
- `port_inj_power_jac()` (page 81) - abstract method to compute voltage-related Jacobian terms
- `port_inj_power_hess_v()` (page 81) - abstract method to compute voltage-related Hessian terms
- `port_inj_power_hess_vz()` (page 82) - abstract method to compute voltage-related Hessian terms

- `port_apparent_power_lim_fcn()` (page 82) - compute port squared apparent power injection constraints
- `port_active_power_lim_fcn()` (page 82) - compute port active power injection constraints
- `port_active_power2_lim_fcn()` (page 82) - compute port squared active power injection constraints
- `port_current_lim_fcn()` (page 83) - compute port squared current injection constraints
- `port_apparent_power_lim_hess()` (page 83) - compute port squared apparent power injection Hessian
- `port_active_power_lim_hess()` (page 84) - compute port active power injection Hessian
- `port_active_power2_lim_hess()` (page 84) - compute port squared active power injection Hessian
- `port_current_lim_hess()` (page 84) - compute port squared current injection Hessian
- `aux_data_va_vm()` (page 85) - abstract method to return voltage angles/magnitudes from auxiliary data

See also `mp.form` (page 74), `mp.form_acc` (page 85), `mp.form_acp` (page 89), `mp.form_dc` (page 91), `mp.nm_element` (page 110).

Property Summary

Y = []
(double) $n_p n_k \times n_n$ matrix **Y** of model parameter coefficients for **v**

L = []
(double) $n_p n_k \times n_z$ matrix **L** of model parameter coefficients for **z**

M = []
(double) $n_p n_k \times n_n$ matrix **M** of model parameter coefficients for **v**

N = []
(double) $n_p n_k \times n_z$ matrix **N** of model parameter coefficients for **z**

i = []
(double) $n_p n_k \times 1$ vector **i** of model parameters

s = []
(double) $n_p n_k \times 1$ vector **s** of model parameters

param_ncols = `struct('Y',2,'L',3,'M',2,'N',3,'i',1,'s',1)`
(struct) specify number of columns for each parameter, where

- 1 => single column (i.e. a vector)
- 2 => n_p columns
- 3 => n_z columns

inln = ''
(function handle) function to compute $\mathbf{i}^{nln}(\mathbf{x})$

snln = ''
(function handle) function to compute $\mathbf{s}^{nln}(\mathbf{x})$

inln_hess = ''
(function handle) function to compute Hessian of $\mathbf{i}^{nln}(\mathbf{x})$

snln_hess = ''

(function handle) function to compute Hessian of $s^{nln}(\mathbf{x})$

Method Summary

model_params()

Get cell array of names of model parameters, i.e. {'Y', 'L', 'M', 'N', 'i', 's'}.

See [mp.form.model_params\(\)](#) (page 75).

model_zvars()

Get cell array of names of non-voltage state variables, i.e. {'zr', 'zi'}.

See [mp.form.model_zvars\(\)](#) (page 75).

port_inj_current(x_, sysx, idx)

Compute port complex current injections from network state.

```
I = nme.port_inj_current(x_, sysx)
I = nme.port_inj_current(x_, sysx, idx)
[I, Iv1, Iv2] = nme.port_inj_current(...)
[I, Iv1, Iv2, Izr, Izi] = nme.port_inj_current(...)
```

Compute the complex current injections for all or a selected subset of ports and, optionally, the components of the Jacobian, that is, the sparse matrices of partial derivatives with respect to each real component of the state. The voltage portion, which depends on the formulation (polar vs cartesian), is delegated to the `port_inj_current_jac()` method implemented by the appropriate subclass.

The state can be provided as a stacked aggregate of the state variables (port voltages and non-voltage states) for the full collection of network model elements of this type, or as the combined state for the entire network.

Inputs

- **x_** (complex double) – state vector \mathbf{x}
- **sysx** (0 or 1) – which state is provided in $\mathbf{x}_$
 - 0 – class aggregate state
 - 1 – (default) full system state
- **idx** (integer) – (optional) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

- **I** (complex double) – vector of port complex current injections, $\mathbf{g}^I(\mathbf{x})$
- **Iv1** (complex double) – Jacobian of port complex current injections w.r.t 1st voltage component, \mathbf{g}_θ^I (polar) or \mathbf{g}_u^I (cartesian)
- **Iv2** (complex double) – Jacobian of port complex current injections w.r.t 2nd voltage component, \mathbf{g}_ν^I (polar) or \mathbf{g}_w^I (cartesian)
- **Izr** (complex double) – Jacobian of port complex current injections w.r.t real part of non-voltage state, $\mathbf{g}_{z_r}^I$
- **Izi** (complex double) – Jacobian of port complex current injections w.r.t imaginary part of non-voltage state, $\mathbf{g}_{z_i}^I$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_power\(\)](#) (page 79).

port_inj_power(x_, sysx, idx)

Compute port complex power injections from network state.

```
S = nme.port_inj_power(x_, sysx)
S = nme.port_inj_power(x_, sysx, idx)
```

(continues on next page)

(continued from previous page)

```
[S, Sv1, Sv2] = nme.port_inj_power(...)
[S, Sv1, Sv2, Szr, Szi] = nme.port_inj_power(...)
```

Compute the complex power injections for all or a selected subset of ports and, optionally, the components of the Jacobian, that is, the sparse matrices of partial derivatives with respect to each real component of the state. The voltage portion, which depends on the formulation (polar vs cartesian), is delegated to the `port_inj_power_jac()` method implemented by the appropriate subclass.

The state can be provided as a stacked aggregate of the state variables (port voltages and non-voltage states) for the full collection of network model elements of this type, or as the combined state for the entire network.

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **sysx** (*0 or 1*) – which state is provided in **x_**
 - 0 – class aggregate state
 - 1 – (*default*) full system state
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

- **S** (*complex double*) – vector of port complex power injections, $\mathbf{g}^S(\mathbf{x})$
- **Sv1** (*complex double*) – Jacobian of port complex power injections w.r.t 1st voltage component, \mathbf{g}_θ^S (polar) or \mathbf{g}_u^S (cartesian)
- **Sv2** (*complex double*) – Jacobian of port complex power injections w.r.t 2nd voltage component, \mathbf{g}_ν^S (polar) or \mathbf{g}_w^S (cartesian)
- **Szr** (*complex double*) – Jacobian of port complex power injections w.r.t real part of non-voltage state, $\mathbf{g}_{z_r}^S$
- **Szi** (*complex double*) – Jacobian of port complex power injections w.r.t imaginary part of non-voltage state, $\mathbf{g}_{z_i}^S$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_current\(\)](#) (page 79).

port_inj_current_hess(x_, lam, sysx, idx)

Compute Hessian of port current injections from network state.

```
H = nme.port_inj_current_hess(x_, lam)
H = nme.port_inj_current_hess(x_, lam, sysx)
H = nme.port_inj_current_hess(x_, lam, sysx, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the port current injection Jacobian by a vector $\boldsymbol{\lambda}$.

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **lam** (*double*) – vector $\boldsymbol{\lambda}$ of multipliers, one for each port
- **sysx** (*0 or 1*) – which state is provided in **x_**
 - 0 – class aggregate state
 - 1 – (*default*) full system state
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

H (*complex double*) – sparse Hessian matrix of port complex current injections corresponding to specified $\boldsymbol{\lambda}$, namely $\mathbf{g}_{\mathbf{xx}}^I(\boldsymbol{\lambda})$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_current\(\)](#) (page 79).

port_inj_power_hess(*x_*, *lam*, *sysx*, *idx*)

Compute Hessian of port power injections from network state.

```
H = nme.port_inj_power_hess(x_, lam)
H = nme.port_inj_power_hess(x_, lam, sysx)
H = nme.port_inj_power_hess(x_, lam, sysx, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the port power injection Jacobian by a vector λ .

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **lam** (*double*) – vector λ of multipliers, one for each port
- **sysx** (*0 or 1*) – which state is provided in **x_**
 - 0 – class aggregate state
 - 1 – (*default*) full system state
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

H (*complex double*) – sparse Hessian matrix of port complex power injections corresponding to specified λ , namely $\mathbf{g}_{\mathbf{xx}}^S(\lambda)$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_power\(\)](#) (page 79).

port_inj_current_jac(*n*, *v_*, *Y*, *M*, *invdiagvic*, *diagSlincJ*)

Abstract method to compute voltage-related Jacobian terms.

Called by [port_inj_current\(\)](#) (page 79) to compute voltage-related Jacobian terms. See [mp.form_acc.port_inj_current_jac\(\)](#) (page 86) and [mp.form_acp.port_inj_current_jac\(\)](#) (page 90) for details.

port_inj_current_hess_v(*x_*, *lam*, *v_*, *z_*, *diaginvic*, *Y*, *M*, *diagSlincJ*, *diamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by [port_inj_current_hess\(\)](#) (page 80) to compute voltage-related Hessian terms. See [mp.form_acc.port_inj_current_hess_v\(\)](#) (page 86) and [mp.form_acp.port_inj_current_hess_v\(\)](#) (page 90) for details.

port_inj_current_hess_vz(*x_*, *lam*, *v_*, *z_*, *diaginvic*, *N*, *diamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by [port_inj_current_hess\(\)](#) (page 80) to compute voltage/non-voltage-related Hessian terms. See [mp.form_acc.port_inj_current_hess_vz\(\)](#) (page 86) and [mp.form_acp.port_inj_current_hess_vz\(\)](#) (page 90) for details.

port_inj_power_jac(*n*, *v_*, *Y*, *M*, *diagv*, *diagvi*, *diagIlincJ*)

Abstract method to compute voltage-related Jacobian terms.

Called by [port_inj_power\(\)](#) (page 79) to compute voltage-related Jacobian terms. See [mp.form_acc.port_inj_power_jac\(\)](#) (page 87) and [mp.form_acp.port_inj_power_jac\(\)](#) (page 90) for details.

port_inj_power_hess_v(*x_*, *lam*, *v_*, *z_*, *diagvi*, *Y*, *M*, *diagIlincJ*, *diamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_power_hess()` (page 81) to compute voltage-related Hessian terms. See `mp.form_acc.port_inj_power_hess_v()` (page 87) and `mp.form_acp.port_inj_power_hess_v()` (page 90) for details.

port_inj_power_hess_vz(*x_*, *lam*, *v_*, *z_*, *diagvi*, *L*, *diamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_power_hess()` (page 81) to compute voltage/non-voltage-related Hessian terms. See `mp.form_acc.port_inj_power_hess_vz()` (page 87) and `mp.form_acp.port_inj_power_hess_vz()` (page 91) for details.

port_apparent_power_lim_fcn(*x_*, *nm*, *idx*, *hmax*)

Compute port squared apparent power injection constraints.

```
h = nme.port_apparent_power_lim_fcn(x_, nm, idx, hmax)
[h, dh] = nme.port_apparent_power_lim_fcn(x_, nm, idx, hmax)
```

Compute constraint function and optionally the Jacobian for the limit on port squared apparent power injections based on complex outputs of `port_inj_power()` (page 79).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **nm** (`mp.net_model` (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports
- **hmax** (*double*) – vector of squared apparent power limits

Outputs

- **h** (*double*) – constraint function, $\mathbf{h}^{\text{flow}}(\mathbf{x})$
- **dh** (*double*) – constraint Jacobian, $\mathbf{h}_x^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()` (page 79).

port_active_power_lim_fcn(*x_*, *nm*, *idx*, *hmax*)

Compute port active power injection constraints.

```
h = nme.port_active_power_lim_fcn(x_, nm, idx, hmax)
[h, dh] = nme.port_active_power_lim_fcn(x_, nm, idx, hmax)
```

Compute constraint function and optionally the Jacobian for the limit on port active power injections based on complex outputs of `port_inj_power()` (page 79).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **nm** (`mp.net_model` (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports
- **hmax** (*double*) – vector of active power limits

Outputs

- **h** (*double*) – constraint function, $\mathbf{h}^{\text{flow}}(\mathbf{x})$
- **dh** (*double*) – constraint Jacobian, $\mathbf{h}_x^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()` (page 79).

port_active_power2_lim_fcn(*x_*, *nm*, *idx*, *hmax*)

Compute port squared active power injection constraints.

```
h = nme.port_active_power2_lim_fcn(x_, nm, idx, hmax)
[h, dh] = nme.port_active_power2_lim_fcn(x_, nm, idx, hmax)
```

Compute constraint function and optionally the Jacobian for the limit on port squared active power injections based on complex outputs of [port_inj_power\(\)](#) (page 79).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports
- **hmax** (*double*) – vector of squared active power limits

Outputs

- **h** (*double*) – constraint function, $\mathbf{h}^{\text{flow}}(\mathbf{x})$
- **dh** (*double*) – constraint Jacobian, $\mathbf{h}_x^{\text{flow}}$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_power\(\)](#) (page 79).

port_current_lim_fcn(*x_, nm, idx, hmax*)

Compute port squared current injection constraints.

```
h = nme.port_current_lim_fcn(x_, nm, idx, hmax)
[h, dh] = nme.port_current_lim_fcn(x_, nm, idx, hmax)
```

Compute constraint function and optionally the Jacobian for the limit on port squared current injections based on complex outputs of [port_inj_current\(\)](#) (page 79).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports
- **hmax** (*double*) – vector of squared current limits

Outputs

- **h** (*double*) – constraint function, $\mathbf{h}^{\text{flow}}(\mathbf{x})$
- **dh** (*double*) – constraint Jacobian, $\mathbf{h}_x^{\text{flow}}$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_current\(\)](#) (page 79).

port_apparent_power_lim_hess(*x_, lam, nm, idx*)

Compute port squared apparent power injection Hessian.

```
d2H = nme.port_apparent_power_lim_hess(x_, lam, nm, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\mu}$. Results are based on the complex outputs of [port_inj_power\(\)](#) (page 79) and [port_inj_power_hess\(\)](#) (page 81).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Output**d2H** (*double*) – sparse constraint Hessian matrix, $h_{xx}^{\text{flow}}(\mu)$ For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.See also [port_inj_power\(\)](#) (page 79), [port_inj_power_hess\(\)](#) (page 81).**port_active_power_lim_hess**(*x_*, *lam*, *nm*, *idx*)

Compute port active power injection Hessian.

```
d2H = nme.port_active_power_lim_hess(x_, lam, nm, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector μ . Results are based on the complex outputs of [port_inj_power\(\)](#) (page 79) and [port_inj_power_hess\(\)](#) (page 81).

Inputs

- **x_** (*complex double*) – state vector x
- **lam** (*double*) – vector μ of multipliers, one for each port
- **nm** (*mp.net_model* (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Output**d2H** (*double*) – sparse constraint Hessian matrix, $h_{xx}^{\text{flow}}(\mu)$ For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.See also [port_inj_power\(\)](#) (page 79), [port_inj_power_hess\(\)](#) (page 81).**port_active_power2_lim_hess**(*x_*, *lam*, *nm*, *idx*)

Compute port squared active power injection Hessian.

```
d2H = nme.port_active_power2_lim_hess(x_, lam, nm, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector μ . Results are based on the complex outputs of [port_inj_power\(\)](#) (page 79) and [port_inj_power_hess\(\)](#) (page 81).

Inputs

- **x_** (*complex double*) – state vector x
- **lam** (*double*) – vector μ of multipliers, one for each port
- **nm** (*mp.net_model* (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Output**d2H** (*double*) – sparse constraint Hessian matrix, $h_{xx}^{\text{flow}}(\mu)$ For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.See also [port_inj_power\(\)](#) (page 79), [port_inj_power_hess\(\)](#) (page 81).**port_current_lim_hess**(*x_*, *lam*, *nm*, *idx*)

Compute port squared current injection Hessian.

```
d2H = nme.port_current_lim_hess(x_, lam, nm, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector μ . Results are based on the complex outputs of [port_inj_current\(\)](#) (page 79) and [port_inj_current_hess\(\)](#) (page 80).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x}
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Output

d2H (*double*) – sparse constraint Hessian matrix, $\mathbf{h}_{xx}^{\text{flow}}(\boldsymbol{\mu})$

For details on the derivations of the formulas used, see [MATPOWER Technical Note 5](#).

See also [port_inj_current\(\)](#) (page 79), [port_inj_current_hess\(\)](#) (page 80).

aux_data_va_vm(ad)

Abstract method to return voltage angles/magnitudes from auxiliary data.

```
[va, vm] = nme.aux_data_va_vm(ad)
```

Input

ad (*struct*) – struct of auxiliary data

Outputs

- **va** (*double*) – vector of voltage angles corresponding to voltage information stored in auxiliary data
- **vm** (*double*) – vector of voltage magnitudes corresponding to voltage information stored in auxiliary data

Implemented by [mp.form_acc.aux_data_va_vm\(\)](#) (page 87) and [mp.form_acp.aux_data_va_vm\(\)](#) (page 91).

mp.form_acc**class mp.form_acc**

Bases: [mp.form_ac](#) (page 76)

[mp.form_acc](#) (page 85) - Base class for MATPOWER AC cartesian **formulations**.

Used as a mix-in class for all **network model element** classes with an AC network model formulation with a **cartesian** representation for voltages. That is, each concrete network model element class with an AC cartesian formulation must inherit, at least indirectly, from both [mp.nm_element](#) (page 110) and [mp.form_acc](#) (page 85).

Provides implementation of evaluation of voltage-related Jacobian and Hessian terms needed by some [mp.form_ac](#) (page 76) methods.

mp.form_dc Methods:

- [form_name\(\)](#) (page 86) - get char array w/name of formulation ('AC-cartesian')
- [form_tag\(\)](#) (page 86) - get char array w/short label of formulation ('acc')
- [model_vvars\(\)](#) (page 86) - get cell array of names of voltage state variables ({'vr', 'vi'})
- [port_inj_current_jac\(\)](#) (page 86) - compute voltage-related terms of current injection Jacobian
- [port_inj_current_hess_v\(\)](#) (page 86) - compute voltage-related terms of current injection Hessian
- [port_inj_current_hess_vz\(\)](#) (page 86) - compute voltage/non-voltage-related terms of current injection Hessian

- `port_inj_power_jac()` (page 87) - compute voltage-related terms of power injection Jacobian
- `port_inj_power_hess_v()` (page 87) - compute voltage-related terms of power injection Hessian
- `port_inj_power_hess_vz()` (page 87) - compute voltage/non-voltage-related terms of power injection Hessian
- `aux_data_va_vm()` (page 87) - return voltage angles/magnitudes from auxiliary data
- `va_fcn()` (page 87) - compute voltage angle constraints and Jacobian
- `va_hess()` (page 88) - compute voltage angle Hessian
- `vm2_fcn()` (page 88) - compute squared voltage magnitude constraints and Jacobian
- `vm2_hess()` (page 88) - compute squared voltage magnitude Hessian

For more details, see the `sec_nm_formulations_ac` section in the *MATPOWER Developer's Manual* and the derivations in *MATPOWER Technical Note 5*.

See also `mp.form` (page 74), `mp.form_ac` (page 76), `mp.form_acp` (page 89), `mp.nm_element` (page 110).

Method Summary

`form_name()`

Get user-readable name of formulation, i.e. 'AC-cartesian'.

See `mp.form.form_name()` (page 75).

`form_tag()`

Get short label of formulation, i.e. 'acc'.

See `mp.form.form_tag()` (page 75).

`model_vvars()`

Get cell array of names of voltage state variables, i.e. {'vr', 'vi'}.

See `mp.form.model_vvars()` (page 75).

`port_inj_current_jac(n, v_, Y, M, invdiagvic, diagSlineJ)`

Compute voltage-related terms of current injection Jacobian.

```
[Iu, Iw] = nme.port_inj_current_jac(n, v_, Y, M, invdiagvic, diagSlineJ)
```

Called by `mp.form_ac.port_inj_current()` (page 79) to compute voltage-related Jacobian terms.

`port_inj_current_hess_v(x_, lam, v_, z_, diaginvic, Y, M, diagSlineJ, dlamJ)`

Compute voltage-related terms of current injection Hessian.

```
[Iuu, Iuw, Iww] = nme.port_inj_current_hess_v(x_, lam)
[Iuu, Iuw, Iww] = nme.port_inj_current_hess_v(x_, lam, sysx)
[Iuu, Iuw, Iww] = nme.port_inj_current_hess_v(x_, lam, sysx, idx)
[...] = nme.port_inj_current_hess_vz(x_, lam, v_, z_, diaginvic, Y, M,
    diagSlineJ, dlamJ)
```

Called by `mp.form_ac.port_inj_current_hess()` (page 80) to compute voltage-related Hessian terms.

`port_inj_current_hess_vz(x_, lam, v_, z_, diaginvic, N, dlamJ)`

Compute voltage/non-voltage-related terms of current injection Hessian.

```
[Iuzr, Iuzi, Iwzr, Iwzi] = nme.port_inj_current_hess_vz(x_, lam)
[...] = nme.port_inj_current_hess_vz(x_, lam, sysx)
[...] = nme.port_inj_current_hess_vz(x_, lam, sysx, idx)
[...] = nme.port_inj_current_hess_vz(x_, lam, v_, z_, diaginvic, N, dlamJ)
```

Called by `mp.form_ac.port_inj_current_hess()` (page 80) to compute voltage/non-voltage-related Hessian terms.

port_inj_power_jac(*n*, *v_*, *Y*, *M*, *diagv*, *diagvi*, *diagIlineJ*)

Compute voltage-related terms of power injection Jacobian.

```
[Su, Sw] = nme.port_inj_power_jac(...)
```

Called by `mp.form_ac.port_inj_power()` (page 79) to compute voltage-related Jacobian terms.

port_inj_power_hess_v(*x_*, *lam*, *v_*, *z_*, *diagvi*, *Y*, *M*, *diagIlineJ*, *dlamJ*)

Compute voltage-related terms of power injection Hessian.

```
[Suu, Suw, Sww] = nme.port_inj_power_hess_v(x_, lam)
[Suu, Suw, Sww] = nme.port_inj_power_hess_v(x_, lam, sysx)
[Suu, Suw, Sww] = nme.port_inj_power_hess_v(x_, lam, sysx, idx)
[...] = nme.port_inj_power_hess_v(x_, lam, v_, z_, diagvi, Y, M, diagIlineJ,
↪ dlamJ)
```

Called by `mp.form_ac.port_inj_power_hess()` (page 81) to compute voltage-related Hessian terms.

port_inj_power_hess_vz(*x_*, *lam*, *v_*, *z_*, *diagvi*, *L*, *dlamJ*)

Compute voltage/non-voltage-related terms of power injection Hessian.

```
[Suzr, Suzi, Swzr, Swzi] = nme.port_inj_power_hess_vz(x_, lam)
[...] = nme.port_inj_power_hess_vz(x_, lam, sysx)
[...] = nme.port_inj_power_hess_vz(x_, lam, sysx, idx)
[...] = nme.port_inj_power_hess_vz(x_, lam, v_, z_, diagvi, L, dlamJ)
```

Called by `mp.form_ac.port_inj_power_hess()` (page 81) to compute voltage/non-voltage-related Hessian terms.

aux_data_va_vm(*ad*)

Return voltage angles/magnitudes from auxiliary data.

```
[va, vm] = nme.aux_data_va_vm(ad)
```

Converts from cartesian voltage data stored in *ad.vr* and *ad.vi*.

Input

ad (*struct*) – struct of auxiliary data

Outputs

- **va** (*double*) – vector of voltage angles corresponding to voltage information stored in auxiliary data
- **vm** (*double*) – vector of voltage magnitudes corresponding to voltage information stored in auxiliary data

va_fcn(*xx*, *idx*, *lim*)

Compute voltage angle constraints and Jacobian.

```
g = nme.va_fcn(xx, idx, lim)
[g, dg] = nme.va_fcn(xx, idx, lim)
```

Compute constraint function and optionally the Jacobian for voltage angle limits.

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in **xx{1}**, imaginary part in **xx{2}**
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all
- **lim** (*double or cell array of double*) – constraint bound(s), can be a vector, for equality constraints or an upper bound, or a cell array with {**va_lb**, **va_ub**} for dual-bound constraints

Outputs

- **g** (*double*) – constraint function, $g(x)$
- **dg** (*double*) – constraint Jacobian, g_x

va_hess(xx, lam, idx)

Compute voltage angle Hessian.

```
d2G = nme.va_hess(xx, lam, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of voltages. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector λ .

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in **xx{1}**, imaginary part in **xx{2}**
- **lam** (*double*) – vector λ of multipliers, one for each constraint
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all

Output

d2G (*double*) – sparse constraint Hessian, $g_{xx}(\lambda)$

vm2_fcn(xx, idx, lim)

Compute squared voltage magnitude constraints and Jacobian.

```
g = nme.vm2_fcn(xx, idx, lim)
[g, dg] = nme.vm2_fcn(xx, idx, lim)
```

Compute constraint function and optionally the Jacobian for squared voltage magnitude limits.

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in **xx{1}**, imaginary part in **xx{2}**
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all
- **lim** (*double or cell array of double*) – constraint bound(s), can be a vector, for equality constraints or an upper bound, or a cell array with {**vm2_lb**, **vm2_ub**} for dual-bound constraints

Outputs

- **g** (*double*) – constraint function, $g(x)$
- **dg** (*double*) – constraint Jacobian, g_x

vm2_hess(xx, lam, idx)

Compute squared voltage magnitude Hessian.

```
d2G = nme.vm2_hess(xx, lam, idx)
```

Compute a sparse Hessian matrix for all or a selected subset of voltages. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector λ .

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in `xx{1}`, imaginary part in `xx{2}`
- **lam** (*double*) – vector λ of multipliers, one for each constraint
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all

Output

d2G (*double*) – sparse constraint Hessian, $g_{xx}(\lambda)$

mp.form_acp

class mp.form_acp

Bases: [mp.form_ac](#) (page 76)

[mp.form_acp](#) (page 89) - Base class for MATPOWER AC polar **formulations**.

Used as a mix-in class for all **network model element** classes with an AC network model formulation with a **polar** representation for voltages. That is, each concrete network model element class with an AC polar formulation must inherit, at least indirectly, from both [mp.nm_element](#) (page 110) and [mp.form_acp](#) (page 89).

Provides implementation of evaluation of voltage-related Jacobian and Hessian terms needed by some [mp.form_ac](#) (page 76) methods.

mp.form_dc Methods:

- [form_name\(\)](#) (page 89) - get char array w/name of formulation ('AC-polar')
- [form_tag\(\)](#) (page 90) - get char array w/short label of formulation ('acp')
- [model_vvars\(\)](#) (page 90) - get cell array of names of voltage state variables ({'va', 'vm'})
- [port_inj_current_jac\(\)](#) (page 90) - compute voltage-related terms of current injection Jacobian
- [port_inj_current_hess_v\(\)](#) (page 90) - compute voltage-related terms of current injection Hessian
- [port_inj_current_hess_vz\(\)](#) (page 90) - compute voltage/non-voltage-related terms of current injection Hessian
- [port_inj_power_jac\(\)](#) (page 90) - compute voltage-related terms of power injection Jacobian
- [port_inj_power_hess_v\(\)](#) (page 90) - compute voltage-related terms of power injection Hessian
- [port_inj_power_hess_vz\(\)](#) (page 91) - compute voltage/non-voltage-related terms of power injection Hessian
- [aux_data_va_vm\(\)](#) (page 91) - return voltage angles/magnitudes from auxiliary data

For more details, see the `sec_nm_formulations_ac` section in the *MATPOWER Developer's Manual* and the derivations in *MATPOWER Technical Note 5*.

See also [mp.form](#) (page 74), [mp.form_ac](#) (page 76), [mp.form_acc](#) (page 85), [mp.nm_element](#) (page 110).

Method Summary

form_name()

Get user-readable name of formulation, i.e. 'AC-polar'.

See [mp.form.form_name\(\)](#) (page 75).

form_tag()

Get short label of formulation, i.e. 'acp'.

See [mp.form.form_tag\(\)](#) (page 75).

model_vvars()

Get cell array of names of voltage state variables, i.e. {'va', 'vm'}.

See [mp.form.model_vvars\(\)](#) (page 75).

port_inj_current_jac(*n, v_, Y, M, invdiagvic, diagSlincJ*)

Compute voltage-related terms of current injection Jacobian.

```
[Iva, Ivm] = nme.port_inj_current_jac(n, v_, Y, M, invdiagvic, diagSlincJ)
```

Called by [mp.form_ac.port_inj_current\(\)](#) (page 79) to compute voltage-related Jacobian terms.

port_inj_current_hess_v(*x_, lam, v_, z_, diaginvic, Y, M, diagSlincJ, dlamJ*)

Compute voltage-related terms of current injection Hessian.

```
[Ivava, Ivavm, Ivmvm] = nme.port_inj_current_hess_v(x_, lam)
[Ivava, Ivavm, Ivmvm] = nme.port_inj_current_hess_v(x_, lam, sysx)
[Ivava, Ivavm, Ivmvm] = nme.port_inj_current_hess_v(x_, lam, sysx, idx)
[...] = nme.port_inj_current_hess_vz(x_, lam, v_, z_, diaginvic, Y, M,
    ↪diagSlincJ, dlamJ)
```

Called by [mp.form_ac.port_inj_current_hess\(\)](#) (page 80) to compute voltage-related Hessian terms.

port_inj_current_hess_vz(*x_, lam, v_, z_, diaginvic, N, dlamJ*)

Compute voltage/non-voltage-related terms of current injection Hessian.

```
[Ivazr, Ivazi, Ivmzr, Ivmzi] = nme.port_inj_current_hess_vz(x_, lam)
[...] = nme.port_inj_current_hess_vz(x_, lam, sysx)
[...] = nme.port_inj_current_hess_vz(x_, lam, sysx, idx)
[...] = nme.port_inj_current_hess_vz(x_, lam, v_, z_, diaginvic, N, dlamJ)
```

Called by [mp.form_ac.port_inj_current_hess\(\)](#) (page 80) to compute voltage/non-voltage-related Hessian terms.

port_inj_power_jac(*n, v_, Y, M, diagv, diagvi, diagIlincJ*)

Compute voltage-related terms of power injection Jacobian.

```
[Sva, Svm] = nme.port_inj_power_jac(...)
```

Called by [mp.form_ac.port_inj_power\(\)](#) (page 79) to compute voltage-related Jacobian terms.

port_inj_power_hess_v(*x_, lam, v_, z_, diagvi, Y, M, diagIlincJ, dlamJ*)

Compute voltage-related terms of power injection Hessian.

```
[Svava, Svavm, Svmvm] = nme.port_inj_power_hess_v(x_, lam)
[Svava, Svavm, Svmvm] = nme.port_inj_power_hess_v(x_, lam, sysx)
[Svava, Svavm, Svmvm] = nme.port_inj_power_hess_v(x_, lam, sysx, idx)
[...] = nme.port_inj_power_hess_v(x_, lam, v_, z_, diagvi, Y, M, diagIlineJ,
↪ dlamJ)
```

Called by `mp.form_ac.port_inj_power_hess()` (page 81) to compute voltage-related Hessian terms.

port_inj_power_hess_vz(*x_*, *lam*, *v_*, *z_*, *diagvi*, *L*, *dlamJ*)

Compute voltage/non-voltage-related terms of power injection Hessian.

```
[Svazr, Svazi, Svmzr, Svmzi] = nme.port_inj_power_hess_vz(x_, lam)
[...] = nme.port_inj_power_hess_vz(x_, lam, sysx)
[...] = nme.port_inj_power_hess_vz(x_, lam, sysx, idx)
[...] = nme.port_inj_power_hess_vz(x_, lam, v_, z_, diagvi, L, dlamJ)
```

Called by `mp.form_ac.port_inj_power_hess()` (page 81) to compute voltage/non-voltage-related Hessian terms.

aux_data_va_vm(*ad*)

Return voltage angles/magnitudes from auxiliary data.

```
[va, vm] = nme.aux_data_va_vm(ad)
```

Simply returns voltage data stored in `ad.va` and `ad.vm`.

Input

ad (*struct*) – struct of auxiliary data

Outputs

- **va** (*double*) – vector of voltage angles corresponding to voltage information stored in auxiliary data
- **vm** (*double*) – vector of voltage magnitudes corresponding to voltage information stored in auxiliary data

mp.form_dc

class mp.form_dc

Bases: `mp.form` (page 74)

`mp.form_dc` (page 91) - Base class for MATPOWER DC formulations.

Used as a mix-in class for all **network model element** classes with a DC network model formulation. That is, each concrete network model element class with a DC formulation must inherit, at least indirectly, from both `mp.nm_element` (page 110) and `mp.form_dc` (page 91).

`mp.form_dc` (page 91) defines the port active power injection as a linear function of the state variables \mathbf{x} , that is, the voltage angles θ and non-voltage states \mathbf{z} , as

$$\begin{aligned} \mathbf{g}^P(\mathbf{x}) &= \begin{bmatrix} \underline{B} & \underline{K} \end{bmatrix} \mathbf{x} + \underline{p} \\ &= \underline{B}\theta + \underline{K}\mathbf{z} + \underline{p}, \end{aligned} \tag{3.6}$$

where \underline{B} , \underline{K} , and \underline{p} are the model parameters.

For more details, see the `sec_nm_formulations_dc` section in the *MATPOWER Developer's Manual* and the derivations in *MATPOWER Technical Note 5*.

mp.form_dc Properties:

- \underline{B} (page 92) - $n_p n_k \times n_n$ matrix \underline{B} of model parameters
- \underline{K} (page 92) - $n_p n_k \times n_z$ matrix \underline{K} of model parameters
- \underline{p} (page 92) - $n_p n_k \times 1$ vector \underline{p} of model parameters
- `params_ncols` - specify number of columns for each parameter

mp.form_dc Methods:

- `form_name()` (page 92) - get char array w/name of formulation ('DC')
- `form_tag()` (page 92) - get char array w/short label of formulation ('dc')
- `model_params()` (page 92) - get network model element parameters ({'B', 'K', 'p'})
- `model_vvars()` (page 92) - get cell array of names of voltage state variables ({'va'})
- `model_zvars()` (page 93) - get cell array of names of non-voltage state variables ({'z'})
- `port_inj_power()` (page 93) - compute port power injections from network state

See also `mp.form` (page 74), `mp.form_ac` (page 76), `mp.nm_element` (page 110).

Property Summary

B = []

(double) $n_p n_k \times n_n$ matrix \underline{B} of model parameter coefficients for θ

K = []

(double) $n_p n_k \times n_z$ matrix \underline{K} of model parameter coefficients for z

p = []

(double) $n_p n_k \times 1$ vector \underline{p} of model parameters

param_ncols = `struct('B',2,'K',3,'p',1)`

(struct) specify number of columns for each parameter, where

- 1 => single column (i.e. a vector)
- 2 => n_p columns
- 3 => n_z columns

Method Summary

form_name()

Get user-readable name of formulation, i.e. 'DC'.

See `mp.form.form_name()` (page 75).

form_tag()

Get short label of formulation, i.e. 'dc'.

See `mp.form.form_tag()` (page 75).

model_params()

Get cell array of names of model parameters, i.e. {'B', 'K', 'p'}.

See `mp.form.model_params()` (page 75).

model_vvars()

Get cell array of names of voltage state variables, i.e. {'va'}.

See [mp.form.model_vvars\(\)](#) (page 75).

model_zvars()

Get cell array of names of non-voltage state variables, i.e. {'z'}.

See [mp.form.model_zvars\(\)](#) (page 75).

port_inj_power(x, sysx, idx)

Compute port power injections from network state.

```
P = nme.port_inj_power(x, sysx, idx)
```

Compute the active power injections for all or a selected subset of ports.

The state can be provided as a stacked aggregate of the state variables (port voltages and non-voltage states) for the full collection of network model elements of this type, or as the combined state for the entire network.

Inputs

- **x** (*double*) – state vector \mathbf{x}
- **sysx** (*0 or 1*) – which state is provided in **x**
 - 0 – class aggregate state
 - 1 – (*default*) full system state
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns injections corresponding to all ports

Outputs

P (*double*) – vector of port power injections, $\mathbf{g}^P(\mathbf{x})$

mp.net_model**class mp.net_model**

Bases: [mp.nm_element](#) (page 110), [mp.element_container](#) (page 168), [mp_idx_manager](#)

[mp.net_model](#) (page 93) - Abstract base class for MATPOWER **network model** objects.

The network model defines the states of and connections between network elements, as well as the parameters and functions defining the relationships between states and port injections. A given network model implements a specific network model **formulation**, and defines sets of **nodes**, **ports**, and **states**.

A network model object is primarily a container for network model element ([mp.nm_element](#) (page 110)) objects and *is itself* a network model element. All network model classes inherit from [mp.net_model](#) (page 93) and therefore also from [mp.element_container](#) (page 168), [mp_idx_manager](#), and [mp.nm_element](#) (page 110). Concrete network model classes are also formulation-specific, inheriting from a corresponding subclass of [mp.form](#) (page 74).

By convention, network model variables are named **nm** and network model class names begin with **mp.net_model**.

mp.net_model Properties:

- [the_np](#) (page 95) - total number of ports
- [the_nz](#) (page 95) - total number of non-voltage states
- [nv](#) (page 95) - total number of (real) voltage variables

- `node` (page 95) - `mp_idx_manager` data for nodes
- `port` (page 95) - `mp_idx_manager` data for ports
- `state` (page 95) - `mp_idx_manager` data for non-voltage states

mp.net_model Methods:

- `name()` (page 95) - return name of this network element type ('network')
- `np()` (page 95) - return number of ports for this network element
- `nz()` (page 95) - return number of (*possibly complex*) non-voltage states for this network element
- `build()` (page 95) - create, add, and build network model element objects
- `add_nodes()` (page 95) - elements add nodes, then add corresponding voltage variables
- `add_states()` (page 96) - elements add states, then add corresponding state variables
- `build_params()` (page 96) - build incidence matrices, parameters, add ports for each element
- `stack_matrix_params()` (page 96) - form network matrix parameter by stacking corresponding element parameters
- `stack_vector_params()` (page 96) - form network vector parameter by stacking corresponding element parameters
- `add_vvars()` (page 97) - add voltage variable(s) for each network node
- `add_zvars()` (page 97) - add non-voltage state variable(s) for each network state
- `def_set_types()` (page 97) - define node, state, and port set types for `mp_idx_manager`
- `init_set_types()` (page 97) - initialize structures for tracking/indexing nodes, states, ports
- `display()` (page 97) - display the network model object
- `add_node()` (page 98) - add named set of nodes
- `add_port()` (page 98) - add named set of ports
- `add_state()` (page 98) - add named set of states
- `set_type_idx_map()` (page 98) - map node/port/state index back to named set & index within set
- `set_type_label()` (page 99) - create a user-readable label to identify a node, port, or state
- `add_var()` (page 99) - add a set of variables to the model
- `params_var()` (page 100) - return initial value, bounds, and variable type for variables
- `get_node_idx()` (page 101) - get index information for named node set
- `get_port_idx()` (page 101) - get index information for named port set
- `get_state_idx()` (page 101) - get index information for named state set
- `node_types()` (page 101) - get node type information
- `ensure_ref_node()` (page 102) -
- `set_node_type_ref()` (page 102) - make the specified node a reference node
- `set_node_type_pv()` (page 103) - make the specified node a PV node
- `set_node_type_pq()` (page 103) - make the specified node a PQ node

See the `sec_net_model` section in the *MATPOWER Developer's Manual* for more information.

See also `mp.form` (page 74), `mp.nm_element` (page 110), `mp.task` (page 9), `mp.data_model` (page 30), `mp.math_model` (page 124).

Property Summary

the_np = 0
(integer) total number of ports

the_nz = 0
(integer) total number of non-voltage states

nv = 0
(integer) total number of (real) voltage variables

node = []
(struct) `mp_idx_manager` data for nodes

port = []
(struct) `mp_idx_manager` data for ports

state = []
(struct) `mp_idx_manager` data for non-voltage states

Method Summary

name()
 Return the name of this network element type ('network').

```
name = nm.name()
```

np()
 Return the number of ports for this network element.

```
np = nm.np()
```

nz()
 Return the number of (*possibly complex*) non-voltage states for this network element.

```
nz = nm.nz()
```

build(dm)
 Create, add, and *build()* (page 95) network model element objects.

```
nm.build(dm)
```

Input

dm (*mp.data_model* (page 30)) – data model object

Create and add network model element objects, add nodes and states, and build the parameters for all elements.

See also *add_nodes()* (page 95), *add_states()* (page 96), *build_params()* (page 96).

add_nodes(nm, dm)

Elements add nodes, then add corresponding voltage variables.

```
nm.add_nodes(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Each element can add its nodes, then the network model itself can add additional nodes, and finally corresponding voltage variables are added for each node.

See also [add_vvars\(\)](#) (page 97), [add_states\(\)](#) (page 96).

add_states(nm, dm)

Elements add states, then add corresponding state variables.

```
nm.add_states(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Each element can add its states, then corresponding non-voltage state variables are added for each state.

See also [add_zvars\(\)](#) (page 97), [add_nodes\(\)](#) (page 95).

build_params(nm, dm)

Build incidence matrices and parameters, and add ports for each element.

```
nm.build_params(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

For each element, build connection and state variable incidence matrices and element parameters, and add ports. Then construct the full network connection and state variable incidence matrices.

stack_matrix_params(name, vnotz)

Form network matrix parameter by stacking corresponding element parameters.

```
M = nm.stack_matrix_params(name, vnotz)
```

Inputs

- **name** (*char array*) – name of the parameter of interest
- **vnotz** (*boolean*) – true if columns of parameter correspond to voltage variables, false otherwise

Output

M (*double*) – matrix parameter of interest for the full network

A given matrix parameter (e.g. **Y**) for the full network is formed by stacking the corresponding matrix parameters for each element along the matrix block diagonal.

stack_vector_params(name)

Form network vector parameter by stacking corresponding element parameters.

```
v = nm.stack_vector_params(name)
```

Input

name (*char array*) – name of the parameter of interest

Output

v (*double*) – vector parameter of interest for the full network

A given vector parameter (e.g. *s*) for the full network is formed by vertically stacking the corresponding vector parameters for each element.

add_vvars(nm, dm, idx)

Add voltage variable(s) for each network node.

```
nm.add_vvars(nm, dm)
nm.add_vvars(nm, dm, idx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index for name and indexed variables (*currently unused here*)

Also updates the *nv* property.

See also [add_zvars\(\)](#) (page 97), [add_nodes\(\)](#) (page 95).

add_zvars(nm, dm, idx)

Add non-voltage state variable(s) for each network state.

```
nm.add_zvars(nm, dm)
nm.add_zvars(nm, dm, idx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*cell array*) – indices for named and indexed variables (*currently unused here*)

See also [add_vvars\(\)](#) (page 97), [add_states\(\)](#) (page 96).

def_set_types()

Define node, state, and port set types for *mp_idx_manager*.

```
nm.def_set_types()
```

Define the following set types:

- 'node' - NODES
- 'state' - STATES
- 'port' - PORTS

See also *mp_idx_manager*.

init_set_types()

Initialize structures for tracking/indexing nodes, states, ports.

```
nm.init_set_types()
```

See also *mp_idx_manager*.

display()

Display the network model object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the nodes, ports, states, voltage variables, non-voltage state variables, and network model elements.

See also `mp_idx_manager`.

add_node(*name*, *idx*, *N*)

Add named set of nodes.

```
nm.add_node(name, N)
nm.add_node(name, idx, N)
```

Inputs

- **name** (*char array*) – name for set of nodes
- **idx** (*cell array*) – indices for named, indexed set of nodes
- **N** (*integer*) – number of nodes in set

See also `mp_idx_manager.add_named_set()`.

add_port(*name*, *idx*, *N*)

Add named set of ports.

```
nm.add_port(name, N)
nm.add_port(name, idx, N)
```

Inputs

- **name** (*char array*) – name for set of ports
- **idx** (*cell array*) – indices for named, indexed set of ports
- **N** (*integer*) – number of ports in set

See also `mp_idx_manager.add_named_set()`.

add_state(*name*, *idx*, *N*)

Add named set of states.

```
nm.add_state(name, N)
nm.add_state(name, idx, N)
```

Inputs

- **name** (*char array*) – name for set of states
- **idx** (*cell array*) – indices for named, indexed set of states
- **N** (*integer*) – number of states in set

See also `mp_idx_manager.add_named_set()`.

set_type_idx_map(*set_type*, *idxs*, *dm*, *group_by_name*)

Map node/port/state index back to named set & index within set.

```
s = obj.set_type_idx_map(set_type)
s = obj.set_type_idx_map(set_type, idxs)
s = obj.set_type_idx_map(set_type, idxs, dm)
s = obj.set_type_idx_map(set_type, idxs, dm, group_by_name)
```

Inputs

- **set_type** (*char array*) – 'node', 'port', or 'state'
- **idxs** (*integer*) – vector of indices, defaults to `[1:ns]'`, where `ns` is the full dimension of the set corresponding to the all elements for the specified set type (i.e. node, port, or state)
- **dm** (*[mp.data_model](#)* (page 30)) – data model object

- **group_by_name** (*boolean*) – if true, results are consolidated, with a single entry in **s** for each unique name/idx pair, where the **i** and **j** fields are vectors

Output

s (*struct*) – index map of same dimensions as **idxs**, unless **group_by_name** is true, in which case it is 1 dimensional

Returns a struct of same dimensions as **idxs** specifying, for each index, the corresponding named set and element within the named set for the specified **set_type**. The return struct has the following fields:

- **name** : name of corresponding set
- **idx** : cell array of indices for the name, if named set is indexed
- **i** : index of element within the set
- **e** : external index (i.e. corresponding row in data model)
- **ID** : external ID (i.e. corresponding element ID in data model)
- **j** : (only if **group_by_name** == 1), corresponding index of set type, equal to a particular element of **idxs**

Examples:

```
s = nm.set_type_idx_map('node', 87, dm);
s = nm.set_type_idx_map('port', [38; 49; 93], dm);
s = nm.set_type_idx_map('state');
s = nm.set_type_idx_map('node', [], dm, 1);
```

set_type_label(*set_type, idxs, dm*)

Create a user-readable label to identify a node, port, or state.

```
label = nm.set_type_label(set_type, idxs)
label = nm.set_type_label(set_type, idxs, dm)
```

Inputs

- **set_type** (*char array*) – 'node', 'port', or 'state'
- **idxs** (*integer*) – vector of indices
- **dm** (*mp.data_model* (page 30)) – data model object

Output

label (*cell array*) – same dimensions as **idxs**, where each entry is a char array

Example:

```
labels = nm.set_type_label('port', [1;6;15;20], dm)

labels =

4x1 cell array

    {'gen 1'      }
    {'load 3'      }
    {'branch(1) 9'}
    {'branch(2) 5'}
```

add_var(*vtype, name, idx, varargin*)

Add a set of variables to the model.

```
nm.add_var(vtype, name, N, v0, v1, vu, vt)
nm.add_var(vtype, name, N, v0, v1, vu)
nm.add_var(vtype, name, N, v0, v1)
```

(continues on next page)

(continued from previous page)

```

nm.add_var(vtype, name, N, v0)
nm.add_var(vtype, name, N)
nm.add_var(vtype, name, idx_list, N, v0, vl, vu, vt)
nm.add_var(vtype, name, idx_list, N, v0, vl, vu)
nm.add_var(vtype, name, idx_list, N, v0, vl)
nm.add_var(vtype, name, idx_list, N, v0)
nm.add_var(vtype, name, idx_list, N)

```

Inputs

- **vtype** (*char array*) – variable type, must be a valid struct field name
- **name** (*char array*) – name of variable set
- **idx_list** (*cell array*) – optional index list
- **N** (*integer*) – number of variables in the set
- **v0** (*double*) – N x 1 col vector, initial value of variables, default is 0
- **vl** (*double*) – N x 1 col vector, lower bounds, default is -Inf
- **vu** (*double*) – N x 1 col vector, upper bounds, default is Inf
- **vt** (*char*) – scalar or 1 x N row vector, variable type, default is 'C', valid element values are:
 - 'C' - continuous
 - 'I' - integer
 - 'B' - binary

Essentially identical to the `add_var()` method from `opt_model` of MP-Opt-Model, with the addition of a variable type (`vtype`).

See also `opt_model.add_var()`.

params_var(vtype, name, idx)

Return initial value, bounds, and variable type for variables.

```

[v0, vl, vu] = nm.params_var(vtype)
[v0, vl, vu] = nm.params_var(vtype, name)
[v0, vl, vu] = nm.params_var(vtype, name, idx_list)
[v0, vl, vu, vt] = nm.params_var(...)

```

Inputs

- **vtype** (*char array*) – variable type, must be a valid struct field name
- **name** (*char array*) – name of variable set
- **idx_list** (*cell array*) – optional index list

Outputs

- **v0** (*double*) – N x 1 col vector, initial value of variables
- **vl** (*double*) – N x 1 col vector, lower bounds
- **vu** (*double*) – N x 1 col vector, upper bounds
- **vt** (*char*) – scalar or 1 x N row vector, variable type, valid element values are:
 - 'C' - continuous
 - 'I' - integer
 - 'B' - binary

Essentially identical to the `params_var()` method from `opt_model` of MP-Opt-Model, with the addition of a variable type (`vtype`).

Returns the initial value `v0`, lower bound `vl` and upper bound `vu` for the full variable vector, or for a specific named or named and indexed variable set. Optionally also returns a corresponding char vector `vt` of variable types, where 'C', 'I' and 'B' represent continuous, integer, and binary variables, respectively.

Examples:

```
[vr0, vrmin, vrmax] = obj.params_var('vr');
[pg0, pg_lb, pg_ub] = obj.params_var('zr', 'pg');
[zij0, zij_lb, zij_ub, ztype] = obj.params_var('zi', 'z', {i, j});
```

See also `opt_model.params_var()`.

get_node_idx(name)

Get index information for named node set.

```
[i1 iN] = nm.get_node_idx(name)
nidx = nm.get_node_idx(name)
```

Input

name (*char array*) – name of node set

Outputs

- **i1** (*integer*) – index of first node for name
- **iN** (*integer*) – index of last node for name
- **nidx** (*integer or cell array*) – indices of nodes for name, equal to either `[i1:iN]'` or `{[i1(1):iN(1)]', ..., [i1(n):iN(n)]'}`

get_port_idx(name)

Get index information for named port set.

```
[i1 iN] = nm.get_port_idx(name)
pidx = nm.get_port_idx(name)
```

Input

name (*char array*) – name of port set

Outputs

- **i1** (*integer*) – index of first port for name
- **iN** (*integer*) – index of last port for name
- **pidx** (*integer or cell array*) – indices of ports for name, equal to either `[i1:iN]'` or `{[i1(1):iN(1)]', ..., [i1(n):iN(n)]'}`

get_state_idx(name)

Get index information for named state set.

```
[i1 iN] = nm.get_state_idx(name)
sidx = nm.get_state_idx(name)
```

Input

name (*char array*) – name of state set

Outputs

- **i1** (*integer*) – index of first state for name
- **iN** (*integer*) – index of last state for name
- **sidx** (*integer or cell array*) – indices of states for name, equal to either `[i1:iN]'` or `{[i1(1):iN(1)]', ..., [i1(n):iN(n)]'}`

node_types(nm, dm, idx, skip_ensure_ref)

Get node type information.

```
ntv          = nm.node_types(nm, dm)
[ntv, by_elm] = nm.node_types(nm, dm)
```

(continues on next page)

(continued from previous page)

```
[ref, pv, pq] = nm.node_types(nm, dm)
[ref, pv, pq, by_elm] = nm.node_types(nm, dm)
... = nm.node_types(nm, dm, idx)
... = nm.node_types(nm, dm, idx, skip_ensure_ref)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index (*not used in base method*)
- **skip_ensure_ref** (*boolean*) – unless true, if there is no reference node, the first PV node will be converted to a new reference

Outputs

- **ntv** (*integer*) – node type vector, valid element values are:
 - [mp.NODE_TYPE.REF](#) (page 172)
 - [mp.NODE_TYPE.PV](#) (page 172)
 - [mp.NODE_TYPE.PQ](#) (page 172)
- **ref** (*integer*) – vector of indices of reference nodes
- **pv** (*integer*) – vector of indices of PV nodes
- **pq** (*integer*) – vector of indices of PQ nodes
- **by_elm** (*struct*) – **by_elm(k)** is struct for k-th node-creating element type, with fields:
 - 'name' - name of corresponding node-creating element type
 - 'ntv' - node type vector (if **by_elm** is 2nd output arg)
 - 'ref'/'pv'/'pq' - index vectors into elements of corresponding node-creating element type (if **by_elm** is 4th output arg)

See also [mp.NODE_TYPE](#) (page 172), [ensure_ref_node\(\)](#) (page 102).

ensure_ref_node(dm, ref, pv, pq)

Ensure there is at least one reference node.

```
[ref, pv, pq] = nm.ensure_ref_node(dm, ref, pv, pq)
ntv = nm.ensure_ref_node(dm, ntv)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **ref** (*integer*) – vector of indices of reference nodes
- **pv** (*integer*) – vector of indices of PV nodes
- **pq** (*integer*) – vector of indices of PQ nodes
- **ntv** (*integer*) – node type vector, valid element values are:
 - [mp.NODE_TYPE.REF](#) (page 172)
 - [mp.NODE_TYPE.PV](#) (page 172)
 - [mp.NODE_TYPE.PQ](#) (page 172)

Outputs

- **ref** (*integer*) – updated vector of indices of reference nodes
- **pv** (*integer*) – updated vector of indices of PV nodes
- **pq** (*integer*) – updated vector of indices of PQ nodes
- **ntv** (*integer*) – updated node type vector

set_node_type_ref(dm, idx)

Make the specified node a reference node.

```
nm.set_node_type_ref(dm, idx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type [mp.NODE_TYPE.REF](#) (page 172).

set_node_type_pv(dm, idx)

Make the specified node a PV node.

```
nm.set_node_type_pv(dm, idx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type [mp.NODE_TYPE.PV](#) (page 172).

set_node_type_pq(dm, idx)

Make the specified node a PQ node.

```
nm.set_node_type_pq(dm, idx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type [mp.NODE_TYPE.PQ](#) (page 172).

mp.net_model_ac

class mp.net_model_ac

Bases: [mp.net_model](#) (page 93)

[mp.net_model_ac](#) (page 103) - Abstract base class for MATPOWER AC **network model** objects.

Explicitly a subclass of [mp.net_model](#) (page 93) and implicitly assumed to be a subclass of [mp.form_ac](#) (page 76) as well.

mp.net_model_ac Properties:

- **zr** - vector of real part of complex non-voltage states, z_r
- **zi** - vector of imaginary part of complex non-voltage states, z_i

mp.net_model_ac Methods:

- [def_set_types\(\)](#) (page 104) - add non-voltage state variable set types for [mp_idx_manager](#)
- [build_params\(\)](#) (page 104) - build incidence matrices, parameters, add ports for each element
- [port_inj_nln\(\)](#) (page 104) - compute general nonlinear port injection functions and Jacobians
- [port_inj_nln_hess\(\)](#) (page 105) - compute general nonlinear port injection Hessian
- [nodal_complex_current_balance\(\)](#) (page 105) - compute nodal complex current balance constraints
- [nodal_complex_power_balance\(\)](#) (page 105) - compute nodal complex power balance constraints
- [nodal_complex_current_balance_hess\(\)](#) (page 106) - compute nodal complex current balance Hessian

- `nodal_complex_power_balance_hess()` (page 106) - compute nodal complex power balance Hessian
- `port_inj_soln()` (page 106) - compute the network port power injections at the solution
- `get_va()` (page 106) - get node voltage angle vector

See also `mp.net_model` (page 93), `mp.form` (page 74), `mp.form_ac` (page 76), `mp.nm_element` (page 110).

Method Summary

`def_set_types()`

Add non-voltage state variable set types for `mp_idx_manager`.

```
nm.def_set_types()
```

Add the following set types:

- 'zr' - NON-VOLTAGE VARS REAL (zr)
- 'zi' - NON-VOLTAGE VARS IMAG (zi)

See also `mp.net_model.def_set_types()` (page 97), `mp_idx_manager`.

`build_params(nm, dm)`

Build incidence matrices and parameters, and add ports for each element.

```
nm.build_params(nm, dm)
```

Inputs

- **nm** (`mp.net_model` (page 93)) – network model object
- **dm** (`mp.data_model` (page 30)) – data model object

Call the parent method to do most() of the work, then build the aggregate network model parameters and add the general nonlinear function terms, $s^{nl_n}(\mathbf{x})$ or $i^{nl_n}(\mathbf{x})$, for any elements that define them.

`port_inj_nln(si, x_, sysx, idx)`

Compute general nonlinear port injection functions and Jacobians

```
g = nm.port_inj_nln(si, x_, sysx, idx)
[g, gv1, gv2] = nm.port_inj_nln(si, x_, sysx, idx)
[g, gv1, gv2, gvr, gvi] = nm.port_inj_nln(si, x_, sysx, idx)
```

Compute and assemble the functions, and optionally Jacobians, for the general nonlinear injection functions $s^{nl_n}(\mathbf{x})$ and $i^{nl_n}(\mathbf{x})$ for the full aggregate network model, for all or a selected subset of ports.

Inputs

- **si** ('S' or 'I') – select power or current injection function:
 - 'S' for complex power $s^{nl_n}(\mathbf{x})$
 - 'I' for complex current $i^{nl_n}(\mathbf{x})$
- **x_** (complex double) – state vector \mathbf{x}
- **sysx** (0 or 1) – which state is provided in $\mathbf{x}_$
 - 0 – class aggregate state
 - 1 – (default) full system state
- **idx** (integer) – (optional) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

- **g** (complex double) – nonlinear injection function, $s^{nl_n}(\mathbf{x})$ (or $i^{nl_n}(\mathbf{x})$)
- **gv1** (complex double) – Jacobian w.r.t. 1st voltage variable, $s_{\theta}^{nl_n}$ or $s_u^{nl_n}$ (or $i_{\theta}^{nl_n}$ or $i_u^{nl_n}$)
- **gv2** (complex double) – Jacobian w.r.t. 2nd voltage variable, $s_v^{nl_n}$ or $s_w^{nl_n}$ (or $i_v^{nl_n}$ or $i_w^{nl_n}$)

- **g_{zr}** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $s_{z_r}^{nl}$ (or $i_{z_r}^{nl}$)
- **g_{zi}** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $s_{z_i}^{nl}$ (or $i_{z_i}^{nl}$)

See also [port_inj_nln_hess\(\)](#) (page 105).

port_inj_nln_hess(*si, x_, lam, sysx, idx*)

Compute general nonlinear port injection Hessian.

```
H = nm.port_inj_nln_hess(si, x_, lam)
H = nm.port_inj_nln_hess(si, x_, lam, sysx)
H = nm.port_inj_nln_hess(si, x_, lam, sysx, idx)
```

Compute and assemble the Hessian for the general nonlinear injection functions $s^{nl}(\mathbf{x})$ and $i^{nl}(\mathbf{x})$ for the full aggregate network model, for all or a selected subset of ports. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the corresponding Jacobian by a vector λ .

Inputs

- **si** ('S' or 'I') – select power or current injection function:
 - 'S' for complex power $s^{nl}(\mathbf{x})$
 - 'I' for complex current $i^{nl}(\mathbf{x})$
- **x_** (*complex double*) – state vector \mathbf{x}
- **lam** (*double*) – vector λ of multipliers, one for each port
- **sysx** (0 or 1) – which state is provided in **x_**
 - 0 – class aggregate state
 - 1 – (*default*) full system state
- **idx** (*integer*) – (*optional*) vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Output

H (*complex double*) – sparse Hessian matrix, $s_{xx}^{nl}(\lambda)$ or $i_{xx}^{nl}(\lambda)$

See also [port_inj_nln\(\)](#) (page 104).

nodal_complex_current_balance(*x_*)

Compute nodal complex current balance constraints.

```
G = nm.nodal_complex_current_balance(x_)
[G, Gv1, Gv2, Gzr, Gzi] = nm.nodal_complex_current_balance(x_)
```

Compute constraint function and optionally the Jacobian for the complex current balance equality constraints based on outputs of [mp.form_ac.port_inj_current\(\)](#) (page 79) and the node incidence matrix.

Input

x_ (*complex double*) – state vector \mathbf{x} (full system state)

Outputs

- **G** (*complex double*) – nodal complex current balance constraint function, $\mathbf{g}^{kcl}(\mathbf{x})$
- **Gv1** (*complex double*) – Jacobian w.r.t. 1st voltage variable, $\mathbf{g}_{\theta}^{kcl}$ or \mathbf{g}_u^{kcl}
- **Gv2** (*complex double*) – Jacobian w.r.t. 2nd voltage variable, \mathbf{g}_v^{kcl} or \mathbf{g}_w^{kcl}
- **Gzr** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $\mathbf{g}_{z_r}^{kcl}$
- **Gzi** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $\mathbf{g}_{z_i}^{kcl}$

See also [mp.form_ac.port_inj_current\(\)](#) (page 79), [nodal_complex_current_balance_hess\(\)](#) (page 106).

nodal_complex_power_balance(*x_*)

Compute nodal complex power balance constraints.

```
G = nm.nodal_complex_power_balance(x_)
[G, Gv1, Gv2, Gzr, Gzi] = nm.nodal_complex_power_balance(x_)
```

Compute constraint function and optionally the Jacobian for the complex power balance equality constraints based on outputs of `mp.form_ac.port_inj_power()` (page 79) and the node incidence matrix.

Input

x_ (*complex double*) – state vector \mathbf{x} (full system state)

Outputs

- **G** (*complex double*) – nodal complex power balance constraint function, $\mathbf{g}^{\text{kcl}}(\mathbf{x})$
- **Gv1** (*complex double*) – Jacobian w.r.t. 1st voltage variable, $\mathbf{g}_{\theta}^{\text{kcl}}$ or $\mathbf{g}_u^{\text{kcl}}$
- **Gv2** (*complex double*) – Jacobian w.r.t. 2nd voltage variable, $\mathbf{g}_{\nu}^{\text{kcl}}$ or $\mathbf{g}_w^{\text{kcl}}$
- **Gzr** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $\mathbf{g}_{z_r}^{\text{kcl}}$
- **Gzi** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $\mathbf{g}_{z_i}^{\text{kcl}}$

See also `mp.form_ac.port_inj_power()` (page 79), `nodal_complex_power_balance_hess()` (page 106).

`nodal_complex_current_balance_hess(x_, lam)`

Compute nodal complex current balance Hessian.

```
d2G = nm.nodal_complex_current_balance_hess(x_, lam)
```

Compute the Hessian of the nodal complex current balance constraint. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector λ . Based on `mp.form_ac.port_inj_current_hess()` (page 80).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x} (full system state)
- **lam** (*double*) – vector λ of multipliers, one for each node

Output

d2G (*complex double*) – sparse Hessian matrix, $\mathbf{g}_{xx}^{\text{kcl}}(\lambda)$

See also `mp.form_ac.port_inj_current_hess()` (page 80), `nodal_complex_current_balance()` (page 105).

`nodal_complex_power_balance_hess(x_, lam)`

Compute nodal complex power balance Hessian.

```
d2G = nm.nodal_complex_power_balance_hess(x_, lam)
```

Compute the Hessian of the nodal complex power balance constraint. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector λ . Based on `mp.form_ac.port_inj_power_hess()` (page 81).

Inputs

- **x_** (*complex double*) – state vector \mathbf{x} (full system state)
- **lam** (*double*) – vector λ of multipliers, one for each node

Output

d2G (*complex double*) – sparse Hessian matrix, $\mathbf{g}_{xx}^{\text{kcl}}(\lambda)$

See also `mp.form_ac.port_inj_power_hess()` (page 81), `nodal_complex_power_balance()` (page 105).

`port_inj_soln()`

Compute the network port power injections at the solution.

```
nm.port_inj_soln()
```

Takes the solved network state, computes the port power injections, and saves them in `nm.soln.gs_`.

`get_va(idx)`

Get node voltage angle vector.

```
va = nm.get_va()
va = nm.get_va(idx)
```

Get vector of node voltage angles for all or a selected subset of nodes. Values come from the solution if available, otherwise from the provided initial voltages.

Input

idx (*integer*) – index of subset of voltages of interest; if missing or empty, include all

Output

va (*double*) – vector of voltage angles

mp.net_model_acc

class mp.net_model_acc

Bases: [mp.net_model_ac](#) (page 103), [mp.form_acc](#) (page 85)

[mp.net_model_acc](#) (page 107) - Concrete class for MATPOWER AC cartesian **network model** objects.

This network model class and all of its network model element classes are specific to the AC cartesian formulation and therefore inherit from [mp.form_acc](#) (page 85).

mp.net_model_acc Properties:

- **vr** - vector of real part of complex voltage state variables, *u*
- **vi** - vector of imaginary part of complex voltage state variables, *w*

mp.net_model_acc Methods:

- [net_model_acc\(\)](#) (page 107) - constructor, assign default network model element classes
- [def_set_types\(\)](#) (page 107) - add voltage state variable set types for `mp_idx_manager`
- [initial_voltage_angle\(\)](#) (page 108) - get vector of initial node voltage angles

See also [mp.net_model_ac](#) (page 103), [mp.net_model](#) (page 93), [mp.form_acc](#) (page 85), [mp.form_ac](#) (page 76), [mp.form](#) (page 74), [mp.nm_element](#) (page 110).

Constructor Summary

net_model_acc()

Constructor, assign default network model element classes.

```
nm = net_model_acc()
```

This network model class and all of its network model element classes are specific to the AC cartesian formulation and therefore inherit from [mp.form_acc](#) (page 85).

Method Summary

def_set_types()

Add voltage state variable set types for `mp_idx_manager`.

```
nm.def_set_types()
```

Add the following set types:

- 'vr' - REAL VOLTAGE VARS (vr)
- 'vi' - IMAG VOLTAGE VARS (vi)

See also [mp.net_model_ac.def_set_types\(\)](#) (page 104), [mp.net_model.def_set_types\(\)](#) (page 97), [mp_idx_manager](#).

initial_voltage_angle(idx)

Get vector of initial node voltage angles.

```
va = nm.initial_voltage_angle()
va = nm.initial_voltage_angle(idx)
```

Get vector of initial node voltage angles for all or a selected subset of nodes.

Input

idx (*integer*) – index of subset of voltages of interest; if missing or empty, include all

Output

va (*double*) – vector of initial voltage angles

mp.net_model_acp

class mp.net_model_acp

Bases: [mp.net_model_ac](#) (page 103), [mp.form_acp](#) (page 89)

[mp.net_model_acp](#) (page 108) - Concrete class for MATPOWER AC polar **network model** objects.

This network model class and all of its network model element classes are specific to the AC polar formulation and therefore inherit from [mp.form_acp](#) (page 89).

mp.net_model_acp Properties:

- **va** - vector of angles of complex voltage state variables, θ
- **vm** - vector of magnitudes of complex voltage state variables, ν

mp.net_model_acp Methods:

- [net_model_acp\(\)](#) (page 108) - constructor, assign default network model element classes
- [def_set_types\(\)](#) (page 108) - add voltage state variable set types for [mp_idx_manager](#)
- [initial_voltage_angle\(\)](#) (page 109) - get vector of initial node voltage angles

See also [mp.net_model_ac](#) (page 103), [mp.net_model](#) (page 93), [mp.form_acp](#) (page 89), [mp.form_ac](#) (page 76), [mp.form](#) (page 74), [mp.nm_element](#) (page 110).

Constructor Summary

net_model_acp()

Constructor, assign default network model element classes.

```
nm = net_model_acp()
```

This network model class and all of its network model element classes are specific to the AC polar formulation and therefore inherit from [mp.form_acp](#) (page 89).

Method Summary

def_set_types()

Add voltage state variable set types for [mp_idx_manager](#).

```
nm.def_set_types()
```

Add the following set types:

- 'va' - VOLTAGE ANG VARS (va)
- 'vm' - VOLTAGE MAG VARS (vm)

See also [mp.net_model_ac.def_set_types\(\)](#) (page 104), [mp.net_model.def_set_types\(\)](#) (page 97), [mp_idx_manager](#).

initial_voltage_angle(idx)

Get vector of initial node voltage angles.

```
va = nm.initial_voltage_angle()
va = nm.initial_voltage_angle(idx)
```

Get vector of initial node voltage angles for all or a selected subset of nodes.

Input

idx (*integer*) – index of subset of voltages of interest; if missing or empty, include all

Output

va (*double*) – vector of initial voltage angles

mp.net_model_dc

class mp.net_model_dc

Bases: [mp.net_model](#) (page 93), [mp.form_dc](#) (page 91)

[mp.net_model_dc](#) (page 109) - Concrete class for MATPOWER DC **network model** objects.

This network model class and all of its network model element classes are specific to the DC formulation and therefore inherit from [mp.form_dc](#) (page 91).

mp.net_model_dc Properties:

- **va** (page 110) - vector of voltage states (voltage angles θ)
- **z** (page 110) - vector of non-voltage states z

mp.net_model_dc Methods:

- [net_model_dc\(\)](#) (page 109) - constructor, assign default network model element classes
- [def_set_types\(\)](#) (page 110) - add voltage and non-voltage variable set types for [mp_idx_manager](#)
- [build_params\(\)](#) (page 110) - build incidence matrices, parameters, add ports for each element
- [port_inj_soln\(\)](#) (page 110) - compute the network port injections at the solution

See also [mp.net_model](#) (page 93), [mp.form_dc](#) (page 91), [mp.form](#) (page 74), [mp.nm_element](#) (page 110).

Constructor Summary

net_model_dc()

Constructor, assign default network model element classes.

```
nm = net_model_dc()
```

This network model class and all of its network model element classes are specific to the DC formulation and therefore inherit from [mp.form_dc](#) (page 91).

Property Summary

va = []
(double) vector of voltage states (voltage angles θ)

z = []
(double) vector of non-voltage states z

Method Summary**def_set_types()**

Add voltage and non-voltage variable set types for `mp_idx_manager`.

```
nm.def_set_types()
```

Add the following set types:

- 'va' - VOLTAGE VARS (va)
- 'z' - NON-VOLTAGE VARS (z)

See also [mp.net_model.def_set_types\(\)](#) (page 97), `mp_idx_manager`.

build_params(nm, dm)

Build incidence matrices and parameters, and add ports for each element.

```
nm.build_params(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Call the parent method to do most () of the work, then build the aggregate network model parameters.

port_inj_soln()

Compute the network port injections at the solution.

```
nm.port_inj_soln()
```

Takes the solved network state, computes the port power injections, and saves them in `nm.soln.gp`.

3.4.2 Elements

mp.nm_element**class mp.nm_element**

Bases: `handle`

[mp.nm_element](#) (page 110) - Abstract base class for MATPOWER **network model element** objects.

A network model element object encapsulates all of the network model parameters for a particular element type. All network model element classes inherit from [mp.nm_element](#) (page 110) and also, like the container, from a formulation-specific subclass of [mp.form](#) (page 74). Each element type typically implements its own subclasses, which are further subclassed per formulation. A given network model element object contains the aggregate network model parameters for all online instances of that element type, stored in the set of matrices and vectors that correspond to the formulation.

By convention, network model element variables are named `nme` and network model element class names begin with `mp.nme`.

mp.mm_element Properties:

- `nk` (page 111) - number of elements of this type
- `C` (page 111) - stacked sparse element-node incidence matrices
- `D` (page 112) - stacked sparse incidence matrices for z -variables
- `soln` (page 112) - struct for storing solved states, quantities

mp.mm_element Methods:

- `name()` (page 112) - get name of element type, e.g. 'bus', 'gen'
- `np()` (page 112) - number of ports per element of this type
- `nn()` (page 112) - number of nodes per element, created by this element type
- `nz()` (page 112) - number of non-voltage state variables per element of this type
- `data_model_element()` (page 112) - get the corresponding data model element
- `math_model_element()` (page 113) - get the corresponding math model element
- `count()` (page 113) - get number of online elements in `dm`, set `nk`
- `add_nodes()` (page 113) - add nodes to network model
- `add_states()` (page 113) - add non-voltage states to network model
- `add_vvars()` (page 113) - add real-valued voltage variables to network object
- `add_zvars()` (page 114) - add real-valued non-voltage state variables to network object
- `build_params()` (page 114) - build model parameters from data model
- `get_nv_()` (page 114) - get number of (*possibly complex*) voltage variables
- `x2vz()` (page 114) - get port voltages and non-voltage states from combined state vector
- `node_indices()` (page 115) - construct node indices from data model element connection info
- `incidence_matrix()` (page 115) - construct stacked incidence matrix from set of index vectors
- `node_types()` (page 116) - get node type information
- `set_node_type_ref()` (page 116) - make the specified node a reference node
- `set_node_type_pv()` (page 116) - make the specified node a PV node
- `set_node_type_pq()` (page 117) - make the specified node a PQ node
- `display()` (page 117) - display the network model element object

See the `sec_nm_element` section in the *MATPOWER Developer's Manual* for more information.

See also `mp.net_model` (page 93).

Property Summary

`nk = 0`

(integer) number of elements of this type

C = []

(*sparse integer matrix*) stacked element-node incidence matrices, where $C(i, kk)$ is 1 if port j of element k is connected to node i , and $kk = k + (j-1)*np$

D = []

(*sparse integer matrix*) stacked incidence matrices for z -variables (non-voltage state variables), where $D(i, kk)$ is 1 if z -variable j of element k is the i -th system z -variable and $kk = k + (j-1)*nz$

soln

(*struct*) for storing solved states, quantities

Method Summary

name()

Get name of element type, e.g. 'bus', 'gen'.

```
name = nme.name()
```

Output

name (*char array*) – name of element type, must be a valid struct field name

Implementation provided by an element type specific subclass.

np()

Number of ports per element of this type.

```
np = nme.np()
```

Output

np (*integer*) – number of ports per element of this type

nn()

Number of nodes per element, created by this element type.

```
nn = nme.nn()
```

Output

nn (*integer*) – number of ports per element of this type

nz()

Number of non-voltage state variables per element of this type.

```
nz = nme.nz()
```

Output

nz (*integer*) – number of non-voltage state variables per element of this type

data_model_element(dm, name)

Get the corresponding data model element.

```
dme = nme.data_model_element(dm)
dme = nme.data_model_element(dm, name)
```

Inputs

- **dm** (*mp.data_model* (page 30)) – data model object
- **name** (*char array*) – (*optional*) name of element type (*default is name of this object*)

Output

dme (*mp.dm_element* (page 38)) – data model element object

math_model_element(*mm*, *name*)

Get the corresponding math model element.

```
nme = nme.math_model_element(mm)
nme = nme.math_model_element(mm, name)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – math model object
- **name** (*char array*) – (optional) name of element type (default is name of this object)

Output

nme ([mp.mm_element](#) (page 146)) – math model element object

count(*dm*)

Get number of online elements of this type in *dm*, set *nk*.

```
nk = nme.count(dm)
```

Input

dm ([mp.data_model](#) (page 30)) – data model object

Output

nk (*integer*) – number of online elements of this type

add_nodes(*nm*, *dm*)

Add nodes to network model for this element.

```
nme.add_nodes(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Add nodes to the network model object, based on value *nn* returned by [nn\(\)](#) (page 112). Calls the network model's [add_node\(\)](#) (page 98) *nn* times.

add_states(*nm*, *dm*)

Add non-voltage states to network model for this element.

```
nme.add_states(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Add non-voltage states to the network model object, based on value *nz* returned by [nz\(\)](#) (page 112). Calls the network model's [add_state\(\)](#) (page 98) *nz* times.

add_vvars(*nm*, *dm*, *idx*)

Add real-valued voltage variables to network object.

```
nme.add_vvars(nm, dm, idx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Add real-valued voltage variables (*v*-variables) to the network model object, for each port. Implementation depends on the specific formulation (i.e. subclass of [mp.form](#) (page 74)).

For example, consider an element with np ports and an AC formulation with polar voltage representation. The actual port voltages are complex, but this method would call the network model's `add_var()` (page 99) twice for each port, once for the voltage angle variables and once for the voltage magnitude variables.

Implemented by a formulation-specific subclass.

add_zvars(*nm, dm, idx*)

Add real-valued non-voltage state variables to network object.

```
nme.add_zvars(nm, dm, idx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*cell array*) – indices for named and indexed variables

Add real-valued non-voltage state variables (z -variables) to the network model object. Implementation depends on the specific formulation (i.e. subclass of [mp.form](#) (page 74)).

For example, consider an element with nz z -variables and a formulation in which these are complex. This method would call the network model's `add_var()` (page 99) twice for each complex z -variable, once for the variables representing the real part and once for the imaginary part.

Implemented by a formulation-specific subclass.

build_params(*nm, dm*)

Build model parameters from data model.

```
nme.build_params(nm, dm)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object

Construction of incidence matrices C and D are handled in this base class. Building of the formulation-specific model parameters must be implemented by a formulation-specific subclass. The subclass should call its parent in order to construct the incidence matrices.

See also [incidence_matrix\(\)](#) (page 115), [node_indices\(\)](#) (page 115).

get_nv_(*sysx*)

Get number of (*possibly complex*) voltage variables.

```
nv_ = nme.get_nv_(sysx)
```

Input

sysx (*boolean*) – if true the state $\mathbf{x}_\text{}$ refers to the full (*possibly complex*) system state (*all node voltages and system non-voltage states*), otherwise it is the state vector for this specific element type (*port voltages and element non-voltage states*)

Output

nv_ (*integer*) – number of (*possibly complex*) voltage variables in the state variable $\mathbf{x}_\text{}$, whose meaning depends on the **sysx** input

x2vz(*x_, sysx, idx*)

Get port voltages and non-voltage states from combined state vector.

```
[v_, z_, vi_] = nme.x2vz(x_, sysx, idx)
```

Inputs

- **x_** (*double*) – possibly complex state vector
- **sysx** (*boolean*) – if true the state **x_** refers to the full (possibly complex) system state (all node voltages and system non-voltage states), otherwise it is the state vector for this specific element type (port voltages and element non-voltage states)
- **idx** (*integer*) – vector of port indices of interest

Outputs

- **v_** (*double*) – vector of (possibly complex) port voltages
- **z_** (*double*) – vector of (possibly complex) non-voltage state variables
- **vi_** (*double*) – vector of (possibly complex) port voltages for selected ports only, as indexed by **idx**

This method extracts voltage and non-voltage states from a combined state vector, optionally with voltages for specific ports only.

Note, that this method can operate on multiple state vectors simultaneously, by specifying **x_** as a matrix. In this case, each output will have the same number of columns, one for each column of the input **x_**.

node_indices(*nm, dm, cxn_type, cxn_idx_prop, cxn_type_prop*)

Construct node indices from data model element connection info.

```
nidxs = nme.node_indices(nm, dm)
nidxs = nme.node_indices(nm, dm, cxn_type, cxn_idx_prop)
nidxs = nme.node_indices(nm, dm, cxn_type, cxn_idx_prop, cxn_type_prop)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **cxn_type** (*char array or cell array of char arrays*) – name(s) of type(s) of junction elements, i.e. node-creating elements (e.g. 'bus'), to which this element connects; see [mp.dm_element.cxn_type\(\)](#) (page 41) for more info
- **cxn_idx_prop** (*char array or cell array of char arrays*) – name(s) of property(ies) containing indices of junction elements that define connections (e.g. {'fbus', 'tbus'}); see [mp.dm_element.cxn_idx_prop\(\)](#) (page 41) for more info
- **cxn_type_prop** (*char array or cell array of char arrays*) – name(s) of properties containing type of junction elements for each connection, defaults to '' if **cxn_type** and **cxn_type_prop** are provided, but not **cxn_type_prop**; see [mp.dm_element.cxn_type_prop\(\)](#) (page 41) for more info

Output

nidxs (*cell array*) – 1 x *np* cell array of node index vectors for each port

This method constructs the node index vectors for each port. That is, element *p* of **nidxs** is the vector of indices of the nodes to which port *p* of these elements are connected. These node indices can be used to construct the element-node incidence matrices that form **C**.

By default, the connection information is obtained from the corresponding data model element, as described in the `sec_dm_element_cxn` section in the *MATPOWER Developer's Manual*.

See also [incidence_matrix\(\)](#) (page 115), [mp.dm_element.cxn_type\(\)](#) (page 41), [mp.dm_element.cxn_idx_prop\(\)](#) (page 41), [mp.dm_element.cxn_type_prop\(\)](#) (page 41).

incidence_matrix(*m, varargin*)

Construct stacked incidence matrix from set of index vectors.

```
CD = nme.incidence_matrix(m, idx1, idx2, ...)
```

Inputs

- **m** (*integer*) – total number of nodes or states
- **idx1** (*integer*) – index vector for nodes corresponding to this element's first port, or state variables corresponding to this element's first non-voltage state
- **idx2** (*integer*) – same as **idx1** for second port or non-voltage state, and so on

Output

CD (*sparse matrix*) – stacked incidence matrix (C for ports, D for states)

Forms an $m \times n$ incidence matrix for each input index vector **idx**, where n is the dimension of **idx**, and column j of the corresponding incidence matrix consists of all zeros with a 1 in row **idx**(j).

These incidence matrices are then stacked horizontally to form a single matrix return value.

node_types(*nm, dm, idx*)

Get node type information.

```
ntv          = nme.node_types(nm, dm)
[ref, pv, pq] = nme.node_types(nm, dm)
...          = nme.node_types(nm, dm, idx)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index (*not used in base method*)

Outputs

- **ntv** (*integer*) – node type vector, valid element values are:
 - [mp.NODE_TYPE.REF](#) (page 172)
 - [mp.NODE_TYPE.PV](#) (page 172)
 - [mp.NODE_TYPE.PQ](#) (page 172)
- **ref** (*integer*) – vector of indices of reference nodes
- **pv** (*integer*) – vector of indices of PV nodes
- **pq** (*integer*) – vector of indices of PQ nodes

See also [mp.NODE_TYPE](#) (page 172).

set_node_type_ref(*dm, idx*)

Make the specified node a reference node.

```
nme.set_node_type_ref(dm, idx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type [mp.NODE_TYPE.REF](#) (page 172).

Implementation provided by node-creating subclass.

set_node_type_pv(*dm, idx*)

Make the specified node a PV node.

```
nme.set_node_type_pv(dm, idx)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type [mp.NODE_TYPE.PV](#) (page 172).

Implementation provided by node-creating subclass.

set_node_type_pq(*dm*, *idx*)

Make the specified node a PQ node.

```
nme.set_node_type_pq(dm, idx)
```

Inputs

- **dm** (*mp.data_model* (page 30)) – data model object
- **idx** (*integer*) – index of node to modify, this is the internal network model element index

Set the specified node to type *mp.NODE_TYPE.PQ* (page 172).

Implementation provided by node-creating subclass.

display()

Display the network model element object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the elements, including total number of elements, nodes per element, ports per element, non-voltage state per element, formulation name, tag, and class, and names and dimensions of the model parameters.

mp.nme_branch

class mp.nme_branch

Bases: *mp.nm_element* (page 110)

mp.nme_branch (page 117) - Network model element abstract base class for branch.

Implements the network model element for branch elements, including transmission lines and transformers, with 2 ports per branch.

Method Summary

name()

np()

mp.nme_branch_ac

class mp.nme_branch_ac

Bases: *mp.nme_branch* (page 117)

mp.nme_branch_ac (page 117) - Network model element abstract base class for branch for AC formulations.

Implements building of the admittance parameter \underline{Y} for branches.

Method Summary

build_params(*nm*, *dm*)

Builds the admittance parameter \underline{Y} for branches.

mp.nme_branch_acc

class mp.nme_branch_acc

Bases: [mp.nme_branch_ac](#) (page 117), [mp.form_acc](#) (page 85)

[mp.nme_branch_acc](#) (page 118) - Network model element for branch for AC cartesian voltage formulations.

Implements functions for the voltage angle difference limits and their derivatives and inherits from [mp.form_acc](#) (page 85).

Method Summary

ang_diff_fcn(*xx*, *Aang*, *lang*, *uang*)

ang_diff_hess(*xx*, *lambda*, *Aang*)

mp.nme_branch_acp

class mp.nme_branch_acp

Bases: [mp.nme_branch_ac](#) (page 117), [mp.form_acp](#) (page 89)

[mp.nme_branch_acp](#) (page 118) - Network model element for branch for AC polar voltage formulations.

Inherits from [mp.form_acp](#) (page 89).

mp.nme_branch_dc

class mp.nme_branch_dc

Bases: [mp.nme_branch](#) (page 117), [mp.form_dc](#) (page 91)

[mp.nme_branch_dc](#) (page 118) - Network model element for branch for DC formulations.

Implements building of the branch parameters \underline{B} and \underline{p} , and inherits from [mp.form_dc](#) (page 91).

Method Summary

build_params(*nm*, *dm*)

mp.nme_bus

class mp.nme_bus

Bases: [mp.nm_element](#) (page 110)

[mp.nme_bus](#) (page 119) - Network model element abstract base class for bus.

Implements the network model element for bus elements, with 1 node per bus.

Implements node type methods.

Method Summary

name()

nn()

node_types(*nm, dm, idx*)

set_node_type_ref(*nm, dm, idx*)

set_node_type_pv(*nm, dm, idx*)

set_node_type_pq(*nm, dm, idx*)

mp.nme_bus_acc

class mp.nme_bus_acc

Bases: [mp.nme_bus](#) (page 119), [mp.form_acc](#) (page 85)

[mp.nme_bus_acc](#) (page 119) - Network model element for bus for AC cartesian voltage formulations.

Adds voltage variables V_r and V_i to the network model and inherits from [mp.form_acc](#) (page 85).

Method Summary

add_vvars(*nm, dm, idx*)

mp.nme_bus_acp

class mp.nme_bus_acp

Bases: [mp.nme_bus](#) (page 119), [mp.form_acp](#) (page 89)

[mp.nme_bus_acp](#) (page 119) - Network model element for bus for AC cartesian polar formulations.

Adds voltage variables V_a and V_m to the network model and inherits from [mp.form_acp](#) (page 89).

Method Summary

add_vvars(*nm, dm, idx*)

mp.nme_bus_dc

class mp.nme_bus_dc

Bases: [mp.nme_bus](#) (page 119), [mp.form_dc](#) (page 91)

[mp.nme_bus_dc](#) (page 120) - Network model element for bus for DC formulations.

Adds voltage variable V_a to the network model and inherits from [mp.form_dc](#) (page 91).

Method Summary

add_vvars(*nm, dm, idx*)

mp.nme_gen

class mp.nme_gen

Bases: [mp.nm_element](#) (page 110)

[mp.nme_gen](#) (page 120) - Network model element abstract base class for generator.

Implements the network model element for generator elements, with 1 port and 1 non-voltage state per generator.

Method Summary

name()

np()

nz()

mp.nme_gen_ac

class mp.nme_gen_ac

Bases: [mp.nme_gen](#) (page 120)

[mp.nme_gen_ac](#) (page 120) - Network model element abstract base class for generator for AC formulations.

Adds non-voltage state variables P_g and Q_g to the network model and builds the parameter \underline{N} .

Method Summary

add_zvars(*nm, dm, idx*)

build_params(*nm, dm*)

mp.nme_gen_acc**class mp.nme_gen_acc**

Bases: [mp.nme_gen_ac](#) (page 120), [mp.form_acc](#) (page 85)

[mp.nme_gen_acc](#) (page 121) - Network model element for generator for AC cartesian voltage formulations.

Inherits from [mp.form_acc](#) (page 85).

mp.nme_gen_acp**class mp.nme_gen_acp**

Bases: [mp.nme_gen_ac](#) (page 120), [mp.form_acp](#) (page 89)

[mp.nme_gen_acp](#) (page 121) - Network model element for generator for AC polar voltage formulations.

Inherits from [mp.form_acp](#) (page 89).

mp.nme_gen_dc**class mp.nme_gen_dc**

Bases: [mp.nme_gen](#) (page 120), [mp.form_dc](#) (page 91)

[mp.nme_gen_dc](#) (page 121) - Network model element for generator for DC formulations.

Adds non-voltage state variable P_g to the network model, builds the parameter \underline{K} , and inherits from [mp.form_dc](#) (page 91).

Method Summary

add_zvars(*nm*, *dm*, *idx*)

build_params(*nm*, *dm*)

mp.nme_load**class mp.nme_load**

Bases: [mp.nm_element](#) (page 110)

[mp.nme_load](#) (page 121) - Network model element abstract base class for load.

Implements the network model element for load elements, with 1 port per load.

Method Summary

name()

np()

mp.nme_load_ac

class mp.nme_load_ac

Bases: [mp.nme_load](#) (page 121)

[mp.nme_load_ac](#) (page 122) - Network model element abstract base class for load for AC formulations.

Builds the parameters \underline{s} and \underline{Y} and nonlinear functions $\mathbf{s}^{nl_n}(\mathbf{x})$ and $\mathbf{i}^{nl_n}(\mathbf{x})$.

Method Summary

build_params(*nm*, *dm*)

port_inj_current_nln(*Sd*, *x_*, *sysx*, *idx*)

port_inj_power_nln(*Sd*, *x_*, *sysx*, *idx*)

mp.nme_load_acc

class mp.nme_load_acc

Bases: [mp.nme_load_ac](#) (page 122), [mp.form_acc](#) (page 85)

[mp.nme_load_acc](#) (page 122) - Network model element for load for AC cartesian voltage formulations.

Inherits from [mp.form_acc](#) (page 85).

mp.nme_load_acp

class mp.nme_load_acp

Bases: [mp.nme_load_ac](#) (page 122), [mp.form_acp](#) (page 89)

[mp.nme_load_acp](#) (page 122) - Network model element for load for AC polar voltage formulations.

Inherits from [mp.form_acp](#) (page 89).

mp.nme_load_dc

class mp.nme_load_dc

Bases: [mp.nme_load](#) (page 121), [mp.form_dc](#) (page 91)

[mp.nme_load_dc](#) (page 122) - Network model element for load for DC formulations.

Builds the parameter \underline{p} and inherits from [mp.form_dc](#) (page 91).

Method Summary

build_params(*nm*, *dm*)

mp.nme_shunt

class mp.nme_shunt

Bases: [mp.nm_element](#) (page 110)

[mp.nme_shunt](#) (page 123) - Network model element abstract base class for shunt.

Implements the network model element for shunt elements, with 1 port per shunt.

Method Summary

name()

np()

mp.nme_shunt_ac

class mp.nme_shunt_ac

Bases: [mp.nme_shunt](#) (page 123)

[mp.nme_shunt_ac](#) (page 123) - Network model element abstract base class for shunt for AC formulations.

Builds the parameter Y.

Method Summary

build_params(*nm*, *dm*)

mp.nme_shunt_acc

class mp.nme_shunt_acc

Bases: [mp.nme_shunt_ac](#) (page 123), [mp.form_acc](#) (page 85)

[mp.nme_shunt_acc](#) (page 123) - Network model element for shunt for AC cartesian voltage formulations.

Inherits from [mp.form_acc](#) (page 85).

mp.nme_shunt_acp

class mp.nme_shunt_acp

Bases: [mp.nme_shunt_ac](#) (page 123), [mp.form_acp](#) (page 89)

[mp.nme_shunt_acp](#) (page 123) - Network model element for shunt for AC polar voltage formulations.

Inherits from [mp.form_acp](#) (page 89).

`mp.nme_shunt_dc`

`class mp.nme_shunt_dc`

Bases: `mp.nme_shunt` (page 123), `mp.form_dc` (page 91)

`mp.nme_shunt_dc` (page 124) - Network model element for shunt for DC formulations.

Builds the parameter `p` and inherits from `mp.form_dc` (page 91).

Method Summary

`build_params(nm, dm)`

3.5 Mathematical Model Classes

3.5.1 Containers

`mp.math_model`

`class mp.math_model`

Bases: `mp.element_container` (page 168), `opt_model`

`mp.math_model` (page 124) - Abstract base class for MATPOWER **mathematical model** objects.

The mathematical model, or math model, formulates and defines the mathematical problem to be solved. That is, it determines the variables, constraints, and objective that define the problem. This takes on different forms depending on the task (*e.g. power flow, optimal power flow, etc.*) and the formulation (*e.g. DC, AC-polar-power, etc.*).

A math model object is a container for math model element (`mp.mm_element` (page 146)) objects and it is also an MP-Opt-Model (`opt_model`) object. All math model classes inherit from `mp.math_model` (page 124) and therefore also from `mp.element_container` (page 168), `opt_model`, and `mp_idx_manager`. Concrete math model classes are task and formulation specific. They also sometimes inherit from abstract mix-in classes that are shared across tasks or formulations.

By convention, math model variables are named `mm` and math model class names begin with `mp.math_model`.

`mp.math_model` Properties:

- `aux_data` (page 125) - auxiliary data relevant to the model

`mp.math_model` Methods:

- `task_tag()` (page 125) - returns task tag, e.g. 'PF', 'OPF'
- `task_name()` (page 125) - returns task name, e.g. 'Power Flow', 'Optimal Power Flow'
- `form_tag()` (page 125) - returns network formulation tag, e.g. 'dc', 'acps'
- `form_name()` (page 125) - returns network formulation name, e.g. 'DC', 'AC-polar-power'
- `build()` (page 125) - create, add, and build math model element objects
- `display()` (page 126) - display the math model object
- `add_aux_data()` (page 126) - builds auxiliary data and adds it to the model

- [`build_base_aux_data\(\)`](#) (page 126) - builds base auxiliary data, including node types & variable initial values
- [`add_vars\(\)`](#) (page 126) - add variables to the model
- [`add_system_vars\(\)`](#) (page 126) - add system variables to the model
- [`add_constraints\(\)`](#) (page 127) - add constraints to the model
- [`add_system_constraints\(\)`](#) (page 127) - add system constraints to the model
- [`add_node_balance_constraints\(\)`](#) (page 127) - add node balance constraints to the model
- [`add_costs\(\)`](#) (page 127) - add costs to the model
- [`add_system_costs\(\)`](#) (page 128) - add system costs to the model
- [`solve_opts\(\)`](#) (page 128) - return an options struct to pass to the solver
- [`update_nm_vars\(\)`](#) (page 128) - update network model variables from math model solution
- [`data_model_update\(\)`](#) (page 129) - update data model from math model solution
- [`network_model_x_soln\(\)`](#) (page 129) - convert solved state from math model to network model solution

See the `sec_math_model` section in the *MATPOWER Developer's Manual* for more information.

See also [`mp.task`](#) (page 9), [`mp.data_model`](#) (page 30), [`mp.net_model`](#) (page 93).

Property Summary

`aux_data`

(*struct*) auxiliary data relevant to the model, e.g. can be passed to model constraint functions

Method Summary

`task_tag()`

Returns task tag, e.g. 'PF', 'OPF'.

```
tag = mm.task_tag()
```

`task_name()`

Returns task name, e.g. 'Power Flow', 'Optimal Power Flow'.

```
name = mm.task_name()
```

`form_tag()`

Returns network formulation tag, e.g. 'dc', 'acps'.

```
tag = mm.form_tag()
```

`form_name()`

Returns network formulation name, e.g. 'DC', 'AC-polar-power'.

```
name = mm.form_name()
```

`build(nm, dm, mpop)`

Create, add, and [`build\(\)`](#) (page 125) math model element objects.


```
mm.build(nm, dm, mpopt);
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Create and add network model objects, create and add auxiliary data, and add variables, constraints, and costs.

display()

Display the math model object.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

Displays the details of the variables, constraints, costs, and math model elements.

See also `mp_idx_manager`.

add_aux_data(nm, dm, mpopt)

Builds auxiliary data and adds it to the model.

```
mm.add_aux_data(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Calls the `build_aux_data()` method and assigns the result to the `aux_data` property. The base `build_aux_data()` method, which simply calls [build_base_aux_data\(\)](#) (page 126), is defined in [mp.mm_shared_pfcpf](#) (page 141) (and in [mp.math_model_opf](#) (page 134)) allowing it to be shared across math models for different tasks (PF and CPF).

build_base_aux_data(nm, dm, mpopt)

Builds base auxiliary data, including node types & variable initial values.

```
ad = mm.build_base_aux_data(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

ad (*struct*) – struct of auxiliary data

add_vars(nm, dm, mpopt)

Add variables to the model.

```
mm.add_vars(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system variables, then calls the [add_vars\(\)](#) (page 147) method for each math model element.

add_system_vars(*nm, dm, mpopt*)

Add system variables to the model.

```
mm.add_system_vars(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Variables which correspond to a specific math model element should be added by that element's [add_vars\(\)](#) (page 147) method. Other variables can be added by [add_system_vars\(\)](#) (page 126). In this base class this method does nothing.

add_constraints(*nm, dm, mpopt*)

Add constraints to the model.

```
mm.add_constraints(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system constraints, then calls the [add_constraints\(\)](#) (page 147) method for each math model element.

add_system_constraints(*nm, dm, mpopt*)

Add system constraints to the model.

```
mm.add_system_constraints(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Constraints which correspond to a specific math model element should be added by that element's [add_constraints\(\)](#) (page 147) method. Other constraints can be added by [add_system_constraints\(\)](#) (page 127). In this base class, it simply calls [add_node_balance_constraints\(\)](#) (page 127).

add_node_balance_constraints(*nm, dm, mpopt*)

Add node balance constraints to the model.

```
mm.add_node_balance_constraints(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

In this base class this method does nothing.

add_costs(*nm, dm, mpopt*)

Add costs to the model.

```
mm.add_costs(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system costs, then calls the [add_costs\(\)](#) (page 148) method for each math model element.

```
add_system_costs(nm, dm, mpopt)
```

Add system costs to the model.

```
mm.add_system_costs(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Costs which correspond to a specific math model element should be added by that element's [add_costs\(\)](#) (page 148) method. Other variables can be added by [add_system_costs\(\)](#) (page 128). In this base class this method does nothing.

```
solve_opts(nm, dm, mpopt)
```

Return an options struct to pass to the solver.

```
opt = mm.solve_opts(nm, dm, mpopt)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

opt (*struct*) – options struct for solver

In this base class, returns an empty struct.

```
update_nm_vars(mmx, nm)
```

Update network model variables from math model solution.

```
nm_vars = mm.update_nm_vars(mmx, nm)
```

Inputs

- **mmx** (*double*) – vector of math model variable **x**
- **nm** ([mp.net_model](#) (page 93)) – network model object

Output

nm_vars (*struct*) – updated network model variables

Returns a struct with the network model variables as fields. The `mm.aux_data.var_map` cell array is used to track mappings of math model variables back to network model variables. Each entry is itself a 7-element cell array of the form

```
{nm_var_type, nm_i1, nm_iN, nm_idx, mm_i1, mm_iN, mm_idx}
```

where

- **nm_var_type** - network model variable type (e.g. va, vm, zr, zi)
- **nm_i1** - starting index for network model variable type
- **nm_iN** - ending index for network model variable type

- `nm_idx` - vector of indices for network model variable type
- `mm_i1` - starting index for math model variable
- `mm_iN` - ending index for math model variable
- `mm_idx` - vector of indices for math model variable

Uses either `i1:iN` (if `i1` is not empty) or `idx` as the indices, unless both are empty, in which case it uses `':'`.

`data_model_update(nm, dm, mpopt)`

Update data model from math model solution.

```
dm = mm.data_model_update(nm, dm, mpopt)
```

Inputs

- `nm` ([mp.net_model](#) (page 93)) – network model object
- `dm` ([mp.data_model](#) (page 30)) – data model object
- `mpopt` (*struct*) – MATPOWER options struct

Output

`dm` ([mp.data_model](#) (page 30)) – updated data model object

Calls the [data_model_update\(\)](#) (page 148) method for each math model element.

`network_model_x_soln(nm)`

Convert solved state from math model to network model solution.

```
nm = mm.network_model_x_soln(nm)
```

Input

`nm` ([mp.net_model](#) (page 93)) – network model object

Output

`nm` ([mp.net_model](#) (page 93)) – updated network model object

Calls `convert_x_m2n()` to which is defined in a subclass of in [mp.mm_shared_pfcopf](#) (page 141) (and of [mp.math_model_opf](#) (page 134)) allowing it to be shared across math models for different tasks (PF and CPF).

`mp.math_model_pf`

`class mp.math_model_pf`

Bases: [mp.math_model](#) (page 124)

[mp.math_model_pf](#) (page 129) - Abstract base class for power flow (PF) **math model** objects.

Implements setting up of solver options from MATPOWER options struct.

Method Summary

`task_tag()`

`task_name()`

`add_costs(nm, dm, mpopt)`

`add_system_vars(nm, dm, mpopt)`

`solve_opts(nm, dm, mpopt)`

`mp.math_model_pf_ac`

class `mp.math_model_pf_ac`

Bases: `mp.math_model_pf` (page 129)

`mp.math_model_pf_ac` (page 130) - Power flow (PF) **math model** for AC formulations.

Provides AC-specific and PF-specific subclasses for elements.

Constructor Summary

`math_model_pf_ac()`

`mp.math_model_pf_acci`

class `mp.math_model_pf_acci`

Bases: `mp.math_model_pf_ac` (page 130), `mp.mm_shared_pfcpf_acci` (page 143)

`mp.math_model_pf_acci` (page 130) - Power flow (PF) **math model** for AC-cartesian-current formulation.

Implements formulation-specific node balance constraints and inherits from formulation-specific class for shared PF/CPF code.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`mp.math_model_pf_accs`

class `mp.math_model_pf_accs`

Bases: `mp.math_model_pf_ac` (page 130), `mp.mm_shared_pfcpf_accs` (page 143)

`mp.math_model_pf_accs` (page 130) - Power flow (PF) **math model** for AC-cartesian-power formulation.

Implements formulation-specific node balance constraints and inherits from formulation-specific class for shared PF/CPF code.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

mp.math_model_pf_acpi

class `mp.math_model_pf_acpi`

Bases: `mp.math_model_pf_ac` (page 130), `mp.mm_shared_pfcpf_acpi` (page 144)

`mp.math_model_pf_acpi` (page 131) - Power flow (PF) **math model** for AC-polar-current formulation.

Implements formulation-specific node balance constraints and inherits from formulation-specific class for shared PF/CPF code.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

mp.math_model_pf_acps

class `mp.math_model_pf_acps`

Bases: `mp.math_model_pf_ac` (page 130), `mp.mm_shared_pfcpf_acps` (page 144)

`mp.math_model_pf_acps` (page 131) - Power flow (PF) **math model** for AC-polar-power formulation.

Implements formulation-specific node balance constraints and inherits from formulation-specific class for shared PF/CPF code.

Also includes implementations of methods specific to fast-decoupled power flow.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`gs_x_update(x, f, nm, dm, mpopt)`

`zg_x_update(x, f, nm, dm, mpopt)`

`fd_jac_approx(nm, dm, mpopt)`

`fdpf_B_matrix_models(dm, alg)`

`mp.math_model_pf_dc`

class `mp.math_model_pf_dc`

Bases: [mp.math_model_pf](#) (page 129), [mp.mm_shared_pfcpf_dc](#) (page 144)

[mp.math_model_pf_dc](#) (page 132) - Power flow (PF) **math model** for DC formulation.

Provides formulation-specific and PF-specific subclasses for elements and implements formulation-specific node balance constraints.

Overrides the default [solve_opts\(\)](#) (page 132) method.

Constructor Summary

math_model_pf_dc()

Method Summary

form_tag()

form_name()

add_node_balance_constraints(*nm, dm, mpopt*)

solve_opts(*nm, dm, mpopt*)

`mp.math_model_cpf_acc`

class `mp.math_model_cpf_acc`

Bases: `mp.math_model_cpf`

[mp.math_model_cpf_acc](#) (page 132) - Abstract base class for AC cartesian CPF **math model** objects.

Provides formulation-specific and CPF-specific subclasses for elements.

Constructor Summary

math_model_cpf_acc()

Constructor, assign default network model element classes.

```
mm = math_model_cpf_acc()
```

`mp.math_model_cpf_acci`

class `mp.math_model_cpf_acci`

Bases: [mp.math_model_cpf_acc](#) (page 132), [mp.mm_shared_pfcpf_acci](#) (page 143)

[mp.math_model_cpf_acci](#) (page 132) - CPF **math model** for AC-cartesian-current formulation.

Implements formulation-specific and CPF-specific node balance constraint.

Method Summary

```

form_tag()

form_name()

add_node_balance_constraints(nm, dm, mpopt)

```

mp.math_model_cpf_accs

class mp.math_model_cpf_accs

Bases: [mp.math_model_cpf_acc](#) (page 132), [mp.mm_shared_pfcpf_accs](#) (page 143)

[mp.math_model_cpf_accs](#) (page 133) - CPF **math model** for AC-cartesian-power formulation.

Implements formulation-specific and CPF-specific node balance constraint.

Method Summary

```

form_tag()

form_name()

add_node_balance_constraints(nm, dm, mpopt)

```

mp.math_model_cpf_acp

class mp.math_model_cpf_acp

Bases: mp.math_model_cpf

[mp.math_model_cpf_acp](#) (page 133) - Abstract base class for AC polar CPF **math model** objects.

Provides formulation-specific and CPF-specific subclasses for elements and implementations of event and callback functions for handling voltage limits.

Constructor Summary

```

math_model_cpf_acp()
    Constructor, assign default network model element classes.

```

```
mm = math_model_cpf_acp()
```

Method Summary

```

event_vlim(cx, opt, nm, dm, mpopt)

callback_vlim(k, nx, cx, px, s, opt, nm, dm, mpopt)

```


`mp.math_model_cpf_acpi`

class `mp.math_model_cpf_acpi`

Bases: [`mp.math_model_cpf_acp`](#) (page 133), [`mp.mm_shared_pfcpf_acpi`](#) (page 144)

[`mp.math_model_cpf_acpi`](#) (page 134) - CPF **math model** for AC-polar-current formulation.

Implements formulation-specific and CPF-specific node balance constraint.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`mp.math_model_cpf_acps`

class `mp.math_model_cpf_acps`

Bases: [`mp.math_model_cpf_acp`](#) (page 133), [`mp.mm_shared_pfcpf_acps`](#) (page 144)

[`mp.math_model_cpf_acps`](#) (page 134) - CPF **math model** for AC-polar-power formulation.

Implements formulation-specific and CPF-specific node balance constraint.

Provides methods for warm-starting solver with updated data.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`expand_z_warmstart(nm, ad, varargin)`

`solve_opts_warmstart(opt, ws, nm)`

`mp.math_model_opf`

class `mp.math_model_opf`

Bases: [`mp.math_model`](#) (page 124)

[`mp.math_model_opf`](#) (page 134) - Abstract base class for optimal power flow (OPF) **math model** objects.

Provide implementations for adding system variables to the mathematical model and creating an interior starting point.

Method Summary

`task_tag()`

```

task_name()

build_aux_data(nm, dm, mpopt)

add_system_vars(nm, dm, mpopt)

interior_x0(nm, nm, dm, x0)

interior_va(nm, dm)

```

mp.math_model_opf_ac

class mp.math_model_opf_ac

Bases: [mp.math_model_opf](#) (page 134)

[mp.math_model_opf_ac](#) (page 135) - Abstract base class for AC OPF **math model** objects.

Provide implementation of nodal current and power balance functions and their derivatives, and setup of solver options.

Method Summary

```

nodal_current_balance_fcn(x, nm)

nodal_power_balance_fcn(x, nm)

nodal_current_balance_hess(x, lam, nm)

nodal_power_balance_hess(x, lam, nm)

solve_opts(nm, dm, mpopt)

```

mp.math_model_opf_acc

class mp.math_model_opf_acc

Bases: [mp.math_model_opf_ac](#) (page 135)

[mp.math_model_opf_acc](#) (page 135) - Abstract base class for AC cartesian OPF **math model** objects.

Provides formulation-specific and OPF-specific subclasses for elements.

Implements [convert_x_m2n\(\)](#) (page 135) to convert from math model state to network model state.

Constructor Summary

```

math_model_opf_acc()

```

Method Summary

```

convert_x_m2n(mmx, nm)

interior_va(nm, dm)

```

`mp.math_model_opf_acci`

class `mp.math_model_opf_acci`

Bases: [`mp.math_model_opf_acc`](#) (page 135)

[`mp.math_model_opf_acci`](#) (page 136) - OPF **math model** for AC-cartesian-current formulation.

Implements formulation-specific and OPF-specific node balance constraint and node balance price methods.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`node_power_balance_prices(nm)`

`mp.math_model_opf_acci_legacy`

class `mp.math_model_opf_acci_legacy`

Bases: [`mp.math_model_opf_acci`](#) (page 136), [`mp.mm_shared_opf_legacy`](#) (page 145)

[`mp.math_model_opf_acci_legacy`](#) (page 136) - OPF **math model** for AC-cartesian-current formulation w/legacy extensions.

Provides formulation-specific methods for handling legacy user customization of OPF problem.

Constructor Summary

`math_model_opf_acci_legacy()`

Method Summary

`add_named_set(varargin)`

`def_set_types()`

`init_set_types()`

`build(nm, dm, mpopt)`

`add_vars(nm, dm, mpopt)`

`add_system_costs(nm, dm, mpopt)`

`add_system_constraints(nm, dm, mpopt)`

`legacy_user_var_names()`

`mp.math_model_opf_accs`

class `mp.math_model_opf_accs`

Bases: [`mp.math_model_opf_acc`](#) (page 135)

[`mp.math_model_opf_accs`](#) (page 137) - OPF **math model** for AC-cartesian-power formulation.

Implements formulation-specific and OPF-specific node balance constraint and node balance price methods.

Method Summary

`form_tag()`

`form_name()`

`add_node_balance_constraints(nm, dm, mpopt)`

`node_power_balance_prices(nm)`

`mp.math_model_opf_accs_legacy`

class `mp.math_model_opf_accs_legacy`

Bases: [`mp.math_model_opf_accs`](#) (page 137), [`mp.mm_shared_opf_legacy`](#) (page 145)

[`mp.math_model_opf_accs_legacy`](#) (page 137) - OPF **math model** for AC-cartesian-power formulation w/legacy extensions.

Provides formulation-specific methods for handling legacy user customization of OPF problem.

Constructor Summary

`math_model_opf_accs_legacy()`

Method Summary

`add_named_set(varargin)`

`def_set_types()`

`init_set_types()`

`build(nm, dm, mpopt)`

`add_vars(nm, dm, mpopt)`

`add_system_costs(nm, dm, mpopt)`

`add_system_constraints(nm, dm, mpopt)`

`legacy_user_var_names()`

mp.math_model_opf_acp

class mp.math_model_opf_acp

Bases: [mp.math_model_opf_ac](#) (page 135)

[mp.math_model_opf_acp](#) (page 138) - Abstract base class for AC polar OPF **math model** objects.

Provides formulation-specific and OPF-specific subclasses for elements.

Implements [convert_x_m2n\(\)](#) (page 138) to convert from math model state to network model state.

Constructor Summary

math_model_opf_acp()

Method Summary

convert_x_m2n(*mmx*, *nm*)

mp.math_model_opf_acpi

class mp.math_model_opf_acpi

Bases: [mp.math_model_opf_acp](#) (page 138)

[mp.math_model_opf_acpi](#) (page 138) - OPF **math model** for AC-polar-current formulation.

Implements formulation-specific and OPF-specific node balance constraint and node balance price methods.

Method Summary

form_tag()

form_name()

add_node_balance_constraints(*nm*, *dm*, *mpopt*)

node_power_balance_prices(*nm*)

mp.math_model_opf_acpi_legacy

class mp.math_model_opf_acpi_legacy

Bases: [mp.math_model_opf_acpi](#) (page 138), [mp.mm_shared_opf_legacy](#) (page 145)

[mp.math_model_opf_acpi_legacy](#) (page 138) - OPF **math model** for AC-polar-current formulation w/legacy extensions.

Provides formulation-specific methods for handling legacy user customization of OPF problem.

Constructor Summary

math_model_opf_acpi_legacy()

Method Summary

```

add_named_set(varargin)

def_set_types()

init_set_types()

build(nm, dm, mpopt)

add_vars(nm, dm, mpopt)

add_system_costs(nm, dm, mpopt)

add_system_constraints(nm, dm, mpopt)

legacy_user_var_names()

```

mp.math_model_opf_acps

class mp.math_model_opf_acps

Bases: [mp.math_model_opf_acp](#) (page 138)

[mp.math_model_opf_acps](#) (page 139) - OPF **math model** for AC-polar-power formulation.

Implements formulation-specific and OPF-specific node balance constraint and node balance price methods.

Method Summary

```

form_tag()

form_name()

add_node_balance_constraints(nm, dm, mpopt)

node_power_balance_prices(nm)

```

mp.math_model_opf_acps_legacy

class mp.math_model_opf_acps_legacy

Bases: [mp.math_model_opf_acps](#) (page 139), [mp.mm_shared_opf_legacy](#) (page 145)

[mp.math_model_opf_acps_legacy](#) (page 139) - OPF **math model** for AC-polar-power formulation w/legacy extensions.

Provides formulation-specific methods for handling legacy user customization of OPF problem.

Constructor Summary

```

math_model_opf_acps_legacy()

```

Method Summary

```

add_named_set(varargin)

```

```
def_set_types()
init_set_types()
build(nm, dm, mpopt)
add_vars(nm, dm, mpopt)
add_system_costs(nm, dm, mpopt)
add_system_constraints(nm, dm, mpopt)
legacy_user_var_names()
```

mp.math_model_opf_dc

class `mp.math_model_opf_dc`

Bases: [mp.math_model_opf](#) (page 134)

[mp.math_model_opf_dc](#) (page 140) - Optimal Power flow (OPF) **math model** for DC formulation.

Provides formulation-specific and OPF-specific subclasses for elements.

Provides implementation of nodal balance constraint method and setup of solver options.

Implements [convert_x_m2n\(\)](#) (page 140) to convert from math model state to network model state.

Constructor Summary

```
math_model_opf_dc()
```

Method Summary

```
form_tag()
```

```
form_name()
```

```
convert_x_m2n(mmx, nm)
```

```
add_node_balance_constraints(nm, dm, mpopt)
```

```
solve_opts(nm, dm, mpopt)
```

mp.math_model_opf_dc_legacy

class `mp.math_model_opf_dc_legacy`

Bases: [mp.math_model_opf_dc](#) (page 140), [mp.mm_shared_opf_legacy](#) (page 145)

[mp.math_model_opf_dc](#) (page 140) - OPF **math model** for DC formulation w/legacy extensions.

Provides formulation-specific methods for handling legacy user customization of OPF problem.

Constructor Summary

`math_model_opf_dc_legacy(mpc)`

Method Summary

`add_named_set(varargin)`

`def_set_types()`

`init_set_types()`

`build(nm, dm, mpopt)`

`add_vars(nm, dm, mpopt)`

`add_system_costs(nm, dm, mpopt)`

`add_system_constraints(nm, dm, mpopt)`

`legacy_user_var_names()`

3.5.2 Container Mixins

`mp.mm_shared_pfcpf`

`class mp.mm_shared_pfcpf`

Bases: `handle`

[`mp.mm_shared_pfcpf`](#) (page 141) - Mixin class for PF/CPF **math model** objects.

An abstract mixin class inherited by all power flow (PF) and continuation power flow (CPF) **math model** objects.

Method Summary

`build_aux_data(nm, dm, mpopt)`

`mp.mm_shared_pfcpf_ac`

`class mp.mm_shared_pfcpf_ac`

Bases: [`mp.mm_shared_pfcpf`](#) (page 141)

[`mp.mm_shared_pfcpf_ac`](#) (page 141) - Mixin class for AC PF/CPF **math model** objects.

An abstract mixin class inherited by all AC power flow (PF) and continuation power flow (CPF) **math model** objects.

Method Summary

`add_system_varset_pf(nm, vvar, typ)`

update_z(*nm*, *v_*, *z_*, *ad*, *Sinj*, *idx*)

[update_z\(\)](#) (page 141) - Update/allocate active/reactive injections at slack/PV nodes.

Update/allocate slack know active power injections and slack/PV node reactive power injections.

mp.mm_shared_pfcpf_ac_i

class `mp.mm_shared_pfcpf_ac_i`

Bases: `handle`

[mp.mm_shared_pfcpf_ac_i](#) (page 142) - Mixin class for AC-current PF/CPF **math model** objects.

An abstract mixin class inherited by all AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a current balance formulation.

Code shared between AC cartesian and polar formulations with current balance belongs in this class.

Method Summary

build_aux_data_i(*nm*, *ad*)

mp.mm_shared_pfcpf_acc

class `mp.mm_shared_pfcpf_acc`

Bases: [mp.mm_shared_pfcpf_ac](#) (page 141)

[mp.mm_shared_pfcpf_acc](#) (page 142) - Mixin class for AC cartesian PF/CPF **math model** objects.

An abstract mixin class inherited by all AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a cartesian voltage formulation.

Method Summary

convert_x_m2n(*mmx*, *nm*, *only_v*)

[convert_x_m2n\(\)](#) (page 142) - Convert math model state to network model state.

```
x = mm.pf_convert(mmx, nm)
[v, z] = mm.pf_convert(mmx, nm)
[v, z, x] = mm.pf_convert(mmx, nm,)
... = mm.pf_convert(mmx, nm, only_v)
```

mp.mm_shared_pfcpf_acci

class mp.mm_shared_pfcpf_acci

Bases: [mp.mm_shared_pfcpf_acc](#) (page 142), [mp.mm_shared_pfcpf_ac_i](#) (page 142)

[mp.mm_shared_pfcpf_acci](#) (page 143) - Mixin class for AC-cartesian-current PF/CPF **math model** objects.

An abstract mixin class inherited by AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a cartesian voltage and current balance formulation.

Method Summary

```

    build_aux_data(nm, dm, mpopt)
    add_system_vars_pf(nm, dm, mpopt)
    node_balance_equations(x, nm)

```

mp.mm_shared_pfcpf_accs

class mp.mm_shared_pfcpf_accs

Bases: [mp.mm_shared_pfcpf_acc](#) (page 142)

[mp.mm_shared_pfcpf_accs](#) (page 143) - Mixin class for AC-cartesian-power PF/CPF **math model** objects.

An abstract mixin class inherited by AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a cartesian voltage and power balance formulation.

Method Summary

```

    add_system_vars_pf(nm, dm, mpopt)
    node_balance_equations(x, nm)

```

mp.mm_shared_pfcpf_acp

class mp.mm_shared_pfcpf_acp

Bases: [mp.mm_shared_pfcpf_ac](#) (page 141)

[mp.mm_shared_pfcpf_acp](#) (page 143) - Mixin class for AC polar PF/CPF **math model** objects.

An abstract mixin class inherited by all AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a polar voltage formulation.

Method Summary

```

    convert_x_m2n(mmx, nm, only_v)
    convert\_x\_m2n\(\) (page 143) - Convert math model state to network model state.

```

```

x = mm.pf_convert(mmx, nm)
[v, z] = mm.pf_convert(mmx, nm)
[v, z, x] = mm.pf_convert(mmx, nm)
... = mm.pf_convert(mmx, nm, only_v)

```

`mp.mm_shared_pfcpf_acpi`

class `mp.mm_shared_pfcpf_acpi`

Bases: `mp.mm_shared_pfcpf_acp` (page 143), `mp.mm_shared_pfcpf_ac_i` (page 142)

`mp.mm_shared_pfcpf_acpi` (page 144) - Mixin class for AC-polar-current PF/CPF **math model** objects.

An abstract mixin class inherited by AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a polar voltage and current balance formulation.

Method Summary

`build_aux_data(nm, dm, mpopt)`

`add_system_vars_pf(nm, dm, mpopt)`

`node_balance_equations(x, nm)`

`mp.mm_shared_pfcpf_acps`

class `mp.mm_shared_pfcpf_acps`

Bases: `mp.mm_shared_pfcpf_acp` (page 143)

`mp.mm_shared_pfcpf_acps` (page 144) - Mixin class for AC-polar-power PF/CPF **math model** objects.

An abstract mixin class inherited by AC power flow (PF) and continuation power flow (CPF) **math model** objects that use a polar voltage and power balance formulation.

Method Summary

`build_aux_data(nm, dm, mpopt)`

`add_system_vars_pf(nm, dm, mpopt)`

`node_balance_equations(x, nm, fdpf)`

`mp.mm_shared_pfcpf_dc`

class `mp.mm_shared_pfcpf_dc`

Bases: `mp.mm_shared_pfcpf` (page 141)

`mp.mm_shared_pfcpf_dc` (page 144) - Mixin class for DC power flow (PF) **math model** objects.

An abstract mixin class inherited by DC power flow (PF) **math model** objects.

Method Summary

`build_aux_data(nm, dm, mpopt)`

`add_system_vars_pf(nm, dm, mpopt)`

`convert_x_m2n(mmx, nm, only_v)`

[`convert_x_m2n\(\)`](#) (page 145) - Convert math model state to network model state.

```
x = mm.pf_convert(mmx, nm)
[v, z] = mm.pf_convert(mmx, nm)
[v, z, x] = mm.pf_convert(mmx, nm)
... = mm.pf_convert(mmx, nm, only_v)
```

`update_z(nm, v, z, ad)`

[`update_z\(\)`](#) (page 145) - Update/allocate slack node active power injections.

mp.mm_shared_opf_legacy

`class mp.mm_shared_opf_legacy`

Bases: `handle`

[`mp.mm_shared_opf_legacy`](#) (page 145) - Mixin class for legacy optimal power flow (OPF) **math model** objects.

An abstract mixin class inherited by optimal power flow (OPF) **math model** objects that need to handle legacy user customization mechanisms.

Method Summary

`mod_set_types_legacy()`

`init_set_types_legacy()`

`get_mpc(om)`

`build_legacy(nm, dm, mpopt)`

`add_legacy_user_vars(nm, dm, mpopt)`

`add_legacy_user_costs(nm, dm, dc)`

`add_legacy_user_constraints(nm, dm, mpopt)`

`add_legacy_user_constraints_ac(nm, dm, mpopt)`

`add_legacy_cost(om, name, idx, varargin)`

[`add_legacy_cost\(\)`](#) (page 145) - Add a set of user costs to the model

```
mm.add_legacy_cost(name, cp)
mm.add_legacy_cost(name, idx, varsets)
mm.add_legacy_cost(name, idx_list, cp)
mm.add_legacy_cost(name, idx_list, cp, varsets)
```

`eval_legacy_cost(om, x, name, idx)`

[`eval_legacy_cost\(\)`](#) (page 145) - Evaluate individual or full set of legacy user costs.

```
f = mm.eval_legacy_cost(x ...)
[f, df] = mm.eval_legacy_cost(x ...)
[f, df, d2f] = mm.eval_legacy_cost(x ...)
[f, df, d2f] = mm.eval_legacy_cost(x, name)
[f, df, d2f] = mm.eval_legacy_cost(x, name, idx_list)
```

params_legacy_cost(*om, name, idx*)

[params_legacy_cost\(\)](#) (page 146) - Return cost parameters for legacy user-defined costs.

```
cp = mm.params_legacy_cost()
cp = mm.params_legacy_cost(name)
cp = mm.params_legacy_cost(name, idx)
[cp, vs] = mm.params_legacy_cost(...)
[cp, vs, i1, iN] = mm.params_legacy_cost(...)
```

3.5.3 Elements

mp.mm_element

class mp.mm_element

Bases: `handle`

[mp.mm_element](#) (page 146) - Abstract base class for MATPOWER **mathematical model element** objects.

A math model element object typically does not contain any data, but only the methods that are used to build the math model and update the corresponding data model element once the math model has been solved.

All math model element classes inherit from [mp.mm_element](#) (page 146). Each element type typically implements its own subclasses, which are further subclassed where necessary per task and formulation, as with the container class.

By convention, math model element variables are named `mme` and math model element class names begin with `mp.mme`.

mp.mm_element Methods:

- [name\(\)](#) (page 147) - get name of element type, e.g. 'bus', 'gen'
- [data_model_element\(\)](#) (page 147) - get corresponding data model element
- [network_model_element\(\)](#) (page 147) - get corresponding network model element
- [add_vars\(\)](#) (page 147) - add math model variables for this element
- [add_constraints\(\)](#) (page 147) - add math model constraints for this element
- [add_costs\(\)](#) (page 148) - add math model costs for this element
- [data_model_update\(\)](#) (page 148) - update the corresponding data model element
- [data_model_update_off\(\)](#) (page 148) - update offline elements in corresponding data model element
- [data_model_update_on\(\)](#) (page 148) - update online elements in corresponding data model element

See the `sec_mm_element` section in the *MATPOWER Developer's Manual* for more information.

See also [mp.math_model](#) (page 124).

Method Summary

name()

Get name of element type, e.g. 'bus', 'gen'.

```
name = mme.name()
```

Output

name (*char array*) – name of element type, must be a valid struct field name

Implementation provided by an element type specific subclass.

data_model_element(dm, name)

Get corresponding data model element.

```
dme = mme.data_model_element(dm)
dme = mme.data_model_element(dm, name)
```

Inputs

- **dm** ([mp.data_model](#) (page 30)) – data model object
- **name** (*char array*) – (optional) name of element type (default is name of this object)

Output

dme ([mp.dm_element](#) (page 38)) – data model element object

network_model_element(nm, name)

Get corresponding network model element.

```
nme = mme.network_model_element(nm)
nme = mme.network_model_element(nm, name)
```

Inputs

- **nm** ([mp.net_model](#) (page 93)) – network model object
- **name** (*char array*) – (optional) name of element type (default is name of this object)

Output

nme ([mp.nm_element](#) (page 110)) – network model element object

add_vars(mm, nm, dm, mpopt)

Add math model variables for this element.

```
mme.add_vars(mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Implementation provided by a subclass.

add_constraints(mm, nm, dm, mpopt)

Add math model constraints for this element.

```
mme.add_constraints(obj, mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Implementation provided by a subclass.

add_costs(*mm, nm, dm, mpopt*)

Add math model costs for this element.

```
mme.add_costs(obj, mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Implementation provided by a subclass.

data_model_update(*mm, nm, dm, mpopt*)

Update the corresponding data model element.

```
mme.data_model_update(mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Call [data_model_update_off\(\)](#) (page 148) then [data_model_update_on\(\)](#) (page 148) to update the data model for this element based on the math model solution.

See also [data_model_update_off\(\)](#) (page 148), [data_model_update_on\(\)](#) (page 148).

data_model_update_off(*mm, nm, dm, mpopt*)

Update offline elements in the corresponding data model element.

```
mme.data_model_update_off(mm, nm, dm, mpopt)
```

Inputs

- **mm** ([mp.math_model](#) (page 124)) – mathematical model object
- **nm** ([mp.net_model](#) (page 93)) – network model object
- **dm** ([mp.data_model](#) (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Set export variables for offline elements based on specs returned by [mp.dm_element.export_vars_offline_val\(\)](#) (page 42).

See also [data_model_update\(\)](#) (page 148), [data_model_update_on\(\)](#) (page 148).

data_model_update_on(*mm, nm, dm, mpopt*)

Update online elements in the corresponding data model element.

```
mme.data_model_update_on(mm, nm, dm, mpopt)
```

Inputs

- **mm** (*mp.math_model* (page 124)) – mathematical model object
- **nm** (*mp.net_model* (page 93)) – network model object
- **dm** (*mp.data_model* (page 30)) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Extract the math model solution relevant to this particular element and update the corresponding data model element for online elements accordingly.

Implementation provided by a subclass.

See also *data_model_update()* (page 148), *data_model_update_off()* (page 148).

mp.mme_branch

class `mp.mme_branch`

Bases: *mp.mm_element* (page 146)

mp.mme_branch (page 149) - Math model element abstract base class for branch.

Abstract math model element base class for branch elements, including transmission lines and transformers.

Method Summary

name()

mp.mme_branch_pf_ac

class `mp.mme_branch_pf_ac`

Bases: *mp.mme_branch* (page 149)

mp.mme_branch_pf_ac (page 149) - Math model element for branch for AC power flow.

Math model element class for branch elements, including transmission lines and transformers, for AC power flow problems.

Implements updating the output data in the corresponding data model element for in-service branches from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_branch_pf_dc

class mp.mme_branch_pf_dc

Bases: [mp.mme_branch](#) (page 149)

[mp.mme_branch_pf_dc](#) (page 150) - Math model element for branch for DC power flow.

Math model element class for branch elements, including transmission lines and transformers, for DC power flow problems.

Implements updating the output data in the corresponding data model element for in-service branches from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_branch_opf

class mp.mme_branch_opf

Bases: [mp.mme_branch](#) (page 149)

[mp.mme_branch_opf](#) (page 150) - Math model element abstract base class for branch for OPF.

Math model element abstract base class for branch elements, including transmission lines and transformers, for OPF problems.

Implements methods to prepare data required for angle difference limit constraints and to extract shadow prices for these constraints from the math model solution.

Method Summary

ang_diff_params(*dm, ignore*)

ang_diff_prices(*mm, nme*)

mp.mme_branch_opf_ac

class mp.mme_branch_opf_ac

Bases: [mp.mme_branch_opf](#) (page 150)

[mp.mme_branch_opf_ac](#) (page 150) - Math model element abstract base class for branch for AC OPF.

Math model element abstract base class for branch elements, including transmission lines and transformers, for AC OPF problems.

Implements methods for adding of branch flow constraints and for updating the output data in the corresponding data model element for in-service branches from the math model solution.

Method Summary

add_constraints(*mm, nm, dm, mpopt*)

`data_model_update_on(mm, nm, dm, mpopt)`

mp.mme_branch_opf_acc

class `mp.mme_branch_opf_acc`

Bases: `mp.mme_branch_opf_ac` (page 150)

`mp.mme_branch_opf_acc` (page 151) - Math model element for branch for AC cartesian voltage OPF.

Math model element class for branch elements, including transmission lines and transformers, for AC cartesian voltage OPF problems.

Implements method for adding branch angle difference constraints and overrides method to extract shadow prices for these constraints from the math model solution.

Method Summary

`add_constraints(mm, nm, dm, mpopt)`

`ang_diff_prices(mm, nme)`

mp.mme_branch_opf_acp

class `mp.mme_branch_opf_acp`

Bases: `mp.mme_branch_opf_ac` (page 150)

`mp.mme_branch_opf_acp` (page 151) - Math model element for branch for AC polar voltage OPF.

Math model element class for branch elements, including transmission lines and transformers, for AC polar voltage OPF problems.

Implements method for adding branch angle difference constraints.

Method Summary

`add_constraints(mm, nm, dm, mpopt)`

mp.mme_branch_opf_dc

class `mp.mme_branch_opf_dc`

Bases: `mp.mme_branch_opf` (page 150)

`mp.mme_branch_opf_dc` (page 151) - Math model element for branch for DC OPF.

Math model element class for branch elements, including transmission lines and transformers, for DC OPF problems.

Implements methods for adding of branch flow and angle difference constraints and for updating the output data in the corresponding data model element for in-service branches from the math model solution.

Method Summary**add_constraints**(*mm, nm, dm, mpopt*)**data_model_update_on**(*mm, nm, dm, mpopt*)**mp.mme_bus****class** `mp.mme_bus`Bases: `mp.mm_element` (page 146)`mp.mme_bus` (page 152) - Math model element abstract base class for bus.

Abstract math model element base class for bus elements.

Method Summary**name**()**mp.mme_bus_pf_ac****class** `mp.mme_bus_pf_ac`Bases: `mp.mme_bus` (page 152)`mp.mme_bus_pf_ac` (page 152) - Math model element for bus for AC power flow.

Math model element class for bus elements for AC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service buses from the math model solution.

Method Summary**data_model_update_on**(*mm, nm, dm, mpopt*)**mp.mme_bus_pf_dc****class** `mp.mme_bus_pf_dc`Bases: `mp.mme_bus` (page 152)`mp.mme_bus_pf_dc` (page 152) - Math model element for bus for DC power flow.

Math model element class for bus elements for DC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service buses from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpop)`

mp.mme_bus_opf_ac

class `mp.mme_bus_opf_ac`

Bases: `mp.mme_bus` (page 152)

`mp.mme_bus_opf_ac` (page 153) - Math model element abstract base class for bus for AC OPF.

Abstract math model element class for bus elements for AC OPF problems.

Implements method for forming an interior initial point for voltage magnitudes.

Method Summary

`interior_vm(mm, nm, dm)`

return vm equal to avg of clipped limits

mp.mme_bus_opf_acc

class `mp.mme_bus_opf_acc`

Bases: `mp.mme_bus_opf_ac` (page 153)

`mp.mme_bus_opf_acc` (page 153) - Math model element for bus for AC cartesian voltage OPF.

Math model element class for bus elements for AC cartesian voltage OPF problems.

Implements methods for adding constraints for reference voltage angle, fixed voltage magnitudes and voltage magnitude limits, for forming an interior initial point and for updating the output data in the corresponding data model element for in-service buses from the math model solution.

Method Summary

`add_constraints(mm, nm, dm, mpop)`

`interior_x0(mm, nm, dm, x0)`

`data_model_update_on(mm, nm, dm, mpop)`

mp.mme_bus_opf_acp

class `mp.mme_bus_opf_acp`

Bases: `mp.mme_bus_opf_ac` (page 153)

`mp.mme_bus_opf_acp` (page 153) - Math model element for bus for AC polar voltage OPF.

Math model element class for bus elements for AC polar voltage OPF problems.

Implements methods for forming an interior initial point and for updating the output data in the corresponding data model element for in-service buses from the math model solution.

Method Summary**interior_x0**(*mm, nm, dm, x0*)**data_model_update_on**(*mm, nm, dm, mpopt*)**mp.mme_bus_opf_dc****class** `mp.mme_bus_opf_dc`Bases: `mp.mme_bus` (page 152)`mp.mme_bus_opf_dc` (page 154) - Math model element for bus for DC OPF.

Math model element class for bus elements for DC OPF problems.

Implements methods for forming an interior initial point and for updating the output data in the corresponding data model element for in-service buses from the math model solution.

Method Summary**interior_x0**(*mm, nm, dm, x0*)**data_model_update_on**(*mm, nm, dm, mpopt*)**mp.mme_gen****class** `mp.mme_gen`Bases: `mp.mm_element` (page 146)`mp.mme_gen` (page 154) - Math model element abstract base class for generator.

Abstract math model element base class for generator elements.

Method Summary**name**()**mp.mme_gen_pf_ac****class** `mp.mme_gen_pf_ac`Bases: `mp.mme_gen` (page 154)`mp.mme_gen_pf_ac` (page 154) - Math model element for generator for AC power flow.

Math model element class for generator elements for AC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service generators from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

`mp.mme_gen_pf_dc`

class `mp.mme_gen_pf_dc`

Bases: [mp.mme_gen](#) (page 154)

[mp.mme_gen_pf_dc](#) (page 155) - Math model element for generator for DC power flow.

Math model element class for generator elements for DC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service generators from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

`mp.mme_gen_opf`

class `mp.mme_gen_opf`

Bases: [mp.mme_gen](#) (page 154)

[mp.mme_gen_opf](#) (page 155) - Math model element abstract base class for generator for OPF.

Math model element abstract base class for generator elements for OPF problems.

Implements methods to add costs, including piecewise linear cost variables, and to form an interior initial point for cost variables.

Property Summary

cost

struct for [cost](#) (page 155) parameters with fields:

- `poly_p` - polynomial costs for active power, struct returned by [mp.cost_table.poly_params\(\)](#) (page 165), with fields:
 - `have_quad_cost`
 - `i0, i1, i2, i3`
 - `k, c, Q`
- `poly_q` - polynomial costs for reactive power (*same struct as poly_p*)
- `pwl` - piecewise linear costs for active & reactive struct returned by [mp.cost_table.pwl_params\(\)](#) (page 166), with fields:
 - `n, i, A, b`

Method Summary

`add_vars(mm, nm, dm, mpopt)`

`add_costs(mm, nm, dm, mpopt)`

`interior_x0(mm, nm, dm, x0)`

mp.mme_gen_opf_ac

class mp.mme_gen_opf_ac

Bases: [mp.mme_gen_opf](#) (page 155)

[mp.mme_gen_opf_ac](#) (page 156) - Math model element for generator for AC OPF.

Math model element class for generator elements for AC OPF problems.

Implements methods for buliding and adding PQ capability constraints, dispatchable load power factor constraints, polynomial costs, and for updating the output data in the corresponding data model element for in-service generators from the math model solution.

Method Summary

add_constraints(*mm, nm, dm, mpopt*)

add_costs(*mm, nm, dm, mpopt*)

pq_capability_constraint(*dme, base_mva*)
from legacy [makeApq\(\)](#) (page 291)

has_pq_cap(*gen, upper_lower*)
from legacy [hasPQcap\(\)](#) (page 341)

disp_load_constant_pf_constraint(*dm*)
from legacy [makeAvl\(\)](#) (page 292)

build_cost_params(*dm*)

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_gen_opf_dc

class mp.mme_gen_opf_dc

Bases: [mp.mme_gen_opf](#) (page 155)

[mp.mme_gen_opf_dc](#) (page 156) - Math model element for generator for DC OPF.

Math model element class for generator elements for DC OPF problems.

Implements methods for buliding cost parameters, adding piecewise linear cost constraints, and for updating the output data in the corresponding data model element for in-service generators from the math model solution.

Method Summary

add_constraints(*mm, nm, dm, mpopt*)

build_cost_params(*dm*)

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_load**class mp.mme_load**

Bases: [mp.mm_element](#) (page 146)

[mp.mme_load](#) (page 157) - Math model element abstract base class for load.

Abstract math model element base class for load elements.

Method Summary

name()

mp.mme_load_pf_ac**class mp.mme_load_pf_ac**

Bases: [mp.mme_load](#) (page 157)

[mp.mme_load_pf_ac](#) (page 157) - Math model element for load for AC power flow.

Math model element class for load elements for AC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service loads from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpop*)

mp.mme_load_pf_dc**class mp.mme_load_pf_dc**

Bases: [mp.mme_load](#) (page 157)

[mp.mme_load_pf_dc](#) (page 157) - Math model element for load for DC power flow.

Math model element class for load elements for DC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service loads from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpop*)

mp.mme_load_cpf

class mp.mme_load_cpf

Bases: [mp.mme_load_pf_ac](#) (page 157)

[mp.mme_load_cpf](#) (page 158) - Math model element for load for CPF.

Math model element class for load elements for AC CPF problems.

Implements method for updating the output data in the corresponding data model element for in-service loads from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_shunt

class mp.mme_shunt

Bases: [mp.mm_element](#) (page 146)

[mp.mme_shunt](#) (page 158) - Math model element abstract base class for shunt.

Abstract math model element base class for shunt elements.

Method Summary

name()

mp.mme_shunt_pf_ac

class mp.mme_shunt_pf_ac

Bases: [mp.mme_shunt](#) (page 158)

[mp.mme_shunt_pf_ac](#) (page 158) - Math model element for shunt for AC power flow.

Math model element class for shunt elements for AC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service shunts from the math model solution.

Method Summary

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_shunt_pf_dc

class mp.mme_shunt_pf_dc

Bases: [mp.mme_shunt](#) (page 158)

[mp.mme_shunt_pf_dc](#) (page 159) - Math model element for shunt for DC power flow.

Math model element class for shunt elements for DC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service shunts from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

mp.mme_shunt_cpf

class mp.mme_shunt_cpf

Bases: [mp.mme_shunt_pf_ac](#) (page 158)

[mp.mme_shunt_cpf](#) (page 159) - Math model element for shunt for CPF.

Math model element class for shunt elements for AC CPF problems.

Implements method for updating the output data in the corresponding data model element for in-service shunts from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

3.6 Miscellaneous Classes

3.6.1 mp_table

class mp_table

[mp_table](#) (page 159) - Very basic table-compatible class for Octave or older Matlab.

```

T = mp_table(var1, var2, ...);
T = mp_table(..., 'VariableNames', {name1, name2, ...});
T = mp_table(..., 'RowNames', {name1, name2, ...});
T = mp_table(..., 'DimensionNames', {name1, name2, ...});

```

Implements a very basic table array class focused the ability to store and access named variables of different types in a way that is compatible with MATLAB's built-in table class. Other features, such as table joining, etc., are not implemented.

Important: Since the dot syntax `T.<var_name>` is used to access table variables, you must use a functional syntax `<method>(T, ...)`, as opposed to the object-oriented `T.<method>(...)`, to call `mp_table` methods.

mp_table Methods:

- `mp_table()` (page 160) - construct object
- `istable()` (page 160) - true for `mp_table` (page 159) objects
- `size()` (page 160) - dimensions of table
- `isempty()` (page 160) - true if table has no columns or no rows
- `end()` (page 161) - used to index last row or variable/column
- `subsref()` (page 161) - indexing a table to retrieve data
- `subsasgn()` (page 161) - indexing a table to assign data
- `horzcat()` (page 162) - concatenate tables horizontally
- `vertcat()` (page 162) - concatenate tables vertically
- `display()` (page 162) - display table contents

See also `table`.

Constructor Summary

mp_table(*varargin*)

Constructs the object.

```
T = mp_table(var1, var2, ...)
T = mp_table(..., 'VariableNames', {name1, name2, ...})
T = mp_table(..., 'RowNames', {name1, name2, ...})
T = mp_table(..., 'DimensionNames', {name1, name2, ...})
```

Method Summary

istable()

Returns true.

```
TorF = istable(T)
```

Unfortunately, this is not really useful until Octave implements a built-in `istable()` (page 160) that this can override.

size(*dim*)

Returns dimensions of table.

```
[m, n] = size(T)
m = size(T, 1)
n = size(T, 2)
```

isempty()

Returns `true` if the table has no columns or no rows.

```
TorF = isempty(T)
```

end(*k, n*)

Used to index the last row or column of the table.

```
last_var = T{:, end}
last_row = T(end, :)
```

subsref(*s*)

Called when indexing a table to retrieve data.

```
sub_T = T(i, *)
sub_T = T(i1:iN, *)
sub_T = T(:, *)
sub_T = T(*, j)
sub_T = T(*, j1:jN)
sub_T = T(*, :)
sub_T = T(*, <str>)
sub_T = T(*, <cell>)
var_<name> = T.<name>
val = T.<name>(i)
val = T.<name>(i1:iN)
val = T.<name>{i}
val = T.<name>{i1:iN}
val = T.<name>(*, :)
val = T.<name>(*, j)
var_<j> = T{:, j}
var_<str> = T{:, <str>}
val = T{i, *}
val = T{i1:iN, *}
val = T{:, *}
val = T{* , j}
val = T{* , j1:jN}
val = T{* , :}
val = T{* , <str>}
val = T{* , <cell>}
```

subsasgn(*s, b*)

Called when indexing a table to assign data.

```
T(i, *) = sub_T
T(i1:iN, *) = sub_T
T(:, *) = sub_T
T(*, j) = sub_T
T(*, j1:jN) = sub_T
T(*, :) = sub_T
T(*, <str>) = sub_T
T(*, <cell>) = sub_T
T.<name> = val
T.<name>(i) = val
T.<name>(i1:iN) = val
T.<name>{i} = val
T.<name>{i1:iN} = val
```

(continues on next page)

(continued from previous page)

```

T.<name>(*, :) = val
T.<name>(*, j) = val
T{:, j} = var_<j>
T{:, <str>} = var_<str>
T{i, *} = val
T{i1:iN, *} = val
T{:, *} = val
T{*, j} = val
T{*, j1:jN} = val
T{*, :} = val
T{*, <str>} = val
T{*, <cell>} = val

```

horzcat(varargin)

Concatenate tables horizontally.

```
T = [T1 T2]
```

vertcat(varargin)

Concatenate tables vertically.

```
T = [T1; T2]
```

display()

Display the table contents.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

By default it displays only the first and last 10 rows if there are more than 25 rows.

Does not currently display the contents of any nested tables.

static extract_named_args(args)

Extracts special named constructor arguments.

```

[var_names, row_names, dim_names, args] = extract_named_args(var1, var2, ...
↪)
[...] = extract_named_args(..., 'VariableNames', {name1, name2, ...})
[...] = extract_named_args(..., 'RowNames', {name1, name2, ...})
[...] = extract_named_args(..., 'DimensionNames', {name1, name2, ...})

```

Used to extract named arguments, 'VariableNames', 'RowNames', and 'DimensionNames', to pass to constructor.

3.6.2 mp_table_subclass

class mp_table_subclass

mp_table_subclass (page 163) - Class that acts like a table but isn't one.

Addresses two issues with inheriting from **table** classes (**table**) or *mp_table* (page 159)).

1. In MATLAB, **table** is a sealed class, so you cannot inherit from it. You can, however, use a subclass of *mp_table* (page 159), but that can result in the next issue under Octave.
2. While nesting of tables works just fine in general, when using *mp_table* (page 159) in Octave (at least up through 8.4.0), you cannot nest a subclass of *mp_table* (page 159) inside another *mp_table* (page 159) object because of this bug: <https://savannah.gnu.org/bugs/index.php?65037>.

To work around these issues, your “table subclass” can inherit from **this** class. An object of this class **isn't** a **table** or *mp_table* (page 159) object, but rather it **contains** one and attempts to act like one. That is, it delegates method calls (currently only those available in *mp_table* (page 159), listed below) to the contained table object.

The class of the contained table object is either **table** or *mp_table* (page 159) and is determined by *mp_table_class()* (page 8).

Limitations

1. The Octave bug mentioned above also affects tables that inherit from *mp_table_subclass* (page 163). That is, such tables can be nested inside tables of type **table** or *mp_table* (page 159), but not inside tables that are or inherit from *mp_table_subclass* (page 163).
 2. In MATLAB, when nesting an *mp_table_subclass* (page 163) object within another *mp_table_subclass* (page 163) object, one cannot use multi-level indexing directly. E.g. If T2 is a variable in T1 and x is a variable in T2, attempting `x = T1.T2.x` will result in an error. The indexing must be done in multiple steps `T2 = T1.T2; x = T2.x`. Note: This only applies to MATLAB, where the contained table is a **table**. It works just fine in Octave, where the contained table is an *mp_table* (page 159).
-

Important: Since the dot syntax `T.<var_name>` is used to access table variables, you must use a functional syntax `<method>(T, ...)`, as opposed to the object-oriented `T.<method>(...)`, to call methods of this class or subclasses, as with *mp_table*.

mp.mp_table_subclass Properties:

- `tab` - (*table* or *mp_table*) contained table object this class emulates

mp.cost_table Methods:

- `mp_table_subclass()` - construct object
- `get_table()` (page 164) - return the table stored in `tab`
- `set_table()` (page 164) - assign a table to `tab`
- `istable()` - true for *mp_table* (page 159) objects
- `size()` - dimensions of table
- `isempty()` - true if table has no columns or no rows
- `end()` - used to index last row or variable/column
- `subsref()` - indexing a table to retrieve data

- `subsasgn()` - indexing a table to assign data
- `horzcat()` - concatenate tables horizontally
- `vertcat()` - concatenate tables vertically
- `display()` - display table contents

See also [mp_table](#) (page 159), [mp_table_class\(\)](#) (page 8).

Method Summary

`get_table()`

```
T = get_table(obj)
```

`set_table(T)`

```
set_table(obj, T)
```

3.6.3 mp.cost_table

`class mp.cost_table`

Bases: [mp_table_subclass](#) (page 163)

[mp.cost_table](#) (page 164) - Table for (polynomial and piecewise linear) cost parameters.

```
T = cost_table(poly_n, poly_coef, pwl_n, pwl_qty, pwl_cost);
```

Important: Since the dot syntax `T.<var_name>` is used to access table variables, you must use a functional syntax `<method>(T, ...)`, as opposed to the object-oriented `T.<method>(...)`, to call standard `mp.cost_table` methods.

Standard table subscripting syntax is not available within methods of this class (references built-in `subsref()` and `subsasgn()` rather than the versions overridden by the table class). For this reason, some method implementations are delegated to static methods in [mp.cost_table_utils](#) (page 167) where that syntax is available, making the code more readable.

`mp.cost_table` Methods:

- [cost_table\(\)](#) (page 165) - construct object
- [poly_params\(\)](#) (page 165) - create struct of polynomial parameters from [mp.cost_table](#) (page 164)
- [pwl_params\(\)](#) (page 166) - create struct of piecewise linear parameters from [mp.cost_table](#) (page 164)
- [max_pwl_cost\(\)](#) (page 166) - get maximum cost component used to specify pwl costs

An [mp.cost_table](#) (page 164) has the following columns:

Name	Type	Description
<code>poly_n</code>	<i>integer</i>	n_{poly} , number of coefficients in polynomial cost curve, $f_{\text{poly}}(x) = c_0 + c_1x + \dots + c_Nx^N$, where $n_{\text{poly}} = N + 1$
<code>poly_coef</code>	<i>double</i>	matrix of coefficients c_j , of polynomial cost $f_{\text{poly}}(x)$, where c_j is found in column $j + 1$
<code>pwl_n</code>	<i>double</i>	n_{pwl} , number of data points $(x_1, f_1), (x_2, f_2), \dots, (x_N, f_N)$ defining a piecewise linear cost curve, $f_{\text{pwl}}(x)$ where $N = n_{\text{pwl}}$
<code>pwl_qty</code>	<i>double</i>	matrix of <i>quantity</i> coordinates x_j for piecewise linear cost $f_{\text{pwl}}(x)$, where x_j is found in column j
<code>pwl_cost</code>	<i>double</i>	matrix of <i>cost</i> coordinates f_j for piecewise linear cost $f_{\text{pwl}}(x)$, where f_j is found in column j

See also [mp.cost_table_utils](#) (page 167), [mp_table_subclass](#) (page 163).

Constructor Summary

cost_table(varargin)

```
T = cost_table()
T = cost_table(poly_n, poly_coef, pwl_n, pwl_qty, pwl_cost)
```

For descriptions of the inputs, see the corresponding column in the class documentation above.

Inputs

- **poly_n** (col vector of integers)
- **poly_coef** (matrix of doubles)
- **pwl_n** (col vector of integers)
- **pwl_qty** (matrix of doubles)
- **pwl_cost** (matrix of doubles)

Outputs

T ([mp.cost_table](#) (page 164)) – the cost table object

Method Summary

poly_params(idx, pu_base)

```
p = poly_params(obj, idx, pu_base)
```

Inputs

- **obj** ([mp.cost_table](#) (page 164)) – the cost table
- **idx** – (integer) : index vector of rows of interest, empty for all rows
- **pu_base** (double) – base used to scale quantities to per unit

Outputs

- p** (struct) – polynomial cost parameters, struct with fields:
- **have_quad_cost** - true if any polynomial costs have order quadratic or less
 - **i0** - row indices for constant costs
 - **i1** - row indices for linear costs
 - **i2** - row indices for quadratic costs
 - **i3** - row indices for order 3 or higher costs
 - **k** - constant term for all quadratic and lower order costs
 - **c** - linear term for all quadratic and lower order costs
 - **Q** - quadratic term for all quadratic and lower order costs

Implementation in `mp.cost_table_utils.poly_params()` (page 167).

pwl_params(*idx*, *pu_base*, *varargin*)

```
p = pwl_params(obj, idx, pu_base)
p = pwl_params(obj, idx, pu_base, ng, dc)
```

Inputs

- **obj** (`mp.cost_table` (page 164)) – the cost table
- **idx** – (integer) : index vector of rows of interest, empty for all rows
- **pu_base** (*double*) – base used to scale quantities to per unit
- **ng** (*integer*) – number of units, default is # of rows in cost
- **dc** (*boolean*) – true if DC formulation (ng variables), otherwise AC formulation (2*ng variables), default is 1

Outputs

- **p** (*struct*) – piecewise linear cost parameters, struct with fields:
 - **n** - number of piecewise linear costs
 - **i** - row indices for piecewise linear costs
 - **A** - constraint coefficient matrix for CCV formulation
 - **b** - constraint RHS vector for CCV formulation

Implementation in `mp.cost_table_utils.pwl_params()` (page 168).

max_pwl_cost()

```
maxc = max_pwl_cost(obj)
```

Input

- **obj** (`mp.cost_table` (page 164)) – the cost table

Output

- **maxc** (*double*) – maximum cost component of all breakpoints used to specify piecewise linear costs

Implementation in `mp.cost_table_utils.max_pwl_cost()` (page 168).

static poly_cost_fcn(*xx*, *x_scale*, *ccm*, *idx*)

```
f = mp.cost_table.poly_cost_fcn(xx, x_scale, ccm, idx)
[f, df] = mp.cost_table.poly_cost_fcn(...)
[f, df, d2f] = mp.cost_table.poly_cost_fcn(...)
```

Evaluates the sum of a set of polynomial cost functions $f(x) = \sum_{i \in I} f_i(x_i)$, and optionally the gradient and Hessian.

Inputs

- **xx** (*single element cell array of double*) – first element is a vector of the pre-scaled quantities x/α used to compute the costs
- **x_scale** (*double*) – scalar α used to scale the quantity value before evaluating the polynomial cost
- **ccm** (*double*) – cost coefficient matrix, element (i,j) is the coefficient of the $(j-1)$ order term for cost i
- **idx** (*integer*) – index vector of subset I of rows of **xx**{1} and **ccm** of interest

Outputs

- **f** (*double*) – value of cost function $f(x)$
- **df** (*vector of double*) – (optional) gradient of cost function
- **d2f** (*matrix of double*) – (optional) Hessian of cost function

static eval_poly_fcn(c, x)

```
f = mp.cost_table.eval_poly_fcn(c, x)
```

Evaluate a vector of polynomial functions, where ...

```
f = c(:,1) + c(:,2) .* x + c(:,3) .* x^2 + ...
```

Inputs

- **c** (*matrix of double*) – coefficient matrix, element (i,j) is the coefficient of the $(j-1)$ order term for i -th element of f
- **x** (*vector of double*) – vector of input values

Outputs

f (*vector of double*) – value of functions

static diff_poly_fcn(c)

```
c = mp.cost_table.diff_poly_fcn(c)
```

Compute the coefficient matrix for the derivatives of a set of polynomial functions from the coefficients of the functions.

Inputs

c (*matrix of double*) – coefficient matrix for the functions, element (i,j) is the coefficient of the $(j-1)$ order term of the i -th function

Outputs

c (*matrix of double*) – coefficient matrix for the derivatives of the functions, element (i,j) is the coefficient of the $(j-1)$ order term of the derivative of the i -th function

3.6.4 mp.cost_table_utils

class mp.cost_table_utils

[mp.cost_table_utils](#) (page 167) - Static methods for [mp.cost_table](#) (page 164).

Contains the implementation of some methods that would ideally belong in [mp.cost_table](#) (page 164).

Within classes that inherit from [mp_table_subclass](#) (page 163), such as [mp.cost_table](#) (page 164), any subscripting to access the elements of the table must be done through explicit calls to the table's [subsref\(\)](#) and [subsasgn\(\)](#) methods. That is, the normal table subscripting syntax will not work, so working with the table becomes extremely cumbersome.

This purpose of this class is to provide the implementation for [mp.cost_table](#) (page 164) methods that **do** allow access to that table via normal table subscripting syntax.

mp.cost_table_util Methods:

- [poly_params\(\)](#) (page 167) - create struct of polynomial parameters from [mp.cost_table](#) (page 164)
- [pwl_params\(\)](#) (page 168) - create struct of piecewise linear parameters from [mp.cost_table](#) (page 164)
- [max_pwl_cost\(\)](#) (page 168) - get maximum cost component used to specify pwl costs

See also [mp.cost_table](#) (page 164).

Method Summary

static `poly_params(cost, idx, pu_base)`

```
p = mp.cost_table_utils.poly_params(cost, idx, pu_base)
```

Implementation for `mp.cost_table.poly_params()` (page 165). See `mp.cost_table.poly_params()` (page 165) for details.

static `pwl_params(cost, idx, pu_base, ng, dc)`

```
p = mp.cost_table_utils.pwl_params(cost, idx, pu_base)
p = mp.cost_table_utils.pwl_params(cost, idx, pu_base, ng, dc)
```

Implementation for `mp.cost_table.pwl_params()` (page 166). See `mp.cost_table.pwl_params()` (page 166) for details.

static `max_pwl_cost(cost)`

```
maxc = mp.cost_table_utils.max_pwl_cost(cost)
```

Implementation for `mp.cost_table.max_pwl_cost()` (page 166). See `mp.cost_table.max_pwl_cost()` (page 166) for details.

3.6.5 mp.element_container

class `mp.element_container`

Bases: `handle`

`mp.element_container` (page 168) - Mix-in class to handle named/ordered element object array.

Implements an element container that is used for MATPOWER model and data model converter objects. Provides the properties to store the constructors for each element and the elements themselves. Also provides a method to modify an existing set of element constructors.

mp.element_container Properties:

- `element_classes` (page 168) - cell array of element constructors
- `elements` (page 168) - a `mp.mapped_array` (page 169) to hold the element objects

mp.element_container Methods:

- `modify_element_classes()` (page 168) - modify an existing set of element constructors

See also `mp.mapped_array` (page 169).

Property Summary

element_classes

Cell array of function handles of constructors for individual elements, filled by constructor of subclass.

elements

A mapped array (`mp.mapped_array` (page 169)) to hold the element objects included inside this container object.

Method Summary

modify_element_classes(*class_list*)

Modify an existing set of element constructors.

```
obj.modify_element_classes(class_list)
```

Input

class_list (*cell array*) – list of **element class modifiers**, where each modifier is one of the following:

1. a handle to a constructor to **append** to `obj.element_classes`, *or*
2. a char array B, indicating to **remove** any element E in the list for which `isa(E(), B)` is `true`, *or*
3. a 2-element cell array {A,B} where A is a handle to a constructor to **replace** any element E in the list for which `isa(E(), B)` is `true`, i.e. B is a char array

Also accepts a single element class modifier of type 1 or 2 (*A single type 3 modifier has to be enclosed in a single-element cell array to keep it from being interpreted as a list of 2 modifiers*).

Can be used to modify the list of element constructors in the `element_classes` property by appending, removing, or replacing entries. See `tab_element_class_modifiers` in the [MATPOWER Developer's Manual](#) for more information.

3.6.6 mp.mapped_array

class mp.mapped_array

Bases: handle

mp.mapped_array (page 169) - Cell array indexed by name as well as numeric index.

Currently, arrays are only 1-D.

Example usage:

```
% create a mapped array object
ma = mp.mapped_array({30, 40, 50}, {'width', 'height', 'depth'});

% treat it like a cell array
ma{3} = 60;
height = ma{2};
for i = 1:length(ma)
    disp( ma{i} );
end

% treat it like a struct
ma.width = 20;
depth = ma.depth;

% add elements
ma.add_elements({'red', '25 lbs'}, {'color', 'weight'});

% delete elements
ma.delete_elements([3 5]);
ma.delete_elements('height');
```

(continues on next page)

(continued from previous page)

```
% check for named element
ma.has_name('color');
```

mp.mapped_array Methods:

- `mapped_array()` (page 170) - constructor
- `copy()` (page 170) - create a duplicate of the mapped array object
- `length()` (page 170) - return number of elements in mapped array
- `size()` (page 170) - return dimensions of mapped array
- `add_names()` (page 171) - add or modify names of elements
- `add_elements()` (page 171) - append elements to the end of the mapped array
- `delete_elements()` (page 171) - delete elements from the mapped array
- `has_name()` (page 171) - return `true` if the name exists in the mapped array
- `name2idx()` (page 171) - return the index corresponding to a name
- `subsref()` (page 171) - called when indexing a mapped array to retrieve data
- `subsasgn()` (page 172) - called when indexing a mapped array to assign data
- `display()` (page 172) - display the mapped array structure

Constructor Summary**mapped_array**(*varargin*)

```
obj = mp.mapped_array(vals)
obj = mp.mapped_array(vals, names)
```

Inputs

- **vals** (*cell array*) – values to be stored
- **names** (*cell array of char arrays*) – names for each element in **vals**, where a valid name is any valid variable name that is not one of the methods of this class. If names are not provided, it is equivalent to a cell array, except that names can be added later.

Method Summary**copy()**

Create a duplicate of the mapped array object.

```
new_obj = obj.copy();
```

length()

Return number of elements in mapped array.

```
num_elements = obj.length();
```

size(dim)

Return dimensions of mapped array. First dimension is 1, second matches the length.

```
[m, n] = obj.size();
m = obj.size(1);
n = obj.size(2);
```

add_names(i0, names)

Add or modify names of elements.

```
obj.add_names(i0, names)
```

Inputs

- **i0** (*cell array*) – index of element corresponding to first name provided in names
- **names** (*char array or cell array of char arrays*) – the names to assign

Adds or overwrites the names for elements starting at the specified index.

add_elements(vals, names)

Append elements to the end of the mapped array.

```
obj.add_elements(vals);
obj.add_elements(vals, names);
```

Inputs

- **vals** – single value or cell array of values
- **names** (*char array or cell array of char arrays*) – (optional) corresponding names

The two arguments must be both cell arrays of the same dimension or a single value and single name.

See also [delete_elements\(\)](#) (page 171).

delete_elements(refs)

Delete elements from the mapped array.

```
obj.delete_elements(idx);
obj.delete_elements(names);
```

Inputs

- **idx** (*scalar or vector integer*) – index(indices) of element(s) to delete
- **names** (*char array or cell array of char arrays*) – name(s) of element(s) to delete

See also [add_elements\(\)](#) (page 171).

has_name(name)

Return true if the name exists in the mapped array.

```
TorF = obj.has_name(name);
```

Input

name (*char array*) – name to check

name2idx(name)

Return the numerical index in the array corresponding to a name.

```
idx = obj.name2idx(name);
```

Input

name (*char array*) – name corresponding to desired index

subsref(*s*)

Called when indexing a table to retrieve data.

```
val = obj.<name>;  
val = obj{idx};
```

subsasgn(*s*, *b*)

Called when indexing a table to assign data.

```
obj.<name> = val;  
obj{idx} = val;
```

display()

Display the mapped array structure.

This method is called automatically when omitting a semicolon on a line that returns an object of this class.

3.6.7 mp.NODE_TYPE

class mp.NODE_TYPE

[mp.NODE_TYPE](#) (page 172) - Defines enumerated type for node types.

mp.NODE_TYPE Properties:

- [PQ](#) (page 172) - PQ node (= 1)
- [PV](#) (page 172) - PV node (= 2)
- [REF](#) (page 172) - reference node (= 3)
- [NONE](#) (page 172) - isolated node (= 4)

mp.NODE_TYPE Methods:

- [is_valid\(\)](#) (page 172) - returns true if the value is a valid node type

All properties are Constant properties and the class is a Sealed class. So the properties function as global constants which do not create an instance of the class, e.g. [mp.NODE_TYPE.REF](#) (page 172).

Property Summary

PQ = 1

PQ node

PV = 2

PV node

REF = 3

reference node

NONE = 4

isolated node

Method Summary

static is_valid(val)

Returns true if the value is a valid node type.

```
TorF = mp.NODE_TYPE.is_valid(val)
```

Input

val (*integer*) – node type value to check for validity

Output

TorF (*boolean*) – true if val is a valid node type

3.6.8 mp.sm_legacy_cost

class mp.sm_legacy_cost

Bases: mp.set_manager_opt_model

[mp.sm_legacy_cost](#) (page 173) - MP Set Manager class for legacy costs.

```
sm = mp.sm_legacy_cost()
sm = mp.sm_legacy_cost(label)
```

MP Set Manager class for legacy costs of the form described in [opf_model.add_legacy_cost\(\)](#) (page 227).

mp.sm_legacy_cost Properties:

- [cache](#) (page 173) - struct for caching aggregated parameters for the set

mp.sm_legacy_cost Methods:

- [sm_legacy_cost\(\)](#) (page 173) - constructor

See also [mp.set_manager](#), [mp.set_manager_opt_model](#).

Constructor Summary

sm_legacy_cost(varargin)

Constructor.

```
sm = mp.sm_legacy_cost(label)
```

Property Summary

cache = []

struct for caching aggregated parameters for legacy costs

3.7 MATPOWER Extension Classes

3.7.1 Base

mp.extension

class mp.extension

Bases: `handle`

[mp.extension](#) (page 174) - Abstract base class for MATPOWER extensions.

This class serves as the framework for the **MATPOWER extension** API, providing a way to bundle a set of class additions and modifications together into a single named package.

By default the methods in this class do nothing, but they can be overridden to customize essentially any aspect of a MATPOWER run. The first 5 methods are used to modify the default classes used to construct the task, data model converter, data, network, and/or mathematical model objects. The last 4 methods are used to add to or modify the classes used to construct the elements for each of the container types.

By convention, MATPOWER extension objects (or cell arrays of them) are named `mpx` and MATPOWER extension class names begin with `mp.xt`.

mp.extension Methods:

- [task_class\(\)](#) (page 174) - return handle to constructor for task object
- [dmc_class\(\)](#) - return handle to constructor for data model converter object
- [dm_class\(\)](#) - return handle to constructor for data model object
- [nm_class\(\)](#) - return handle to constructor for network model object
- [mm_class\(\)](#) - return handle to constructor for mathematical object
- [dmc_element_classes\(\)](#) (page 175) - return element class modifiers for data model converter elements
- [dm_element_classes\(\)](#) (page 176) - return element class modifiers for data model elements
- [nm_element_classes\(\)](#) (page 176) - return element class modifiers for network model elements
- [mm_element_classes\(\)](#) (page 176) - return element class modifiers for mathematical model elements

See the `sec_customizing` and `sec_extensions` sections in the *MATPOWER Developer's Manual* for more information, and specifically the `sec_element_classes` section and the `tab_element_class_modifiers` table for details on *element class modifiers*.

Example MATPOWER extensions:

- [mp.xt_reserves](#) (page 177) - adds fixed zonal reserves to OPF
- [mp.xt_3p](#) (page 182) - adds example prototype unbalanced three-phase elements for AC PF, CPF, and OPF

See also [mp.task](#) (page 9), [mp.dm_converter](#) (page 62), [mp.data_model](#) (page 30), [mp.net_model](#) (page 93), [mp.math_model](#) (page 124), [mp.dmc_element](#) (page 65), [mp.dm_element](#) (page 38), [mp.nm_element](#) (page 110), [mp.mm_element](#) (page 146).

Method Summary

task_class(*task_class*, *mpopt*)

Return handle to constructor for task object.

```
task_class = mpx.task_class(task_class, mpopt)
```

Inputs

- **task_class** (*function handle*) – default task constructor
- **mpopt** (*struct*) – MATPOWER options struct

Output

task_class (*function handle*) – updated task constructor

dm_converter_class(*dmc_class, fmt, mpopt*)

Return handle to constructor for data model converter object.

```
dmc_class = mpx.dm_converter_class(dmc_class, fmt, mpopt)
```

Inputs

- **dmc_class** (*function handle*) – default data model converter constructor
- **fmt** (*char array*) – data format tag, e.g. 'mpc2'
- **mpopt** (*struct*) – MATPOWER options struct

Output

dmc_class (*function handle*) – updated data model converter constructor

data_model_class(*dm_class, task_tag, mpopt*)

Return handle to constructor for data model object.

```
dm_class = mpx.data_model_class(dm_class, task_tag, mpopt)
```

Inputs

- **dm_class** (*function handle*) – default data model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

dm_class (*function handle*) – updated data model constructor

network_model_class(*nm_class, task_tag, mpopt*)

Return handle to constructor for network model object.

```
nm_class = mpx.network_model_class(nm_class, task_tag, mpopt)
```

Inputs

- **nm_class** (*function handle*) – default network model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

nm_class (*function handle*) – updated network model constructor

math_model_class(*mm_class, task_tag, mpopt*)

Return handle to constructor for mathematical model object.

```
mm_class = mpx.math_model_class(mm_class, task_tag, mpopt)
```

Inputs

- **mm_class** (*function handle*) – default math model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

mm_class (*function handle*) – updated math model constructor

dmc_element_classes(*dmc_class, fmt, mpopt*)

Return element class modifiers for data model converter elements.

```
dmc_elements = mpx.dmc_element_classes(dmc_class, fmt, mpopt)
```

Inputs

- **dmc_class** (*function handle*) – data model converter constructor
- **fmt** (*char array*) – data format tag, e.g. 'mpc2'
- **mpopt** (*struct*) – MATPOWER options struct

Output

dm_elements (*cell array*) – element class modifiers (see `tab_element_class_modifiers` in the *MATPOWER Developer's Manual*)

dm_element_classes(*dm_class, task_tag, mpopt*)

Return element class modifiers for data model elements.

```
dm_elements = mpx.dm_element_classes(dm_class, task_tag, mpopt)
```

Inputs

- **dm_class** (*function handle*) – data model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

dm_elements (*cell array*) – element class modifiers (see `tab_element_class_modifiers` in the *MATPOWER Developer's Manual*)

nm_element_classes(*nm_class, task_tag, mpopt*)

Return element class modifiers for network model elements.

```
nm_elements = mpx.nm_element_classes(nm_class, task_tag, mpopt)
```

Inputs

- **nm_class** (*function handle*) – network model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

nm_elements (*cell array*) – element class modifiers (see `tab_element_class_modifiers` in the *MATPOWER Developer's Manual*)

mm_element_classes(*mm_class, task_tag, mpopt*)

Return element class modifiers for mathematical model elements.

```
mm_elements = mpx.mm_element_classes(mm_class, task_tag, mpopt)
```

Inputs

- **mm_class** (*function handle*) – mathematical model constructor
- **task_tag** (*char array*) – task tag, e.g. 'PF', 'CPF', 'OPF'
- **mpopt** (*struct*) – MATPOWER options struct

Output

mm_elements (*cell array*) – element class modifiers (see `tab_element_class_modifiers` in the *MATPOWER Developer's Manual*)

3.7.2 OPF Fixed Zonal Reserves Extension

mp.xt_reserves

class mp.xt_reserves

Bases: [mp.extension](#) (page 174)

[mp.xt_reserves](#) (page 177) - MATPOWER extension for OPF with fixed zonal reserves.

For OPF problems, this extension adds two types of elements to the data and mathematical model containers, as well as the data model converter.

The 'reserve_gen' element handles all of the per-generator aspects, such as reserve cost and quantity limit parameters, reserve variables, and constraints on reserve capacity.

The 'reserve_zone' element handles the per-zone aspects, such as generator/zone mappings, zonal reserve requirement parameters and constraints, and zonal reserve prices.

mp.xt_reserves Methods:

- [dmc_element_classes\(\)](#) (page 177) - add two classes to data model converter elements
- [dm_element_classes\(\)](#) (page 177) - add two classes to data model elements
- [mm_element_classes\(\)](#) (page 177) - add two classes to mathematical model elements

See the [sec_customizing](#) and [sec_extensions](#) sections in the *MATPOWER Developer's Manual* for more information, and specifically the [sec_element_classes](#) section and the [tab_element_class_modifiers](#) table for details on *element class modifiers*.

See also [mp.extension](#) (page 174).

Method Summary

dmc_element_classes(*dmc_class, fmt, mpopt*)

Add two classes to data model converter elements.

For 'mpc2' data formats, adds the classes:

- [mp.dmce_reserve_gen_mpc2](#) (page 178)
- [mp.dmce_reserve_zone_mpc2](#) (page 178)

dm_element_classes(*dm_class, task_tag, mpopt*)

Add two classes to data model elements.

For 'OPF' tasks, adds the classes:

- [mp.dme_reserve_gen](#) (page 179)
- [mp.dme_reserve_zone](#) (page 180)

mm_element_classes(*mm_class, task_tag, mpopt*)

Add two classes to mathematical model elements.

For 'OPF' tasks, adds the classes:

- [mp.mme_reserve_gen](#) (page 181)
- [mp.mme_reserve_zone](#) (page 181)

Other classes belonging to [mp.xt_reserves](#) (page 177) extension:

mp.dmce_reserve_gen_mpc2

class mp.dmce_reserve_gen_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_reserve_gen_mpc2](#) (page 178) - Data model converter element for reserve generator for MATPOWER case v2.

Method Summary

name()
data_field()
data_subs()
get_import_size(*mpc*)
get_export_size(*dme*)
table_var_map(*dme, mpc*)
import_cost(*mpc, spec, vn*)
import_qty(*mpc, spec, vn*)
import_ramp(*mpc, spec, vn*)
import(*dme, mpc, varargin*)

mp.dmce_reserve_zone_mpc2

class mp.dmce_reserve_zone_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_reserve_zone_mpc2](#) (page 178) - Data model converter element for reserve zone for MATPOWER case v2.

Method Summary

name()
data_field()
data_subs()
table_var_map(*dme, mpc*)
import_req(*mpc, spec, vn*)
import_zones(*mpc, spec, vn*)

mp.dme_reserve_gen**class mp.dme_reserve_gen**

Bases: [mp.dm_element](#) (page 38), [mp.dme_sharedopf](#) (page 61)

[mp.dme_reserve_gen](#) (page 179) - Data model element for reserve generator.

Implements the data element model for reserve generator elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>gen</code>	<i>integer</i>	ID (uid) of corresponding generator
<code>cost</code>	<i>double</i>	reserve cost (u/MW) ¹
<code>qty</code>	<i>double</i>	available reserve quantity (MW)
<code>ramp10</code>	<i>double</i>	10-minute ramp rate (MW)
<code>r</code>	<i>double</i>	r , reserve allocation (MW)
<code>r_lb</code>	<i>double</i>	lower bound on reserve allocation (MW)
<code>r_ub</code>	<i>double</i>	upper bound on reserve allocation (MW)
<code>total_cost</code>	<i>double</i>	total cost of allocated reserves (u) ¹
<code>prc</code>	<i>double</i>	reserve price (u/MVA_r) ¹
<code>mu_lb</code>	<i>double</i>	shadow price on r lower bound (u/MW) ¹
<code>mu_ub</code>	<i>double</i>	shadow price on r upper bound (u/MW) ¹
<code>mu_pg_ub</code>	<i>double</i>	shadow price on capacity constraint (u/MW) ¹

Property Summary**gen**

index of online gens (for online reserve gens)

r_ub

upper bound on reserve qty (p.u.) for units that are on

Method Summary**name()****label()****labels()****main_table_var_names()****export_vars()****export_vars_offline_val()****update_status(dm)****build_params(dm)****pp_have_section_sum(mppopt, pp_args)****pp_data_sum(dm, rows, out_e, mppopt, fd, pp_args)**

¹ Here u denotes the units of the objective function, e.g. USD.

```
pp_have_section_det(mpop, pp_args)
pp_get_headers_det(dm, out_e, mpop, pp_args)
pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)
pp_have_section_lim(mpop, pp_args)
pp_binding_rows_lim(dm, out_e, mpop, pp_args)
pp_get_headers_lim(dm, out_e, mpop, pp_args)
pp_data_row_lim(dm, k, out_e, mpop, fd, pp_args)
pp_get_footers_det(dm, out_e, mpop, pp_args)
```

mp.dme_reserve_zone

class mp.dme_reserve_zone

Bases: [mp.dm_element](#) (page 38), [mp.dme_sharedopf](#) (page 61)

[mp.dme_reserve_zone](#) (page 180) - Data model element for reserve zone.

Implements the data element model for reserve zone elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>req</code>	<i>double</i>	zonal reserve requirement (<i>MW</i>)
<code>zones</code>	<i>integer</i>	matrix defining generators included in the zone
<code>prc</code>	<i>double</i>	zonal reserve price (<i>u/MW</i>) ¹

Property Summary

zones

zone map for online [zones](#) (page 180) / gens

req

reserve requirement in p.u. for each active zone

Method Summary

name()

label()

labels()

main_table_var_names()

export_vars()

export_vars_offline_val()

¹ Here *u* denotes the units of the objective function, e.g. USD.

```

update_status(dm)

build_params(dm)

pp_have_section_det(mpop, pp_args)

pp_get_headers_det(dm, out_e, mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)

```

mp.mme_reserve_gen

class mp.mme_reserve_gen

Bases: [mp.mm_element](#) (page 146)

[mp.mme_reserve_gen](#) (page 181) - Mathematical model element for reserve generator.

Math model element class for reserve generator elements.

Implements methods for adding reserve variables, costs, and per-generator reserve constraints, and for updating the output data in the corresponding data model element for in-service reserve generators from the math model solution.

Method Summary

```

name()

add_vars(mm, nm, dm, mpop)

add_costs(mm, nm, dm, mpop)

add_constraints(mm, nm, dm, mpop)

data_model_update_on(mm, nm, dm, mpop)

```

mp.mme_reserve_zone

class mp.mme_reserve_zone

Bases: [mp.mm_element](#) (page 146)

[mp.mme_reserve_zone](#) (page 181) - Mathematical model element for reserve zone.

Math model element class for reserve zone elements.

Implements methods for adding reserve zone constraints, and for updating the output data in the corresponding data model element for in-service reserve zones from the math model solution.

Method Summary

```

name()

add_constraints(mm, nm, dm, mpop)

data_model_update_on(mm, nm, dm, mpop)

```


3.7.3 Three-Phase Prototype Extension

`mp.xt_3p`

`class mp.xt_3p`

Bases: [mp.extension](#) (page 174)

[mp.xt_3p](#) (page 182) - MATPOWER extension to add unbalanced three-phase elements.

For AC power flow, continuation power flow, and optimal power flow problems, adds six new element types:

- 'bus3p' - 3-phase bus
- 'gen3p' - 3-phase generator
- 'load3p' - 3-phase load
- 'line3p' - 3-phase distribution line
- 'xfmr3p' - 3-phase transformer
- 'buslink' - 3-phase to single phase linking element

No changes are required for the task or container classes, so only the `..._element_classes` methods are overridden.

The set of data model element classes depends on the task, with each OPF class inheriting from the corresponding class used for PF and CPF.

The set of network model element classes depends on the formulation, specifically whether cartesian or polar representations are used for voltages.

And the set of mathematical model element classes depends on both the task and the formulation.

`mp.xt_3p` Methods:

- [dmc_element_classes\(\)](#) (page 182) - add six classes to data model converter elements
- [dm_element_classes\(\)](#) (page 183) - add six classes to data model elements
- [nm_element_classes\(\)](#) (page 183) - add six classes to network model elements
- [mm_element_classes\(\)](#) (page 183) - add six classes to mathematical model elements

See the `sec_customizing` and `sec_extensions` sections in the *MATPOWER Developer's Manual* for more information, and specifically the `sec_element_classes` section and the `tab_element_class_modifiers` table for details on *element class modifiers*.

See also [mp.extension](#) (page 174).

Method Summary

`dmc_element_classes(dmc_class,fmt,mpopt)`

Add six classes to data model converter elements.

For 'mpc2 data formats, adds the classes:

- [mp.dmce_bus3p_mpc2](#) (page 184)
- [mp.dmce_gen3p_mpc2](#) (page 184)
- [mp.dmce_load3p_mpc2](#) (page 184)
- [mp.dmce_line3p_mpc2](#) (page 185)
- [mp.dmce_xfmr3p_mpc2](#) (page 185)

- `mp.dmce_buslink_mpc2` (page 186)

dm_element_classes(*dm_class*, *task_tag*, *mpopt*)

Add six classes to data model elements.

For 'PF' and 'CPF' tasks, adds the classes:

- `mp.dme_bus3p` (page 186)
- `mp.dme_gen3p` (page 187)
- `mp.dme_load3p` (page 189)
- `mp.dme_line3p` (page 190)
- `mp.dme_xfmr3p` (page 192)
- `mp.dme_buslink` (page 194)

For 'OPF' tasks, adds the classes:

- `mp.dme_bus3p_opf` (page 195)
- `mp.dme_gen3p_opf` (page 195)
- `mp.dme_load3p_opf` (page 195)
- `mp.dme_line3p_opf` (page 196)
- `mp.dme_xfmr3p_opf` (page 196)
- `mp.dme_buslink_opf` (page 196)

nm_element_classes(*nm_class*, *task_tag*, *mpopt*)

Add six classes to network model elements.

For *cartesian* voltage formulations, adds the classes:

- `mp.nme_bus3p_acc` (page 197)
- `mp.nme_gen3p_acc` (page 198)
- `mp.nme_load3p` (page 198)
- `mp.nme_line3p` (page 199)
- `mp.nme_xfmr3p` (page 199)
- `mp.nme_buslink_acc` (page 200)

For *polar* voltage formulations, adds the classes:

- `mp.nme_bus3p_acp` (page 197)
- `mp.nme_gen3p_acp` (page 198)
- `mp.nme_load3p` (page 198)
- `mp.nme_line3p` (page 199)
- `mp.nme_xfmr3p` (page 199)
- `mp.nme_buslink_acp` (page 200)

mm_element_classes(*mm_class*, *task_tag*, *mpopt*)

Add five classes to mathematical model elements.

For 'PF' and 'CPF' tasks, adds the classes:

- `mp.mme_bus3p` (page 200)
- `mp.mme_gen3p` (page 201)
- `mp.mme_line3p` (page 201)
- `mp.mme_xfmr3p` (page 201)
- `mp.mme_buslink_pf_acc` (page 202) (*cartesian*) or `mp.mme_buslink_pf_acp` (page 203) (*polar*)

For 'OPF' tasks, adds the classes:

- `mp.mme_bus3p_opf_acc` (page 203) (*cartesian*) or `mp.mme_bus3p_opf_acp` (page 203) (*polar*)
- `mp.mme_gen3p_opf` (page 204)
- `mp.mme_line3p_opf` (page 204)
- `mp.mme_xfmr3p_opf` (page 204)
- `mp.mme_buslink_opf_acc` (page 205) (*cartesian*) or `mp.mme_buslink_opf_acp` (page 205) (*polar*)

Data model converter element classes belonging to `mp.xt_3p` (page 182) extension:

mp.dmce_bus3p_mpc2

class mp.dmce_bus3p_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_bus3p_mpc2](#) (page 184) - Data model converter element for 3-phase bus for MATPOWER case v2.

Method Summary

name()

data_field()

table_var_map(*dme*, *mpc*)

bus_status_import(*mpc*, *spec*, *vn*, *c*)

mp.dmce_gen3p_mpc2

class mp.dmce_gen3p_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_gen3p_mpc2](#) (page 184) - Data model converter element for 3-phase generator for MATPOWER case v2.

Method Summary

name()

data_field()

table_var_map(*dme*, *mpc*)

mp.dmce_load3p_mpc2

class mp.dmce_load3p_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_load3p_mpc2](#) (page 184) - Data model converter element for 3-phase load for MATPOWER case v2.

Property Summary

bus

Method Summary

name()

`data_field()`

`table_var_map(dme, mpc)`

mp.dmce_line3p_mpc2

class `mp.dmce_line3p_mpc2`

Bases: `mp.dmc_element` (page 65)

`mp.dmce_line3p_mpc2` (page 185) - Data model converter element for 3-phase line for MATPOWER case v2.

Method Summary

`name()`

`data_field()`

`table_var_map(dme, mpc)`

`create_line_construction_table(dme, lc)`

`import(dme, mpc, varargin)`

mp.dmce_xfmr3p_mpc2

class `mp.dmce_xfmr3p_mpc2`

Bases: `mp.dmc_element` (page 65)

`mp.dmce_xfmr3p_mpc2` (page 185) - Data model converter element for 3-phase transformer for MATPOWER case v2.

Method Summary

`name()`

`data_field()`

`table_var_map(dme, mpc)`

mp.dmce_buslink_mpc2

class mp.dmce_buslink_mpc2

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_buslink_mpc2](#) (page 186) - Data model converter element for 1-to-3-phase buslink for MATPOWER case v2.

Method Summary

name()

data_field()

table_var_map(dme, mpc)

Data model element classes belonging to [mp.xt_3p](#) (page 182) extension:

mp.dme_bus3p

class mp.dme_bus3p

Bases: [mp.dm_element](#) (page 38)

[mp.dme_bus3p](#) (page 186) - Data model element for 3-phase bus.

Implements the data element model for 3-phase bus elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
type	<i>integer</i>	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
base_kv	<i>double</i>	base line-to-line voltage (<i>kV</i>)
vm1	<i>double</i>	phase 1 voltage magnitude (<i>p.u.</i>)
vm2	<i>double</i>	phase 2 voltage magnitude (<i>p.u.</i>)
vm3	<i>double</i>	phase 3 voltage magnitude (<i>p.u.</i>)
va1	<i>double</i>	phase 1 voltage angle (<i>degrees</i>)
va2	<i>double</i>	phase 2 voltage angle (<i>degrees</i>)
va3	<i>double</i>	phase 3 voltage angle (<i>degrees</i>)

Property Summary

type

node [type](#) (page 186) vector for buses that are on

vm1_start

initial phase 1 voltage magnitudes (*p.u.*) for buses that are on

vm2_start

initial phase 2 voltage magnitudes (*p.u.*) for buses that are on

vm3_start

initial phase 3 voltage magnitudes (*p.u.*) for buses that are on

va1_start
initial phase 1 voltage angles (radians) for buses that are on

va2_start
initial phase 2 voltage angles (radians) for buses that are on

va3_start
initial phase 3 voltage angles (radians) for buses that are on

vm_control
true if voltage is controlled, for buses that are on

Method Summary

name()

label()

labels()

main_table_var_names()

init_status(dm)

update_status(dm)

build_params(dm)

pp_have_section_det(mpop, pp_args)

pp_get_headers_det(dm, out_e, mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)

mp.dme_gen3p

class mp.dme_gen3p

Bases: [mp.dm_element](#) (page 38)

[mp.dme_gen3p](#) (page 187) - Data model element for 3-phase generator.

Implements the data element model for 3-phase generator elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus	<i>integer</i>	bus ID (uid) of 3-phase bus
vm1_setpoint	<i>double</i>	phase 1 voltage magnitude setpoint (<i>p.u.</i>)
vm2_setpoint	<i>double</i>	phase 2 voltage magnitude setpoint (<i>p.u.</i>)
vm3_setpoint	<i>double</i>	phase 3 voltage magnitude setpoint (<i>p.u.</i>)
pg1	<i>double</i>	phase 1 active power output (<i>kW</i>)
pg2	<i>double</i>	phase 2 active power output (<i>kW</i>)
pg3	<i>double</i>	phase 3 active power output (<i>kW</i>)
qg1	<i>double</i>	phase 1 reactive power output (<i>kVAr</i>)
qg2	<i>double</i>	phase 2 reactive power output (<i>kVAr</i>)
qg3	<i>double</i>	phase 3 reactive power output (<i>kVAr</i>)

Property Summary

bus

bus (page 188) index vector (all gens)

bus_on

vector of indices into online buses for gens that are on

pg1_start

initial phase 1 active power (p.u.) for gens that are on

pg2_start

initial phase 2 active power (p.u.) for gens that are on

pg3_start

initial phase 3 active power (p.u.) for gens that are on

qg1_start

initial phase 1 reactive power (p.u.) for gens that are on

qg2_start

initial phase 2 reactive power (p.u.) for gens that are on

qg3_start

initial phase 3 reactive power (p.u.) for gens that are on

vm1_setpoint

phase 1 generator voltage setpoint for gens that are on

vm2_setpoint

phase 2 generator voltage setpoint for gens that are on

vm3_setpoint

phase 3 generator voltage setpoint for gens that are on

Method Summary

name()**label()****labels()****cxn_type()****cxn_idx_prop()****main_table_var_names()****initialize(*dm*)****update_status(*dm*)****apply_vm_setpoint(*dm*)****build_params(*dm*)****pp_have_section_sum(*mpopt*, *pp_args*)****pp_data_sum(*dm*, *rows*, *out_e*, *mpopt*, *fd*, *pp_args*)**

```

pp_have_section_det(mpop, pp_args)

pp_get_headers_det(dm, out_e, mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)

```

mp.dme_load3p

class mp.dme_load3p

Bases: [mp.dm_element](#) (page 38)

[mp.dme_load3p](#) (page 189) - Data model element for 3-phase load.

Implements the data element model for 3-phase load elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus	<i>integer</i>	bus ID (uid) of 3-phase bus
pd1	<i>double</i>	phase 1 active power demand (<i>kW</i>)
pd2	<i>double</i>	phase 2 active power demand (<i>kW</i>)
pd3	<i>double</i>	phase 3 active power demand (<i>kW</i>)
pf1	<i>double</i>	phase 1 power factor
pf2	<i>double</i>	phase 2 power factor
pf3	<i>double</i>	phase 3 power factor

Property Summary

bus

[bus](#) (page 189) index vector (all loads)

pd1

phase 1 active power demand (p.u.) for loads that are on

pd2

phase 2 active power demand (p.u.) for loads that are on

pd3

phase 3 active power demand (p.u.) for loads that are on

pf1

phase 1 power factor for loads that are on

pf2

phase 2 power factor for loads that are on

pf3

phase 3 power factor for loads that are on

Method Summary

name()

label()


```
labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

initialize(dm)

update_status(dm)

build_params(dm)

pp_have_section_sum(mpop, pp_args)

pp_data_sum(dm, rows, out_e, mpop, fd, pp_args)

pp_have_section_det(mpop, pp_args)

pp_get_headers_det(dm, out_e, mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)
```

mp.dme_line3p

class mp.dme_line3p

Bases: [mp.dm_element](#) (page 38)

[mp.dme_line3p](#) (page 190) - Data model element for 3-phase line.

Implements the data element model for 3-phase distribution line elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus_fr	integer	bus ID (uid) of “from” 3-phase bus
bus_to	integer	bus ID (uid) of “to” 3-phase bus
lc	double	index into line construction table
len	double	line length (<i>miles</i>)
p11_fr	double	phase 1 active power injection at “from” end (<i>kW</i>)
q11_fr	double	phase 1 reactive power injection at “from” end (<i>kVAr</i>)
p12_fr	double	phase 2 active power injection at “from” end (<i>kW</i>)
q12_fr	double	phase 2 reactive power injection at “from” end (<i>kVAr</i>)
p13_fr	double	phase 3 active power injection at “from” end (<i>kW</i>)
q13_fr	double	phase 3 reactive power injection at “from” end (<i>kVAr</i>)
p11_to	double	phase 1 active power injection at “to” end (<i>kW</i>)
q11_to	double	phase 1 reactive power injection at “to” end (<i>kVAr</i>)
p12_to	double	phase 2 active power injection at “to” end (<i>kW</i>)
q12_to	double	phase 2 reactive power injection at “to” end (<i>kVAr</i>)
p13_to	double	phase 3 active power injection at “to” end (<i>kW</i>)
q13_to	double	phase 3 reactive power injection at “to” end (<i>kVAr</i>)

The line construction table in the `lc_tab` property is defined as a table with the following columns:

Name	Type	Description
<code>id</code>	<i>integer</i>	unique line construction ID, referenced from <code>lc</code> column of main data table
<code>r</code>	<i>double</i>	6 resistance parameters for forming symmetric 3x3 series impedance matrix (<i>Ohms per mile</i>)
<code>x</code>	<i>double</i>	6 reactance parameters for forming symmetric 3x3 series impedance matrix (<i>Ohms per mile</i>)
<code>c</code>	<i>double</i>	6 susceptance parameters for forming symmetric 3x3 shunt susceptance matrix (<i>nF per mile</i>)

Property Summary

fbus

bus index vector for “from” bus (all lines)

tbus

bus index vector for “to” bus (all lines)

freq

system frequency, in Hz

lc

index into `lc_tab` for lines that are on

lc_y_idx

index into `ys` and `yc` for lines that are on

len

length for lines that are on

lc_tab

line construction table

ys

cell array of 3x3 series admittance matrices for `lc` rows

yc

cell array of 3x3 shunt admittance matrices for `lc` rows

Method Summary

name()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

lc_table_var_names()

create_line_construction_table(*id, r, x, c*)

initialize(*dm*)

update_status(*dm*)

build_params(*dm*)

vec2symmat(*v*)

Make a symmetric matrix from a vector of 6 values.

symmat2vec(*M*)

Extract a vector of 6 values from a matrix assumed to be symmetric.

pretty_print(*dm, section, out_e, mpopt, fd, pp_args*)

pp_have_section_sum(*mpopt, pp_args*)

pp_data_sum(*dm, rows, out_e, mpopt, fd, pp_args*)

pp_have_section_det(*mpopt, pp_args*)

pp_get_headers_det(*dm, out_e, mpopt, pp_args*)

pp_data_row_det(*dm, k, out_e, mpopt, fd, pp_args*)

mp.dme_xfmr3p

class `mp.dme_xfmr3p`

Bases: `mp.dm_element` (page 38)

`mp.dme_xfmr3p` (page 192) - Data model element for 3-phase transformer.

Implements the data element model for 3-phase transformer elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>bus_fr</code>	<i>integer</i>	bus ID (uid) of “from” 3-phase bus
<code>bus_to</code>	<i>integer</i>	bus ID (uid) of “to” 3-phase bus
<code>r</code>	<i>double</i>	series resistance (<i>p.u.</i>)
<code>x</code>	<i>double</i>	series reactance (<i>p.u.</i>)
<code>base_kva</code>	<i>double</i>	transformer kVA base (<i>kVA</i>)
<code>base_kv</code>	<i>double</i>	transformer kV base (<i>kV</i>)
<code>p11_fr</code>	<i>double</i>	phase 1 active power injection at “from” end (<i>kW</i>)
<code>q11_fr</code>	<i>double</i>	phase 1 reactive power injection at “from” end (<i>kVAr</i>)
<code>p12_fr</code>	<i>double</i>	phase 2 active power injection at “from” end (<i>kW</i>)
<code>q12_fr</code>	<i>double</i>	phase 2 reactive power injection at “from” end (<i>kVAr</i>)
<code>p13_fr</code>	<i>double</i>	phase 3 active power injection at “from” end (<i>kW</i>)
<code>q13_fr</code>	<i>double</i>	phase 3 reactive power injection at “from” end (<i>kVAr</i>)
<code>p11_to</code>	<i>double</i>	phase 1 active power injection at “to” end (<i>kW</i>)
<code>q11_to</code>	<i>double</i>	phase 1 reactive power injection at “to” end (<i>kVAr</i>)
<code>p12_to</code>	<i>double</i>	phase 2 active power injection at “to” end (<i>kW</i>)
<code>q12_to</code>	<i>double</i>	phase 2 reactive power injection at “to” end (<i>kVAr</i>)
<code>p13_to</code>	<i>double</i>	phase 3 active power injection at “to” end (<i>kW</i>)
<code>q13_to</code>	<i>double</i>	phase 3 reactive power injection at “to” end (<i>kVAr</i>)

Property Summary

`fbus`

bus index vector for “from” bus (all transformers)

`tbus`

bus index vector for “to” bus (all transformers)

`r`

series resistance (*p.u.*) for transformers that are on

`x`

series reactance (*p.u.*) for transformers that are on

Method Summary

`name()`

`label()`

`labels()`

`cnx_type()`

`cnx_idx_prop()`

`main_table_var_names()`

`initialize(dm)`

`update_status(dm)`

`build_params(dm)`

`pretty_print(dm, section, out_e, mpopt, fd, pp_args)`

```
pp_have_section_sum(mpop, pp_args)
pp_data_sum(dm, rows, out_e, mpop, fd, pp_args)
pp_have_section_det(mpop, pp_args)
pp_get_headers_det(dm, out_e, mpop, pp_args)
pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)
```

mp.dme_buslink

class mp.dme_buslink

Bases: [mp.dm_element](#) (page 38)

[mp.dme_buslink](#) (page 194) - Data model element for 1-to-3-phase buslink.

Implements the data element model for 1-to-3-phase buslink elements.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
bus	<i>integer</i>	bus ID (uid) of single phase bus
bus3p	<i>integer</i>	bus ID (uid) of 3-phase bus

Property Summary

bus

[bus](#) (page 194) index vector (all buslinks)

bus3p

[bus3p](#) (page 194) index vector (all buslinks)

pg1_start

initial phase 1 active power (p.u.) for buslinks that are on

pg2_start

initial phase 2 active power (p.u.) for buslinks that are on

pg3_start

initial phase 3 active power (p.u.) for buslinks that are on

qg1_start

initial phase 1 reactive power (p.u.) for buslinks that are on

qg2_start

initial phase 2 reactive power (p.u.) for buslinks that are on

qg3_start

initial phase 3 reactive power (p.u.) for buslinks that are on

Method Summary

name()

```

label()
labels()
cxn_type()
cxn_idx_prop()
main_table_var_names()
initialize(dm)
update_status(dm)
build_params(dm)
pp_have_section_det(mpop, pp_args)
pp_get_headers_det(dm, out_e, mpop, pp_args)
pp_data_row_det(dm, k, out_e, mpop, fd, pp_args)

```

mp.dme_bus3p_opf

class mp.dme_bus3p_opf

Bases: [mp.dme_bus3p](#) (page 186), [mp.dme_shared_opf](#) (page 61)

[mp.dme_bus3p_opf](#) (page 195) - Data model element for 3-phase bus for OPF.

To parent class [mp.dme_bus3p](#) (page 186), adds pretty-printing for **lim** sections.

mp.dme_gen3p_opf

class mp.dme_gen3p_opf

Bases: [mp.dme_gen3p](#) (page 187), [mp.dme_shared_opf](#) (page 61)

[mp.dme_gen3p_opf](#) (page 195) - Data model element for 3-phase generator for OPF.

To parent class [mp.dme_gen3p](#) (page 187), adds pretty-printing for **lim** sections.

mp.dme_load3p_opf

class mp.dme_load3p_opf

Bases: [mp.dme_load3p](#) (page 189), [mp.dme_shared_opf](#) (page 61)

[mp.dme_load3p_opf](#) (page 195) - Data model element for 3-phase load for OPF.

To parent class [mp.dme_load3p](#) (page 189), adds pretty-printing for **lim** sections.

mp.dme_line3p_opf

class mp.dme_line3p_opf

Bases: [mp.dme_line3p](#) (page 190), [mp.dme_shared_opf](#) (page 61)

[mp.dme_line3p_opf](#) (page 196) - Data model element for 3-phase line for OPF.

To parent class [mp.dme_line3p](#) (page 190), adds pretty-printing for **lim** sections.

mp.dme_xfmr3p_opf

class mp.dme_xfmr3p_opf

Bases: [mp.dme_xfmr3p](#) (page 192), [mp.dme_shared_opf](#) (page 61)

[mp.dme_xfmr3p_opf](#) (page 196) - Data model element for 3-phase transformer for OPF.

To parent class [mp.dme_xfmr3p](#) (page 192), adds pretty-printing for **lim** sections.

mp.dme_buslink_opf

class mp.dme_buslink_opf

Bases: [mp.dme_buslink](#) (page 194), [mp.dme_shared_opf](#) (page 61)

[mp.dme_buslink_opf](#) (page 196) - Data model element for 1-to-3-phase buslink for OPF.

To parent class [mp.dme_buslink](#) (page 194), adds pretty-printing for **lim** sections.

Network model element classes belonging to [mp.xt_3p](#) (page 182) extension:

mp.nme_bus3p

class mp.nme_bus3p

Bases: [mp.nm_element](#) (page 110)

[mp.nme_bus3p](#) (page 196) - Network model element abstract base class for 3-phase bus.

Implements the network model element for 3-phase bus elements, with 3 nodes per 3-phase bus.

Implements [node_types\(\)](#) (page 196) method.

Method Summary

name()

nm()

node_types(*nm, dm, idx*)

```
ntv = nme.node_types(nm, dm, idx)
[ref, pv, pq] = nme.node_types(nm, dm, idx)
```

Called by the [node_types\(\)](#) (page 101) method of [mp.net_model](#) (page 93).

mp.nme_bus3p_acc**class** mp.nme_bus3p_accBases: [mp.nme_bus3p](#) (page 196), [mp.form_acc](#) (page 85)[mp.nme_bus3p_acc](#) (page 197) - Network model element for 3-phase bus, AC cartesian voltage formulation.Adds voltage variables Vr3 and Vi3 to the network model and inherits from [mp.form_acc](#) (page 85).**Method Summary****add_vvars**(nm, dm, idx)**mp.nme_bus3p_acp****class** mp.nme_bus3p_acpBases: [mp.nme_bus3p](#) (page 196), [mp.form_acp](#) (page 89)[mp.nme_bus3p_acp](#) (page 197) - Network model element for 3-phase bus, AC polar voltage formulation.Adds voltage variables Va3 and Vm3 to the network model and inherits from [mp.form_acp](#) (page 89).**Method Summary****add_vvars**(nm, dm, idx)**mp.nme_gen3p****class** mp.nme_gen3pBases: [mp.nm_element](#) (page 110)[mp.nme_gen3p](#) (page 197) - Network model element abstract base class for 3-phase generator.

Implements the network model element for 3-phase generator elements, with 3 ports and 3 non-voltage states per 3-phase generator.

Adds non-voltage state variables Pg3 and Qg3 to the network model and builds the parameter N.**Method Summary****name**()**np**()**nz**()**add_zvars**(nm, dm, idx)

build_params(*nm*, *dm*)

mp.nme_gen3p_acc

class mp.nme_gen3p_acc

Bases: [mp.nme_gen3p](#) (page 197), [mp.form_acc](#) (page 85)

[mp.nme_gen3p_acc](#) (page 198) - Network model element for 3-phase generator, AC cartesian voltage formulation.

Inherits from [mp.form_acc](#) (page 85).

mp.nme_gen3p_acp

class mp.nme_gen3p_acp

Bases: [mp.nme_gen3p](#) (page 197), [mp.form_acp](#) (page 89)

[mp.nme_gen3p_acp](#) (page 198) - Network model element for 3-phase generator, AC polar voltage formulation.

Inherits from [mp.form_acp](#) (page 89).

mp.nme_load3p

class mp.nme_load3p

Bases: [mp.nm_element](#) (page 110), [mp.form_acp](#) (page 89)

[mp.nme_load3p](#) (page 198) - Network model element for 3-phase load.

Implements the network model element for 3-phase load elements, with 3 ports per 3-phase load.

Builds the parameter `s` and inherits from [mp.form_acp](#) (page 89).

Method Summary

name()

np()

build_params(*nm*, *dm*)

mp.nme_line3p

class mp.nme_line3p

Bases: [mp.nm_element](#) (page 110), [mp.form_acp](#) (page 89)

[mp.nme_line3p](#) (page 199) - Network model element for 3-phase line.

Implements the network model element for 3-phase line elements, with 6 ports per 3-phase line.

Implements building of the admittance parameter \underline{Y} for 3-phase lines and inherits from [mp.form_acp](#) (page 89).

Method Summary

`name()`

`np()`

`build_params(nm, dm)`

`vec2symmat_stacked(vv)`

mp.nme_xfmr3p

class mp.nme_xfmr3p

Bases: [mp.nm_element](#) (page 110), [mp.form_acp](#) (page 89)

[mp.nme_xfmr3p](#) (page 199) - Network model element for 3-phase transformer.

Implements the network model element for 3-phase transformer elements, with 6 ports per transformer.

Implements building of the admittance parameter \underline{Y} for 3-phase transformers and inherits from [mp.form_acp](#) (page 89).

Method Summary

`name()`

`np()`

`build_params(nm, dm)`

mp.nme_buslink

class mp.nme_buslink

Bases: [mp.nm_element](#) (page 110)

[mp.nme_buslink](#) (page 199) - Network model element abstract base class for 1-to-3-phase buslink.

Implements the network model element for 1-to-3-phase buslink elements, with 4 ports and 3 non-voltage states per buslink.

Adds non-voltage state variables \underline{Plink} and \underline{Qlink} to the network model, builds the parameter \underline{N} , and constructs voltage constraints.

Method Summary

`name()`
`np()`
`nz()`
`add_zvars(nm, dm, idx)`
`build_params(nm, dm)`
`voltage_constraints()`

mp.nme_buslink_acc

class `mp.nme_buslink_acc`

Bases: `mp.nme_buslink` (page 199), `mp.form_acc` (page 85)

`mp.nme_buslink_acc` (page 200) - Network model element for 1-to-3-phase buslink, AC cartesian voltage formulation.

Inherits from `mp.form_acc` (page 85).

mp.nme_buslink_acp

class `mp.nme_buslink_acp`

Bases: `mp.nme_buslink` (page 199), `mp.form_acp` (page 89)

`mp.nme_buslink_acp` (page 200) - Network model element for 1-to-3-phase buslink, AC polar voltage formulation.

Inherits from `mp.form_acp` (page 89).

Mathematical model element classes belonging to `mp.xt_3p` (page 182) extension:

mp.mme_bus3p

class `mp.mme_bus3p`

Bases: `mp.mm_element` (page 146)

`mp.mme_bus3p` (page 200) - Math model element for 3-phase bus.

Math model element base class for 3-phase bus elements.

Implements method for updating the output data in the corresponding data model element for in-service 3-phase buses from the math model solution.

Method Summary

`name()`

`data_model_update_on(mm, nm, dm, mpopt)`

mp.mme_gen3p

class `mp.mme_gen3p`

Bases: `mp.mm_element` (page 146)

`mp.mme_gen3p` (page 201) - Math model element for 3-phase generator.

Math model element base class for 3-phase generator elements.

Implements method for updating the output data in the corresponding data model element for in-service 3-phase generators from the math model solution.

Method Summary

`name()`

`data_model_update_on(mm, nm, dm, mpopt)`

mp.mme_line3p

class `mp.mme_line3p`

Bases: `mp.mm_element` (page 146)

`mp.mme_line3p` (page 201) - Math model element for 3-phase line.

Math model element base class for 3-phase line elements.

Implements method for updating the output data in the corresponding data model element for in-service 3-phase lines from the math model solution.

Method Summary

`name()`

`data_model_update_on(mm, nm, dm, mpopt)`

mp.mme_xfmr3p

class `mp.mme_xfmr3p`

Bases: `mp.mm_element` (page 146)

`mp.mme_xfmr3p` (page 201) - Math model element for 3-phase transformer.

Math model element base class for 3-phase transformer elements.

Implements method for updating the output data in the corresponding data model element for in-service 3-phase transformers from the math model solution.

Method Summary

name()

data_model_update_on(*mm, nm, dm, mpopt*)

mp.mme_buslink

class mp.mme_buslink

Bases: [mp.mm_element](#) (page 146)

[mp.mme_buslink](#) (page 202) - Math model element abstract base class for 1-to-3-phase buslink.

Abstract math model element base class for 1-to-3-phase buslink elements.

Method Summary

name()

mp.mme_buslink_pf_ac

class mp.mme_buslink_pf_ac

Bases: [mp.mme_buslink](#) (page 202)

[mp.mme_buslink_pf_ac](#) (page 202) - Math model element abstract base class for 1-to-3-phase buslink for AC PF/CPF.

Abstract math model element base class for 1-to-3-phase buslink elements for AC power flow and CPF problems.

Implements methods for adding per-phase active and reactive power variables and for forming and adding voltage and reactive power constraints.

Method Summary

add_vars(*mm, nm, dm, mpopt*)

add_constraints(*mm, nm, dm, mpopt*)

voltage_constraints(*nme, ad*)

mp.mme_buslink_pf_acc

class mp.mme_buslink_pf_acc

Bases: [mp.mme_buslink_pf_ac](#) (page 202)

[mp.mme_buslink_pf_acc](#) (page 202) - Math model element for 1-to-3-phase buslink for AC cartesian voltage PF/CPF.

Math model element class for 1-to-3-phase buslink elements for AC cartesian power flow and CPF problems.

Implements methods for adding constraints to match voltages across each buslink.

Method Summary

add_constraints(*mm, nm, dm, mpop*)

pf_va_fcn(*nme, xx, A, b*)

pf_vm_fcn(*nme, xx, A, b*)

mp.mme_buslink_pf_acp

class mp.mme_buslink_pf_acp

Bases: [mp.mme_buslink_pf_ac](#) (page 202)

[mp.mme_buslink_pf_acp](#) (page 203) - Math model element for 1-to-3-phase buslink for AC polar voltage PF/CPF.

Math model element class for 1-to-3-phase buslink elements for AC polar power flow and CPF problems.

Implements method for adding constraints to match voltages across each buslink.

Method Summary

add_constraints(*mm, nm, dm, mpop*)

mp.mme_bus3p_opf_acc

class mp.mme_bus3p_opf_acc

Bases: [mp.mme_bus3p](#) (page 200)

[mp.mme_bus3p_opf_acc](#) (page 203) - Math model element for 3-phase bus for AC cartesian voltage OPF.

Math model element class for 3-phase bus elements for AC cartesian voltage OPF problems.

Implements method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_bus3p_opf_acp

class mp.mme_bus3p_opf_acp

Bases: [mp.mme_bus3p](#) (page 200)

[mp.mme_bus3p_opf_acp](#) (page 203) - Math model element for 3-phase bus for AC polar voltage OPF.

Math model element class for 3-phase bus elements for AC polar voltage OPF problems.

Implements method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_gen3p_opf

class mp.mme_gen3p_opf

Bases: [mp.mme_gen3p](#) (page 201)

[mp.mme_gen3p_opf](#) (page 204) - Math model element for 3-phase generator for OPF.

Math model element class for 1-to-3-phase generator elements for OPF problems.

Implements (currently empty) method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_line3p_opf

class mp.mme_line3p_opf

Bases: [mp.mme_line3p](#) (page 201)

[mp.mme_line3p_opf](#) (page 204) - Math model element for 3-phase line for OPF.

Math model element class for 3-phase line elements for OPF problems.

Implements (currently empty) method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_xfmr3p_opf

class mp.mme_xfmr3p_opf

Bases: [mp.mme_xfmr3p](#) (page 201)

[mp.mme_xfmr3p_opf](#) (page 204) - Math model element for 3-phase transformer for OPF.

Math model element class for 3-phase transformer elements for OPF problems.

Implements (currently empty) method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_buslink_opf**class mp.mme_buslink_opf**

Bases: [mp.mme_buslink](#) (page 202)

[mp.mme_buslink_opf](#) (page 205) - Math model element abstract base class for 1-to-3-phase buslink for OPF.

Abstract math model element base class for 1-to-3-phase buslink elements for OPF problems.

Implements (currently empty) method for forming an interior initial point.

Method Summary

interior_x0(*mm, nm, dm, x0*)

mp.mme_buslink_opf_acc**class mp.mme_buslink_opf_acc**

Bases: [mp.mme_buslink_opf](#) (page 205)

[mp.mme_buslink_opf_acc](#) (page 205) - Math model element for 1-to-3-phase buslink for AC cartesian voltage OPF.

Math model element class for 1-to-3-phase buslink elements for AC cartesian OPF problems.

Implements methods for adding constraints to match voltages across each buslink.

Method Summary

add_constraints(*mm, nm, dm, mpop*)

va_fcn(*nme, xx, A, b*)

va_hess(*nme, xx, lam, A*)

vm2_fcn(*nme, xx, A, b*)

vm2_hess(*nme, xx, lam, A*)

mp.mme_buslink_opf_acp**class mp.mme_buslink_opf_acp**

Bases: [mp.mme_buslink_opf](#) (page 205)

[mp.mme_buslink_opf_acp](#) (page 205) - Math model element for 1-to-3-phase buslink for AC polar voltage OPF.

Math model element class for 1-to-3-phase buslink elements for AC polar OPF problems.

Implements method for adding constraints to match voltages across each buslink.

Method Summary

add_constraints(*mm, nm, dm, mpop*)

3.7.4 Legacy DC Line Extension

For more details, see `howto_element`.

`mp.xt_legacy_dcline`

`class mp.xt_legacy_dcline`

Bases: `mp.extension` (page 174)

`mp.xt_legacy_dcline` (page 206) - MATPOWER extension to add legacy DC line elements.

For AC and DC power flow, continuation power flow, and optimal power flow problems, adds a new element type:

- 'legacy_dcline' - legacy DC line

No changes are required for the task or container classes, so only the `..._element_classes` methods are overridden.

The set of data model element classes depends on the task, with each OPF class inheriting from the corresponding class used for PF and CPF.

The set of network model element classes depends on the formulation, specifically whether cartesian or polar representations are used for voltages.

And the set of mathematical model element classes depends on both the task and the formulation.

`mp.xt_legacy_dcline` Methods:

- `dmc_element_classes()` (page 206) - add a class to data model converter elements
- `dm_element_classes()` (page 206) - add a class to data model elements
- `nm_element_classes()` (page 206) - add a class to network model elements
- `mm_element_classes()` (page 207) - add a class to mathematical model elements

See the `sec_customizing` and `sec_extensions` sections in the *MATPOWER Developer's Manual* for more information, and specifically the `sec_element_classes` section and the `tab_element_class_modifiers` table for details on *element class modifiers*.

See also `mp.extension` (page 174).

Method Summary

`dmc_element_classes(dmc_class, fmt, mpopt)`

Add a class to data model converter elements.

For 'mpc2' data formats, adds the classes:

- `mp.dmce_legacy_dcline_mpc2` (page 207)

`dm_element_classes(dm_class, task_tag, mpopt)`

Add a class to data model elements.

For 'PF' and 'CPF' tasks, adds the class:

- `mp.dme_legacy_dcline` (page 208)

For 'OPF' tasks, adds the class:

- `mp.dme_legacy_dcline_opf` (page 210)

nm_element_classes(*nm_class*, *task_tag*, *mpopt*)

Add a class to network model elements.

For DC formulations, adds the class:

- [mp.nme_legacy_dcline_dc](#) (page 212)

For AC *cartesian* voltage formulations, adds the class:

- [mp.nme_legacy_dcline_acc](#) (page 212)

For AC *polar* voltage formulations, adds the class:

- [mp.nme_legacy_dcline_acp](#) (page 212)

mm_element_classes(*mm_class*, *task_tag*, *mpopt*)

Add a class to mathematical model elements.

For 'PF' and 'CPF' tasks, adds the class:

- [mp.mme_legacy_dcline_pf_dc](#) (page 213) (*DC formulation*) or
- [mp.mme_legacy_dcline_pf_ac](#) (page 213) (*AC formulation*)

For 'OPF' tasks, adds the class:

- [mp.mme_legacy_dcline_opf_dc](#) (page 214) (*DC formulation*) or
- [mp.mme_legacy_dcline_opf_ac](#) (page 214) (*AC formulation*)

Data model converter element class belonging to [mp.xt_legacy_dcline](#) (page 206) extension:

mp.dmce_legacy_dcline_mpc2

class `mp.dmce_legacy_dcline_mpc2`

Bases: [mp.dmc_element](#) (page 65)

[mp.dmce_legacy_dcline_mpc2](#) (page 207) - Data model converter element for legacy DC line for MATPOWER case v2.

Method Summary

name()

data_field()

table_var_map(*dme*, *mpc*)

default_export_data_table(*spec*)

dcline_cost_import(*mpc*, *spec*, *vn*)

dcline_cost_export(*dme*, *mpc*, *spec*, *vn*, *ridx*)

Data model element classes belonging to [mp.xt_legacy_dcline](#) (page 206) extension:

mp.dme_legacy_dcline**class mp.dme_legacy_dcline**Bases: [mp.dm_element](#) (page 38)[mp.dme_legacy_dcline](#) (page 208) - Data model element for legacy DC line.

Implements the data element model for legacy DC line elements, with linear line losses.

$$p_{\text{loss}} = l_0 + l_1 p_{\text{fr}}$$

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>bus_fr</code>	<i>integer</i>	bus ID (uid) of “from” bus
<code>bus_to</code>	<i>integer</i>	bus ID (uid) of “to” bus
<code>loss0</code>	<i>double</i>	l_0 , constant term of loss function (MW)
<code>loss1</code>	<i>double</i>	l_1 , linear coefficient of loss function (MW/MW)
<code>vm_setpoint_fr</code>	<i>double</i>	per unit “from” bus voltage magnitude setpoint
<code>vm_setpoint_to</code>	<i>double</i>	per unit “to” bus voltage magnitude setpoint
<code>p_fr_lb</code>	<i>double</i>	lower bound on MW flow at “from” port
<code>p_fr_ub</code>	<i>double</i>	upper bound on MW flow at “from” port
<code>q_fr_lb</code>	<i>double</i>	lower bound on MVar injection into “from” bus
<code>q_fr_ub</code>	<i>double</i>	upper bound on MVar injection into “from” bus
<code>q_to_lb</code>	<i>double</i>	lower bound on MVar injection into “to” bus
<code>q_to_ub</code>	<i>double</i>	upper bound on MVar injection into “to” bus
<code>p_fr</code>	<i>double</i>	MW flow at “from” end (“from” → “to”)
<code>q_fr</code>	<i>double</i>	MVar injection into “from” bus
<code>p_to</code>	<i>double</i>	MW flow at “to” end (“from” → “to”)
<code>q_to</code>	<i>double</i>	MVar injection into “to” bus

Property Summary**fbus**

bus index vector for “from” port (port 1) (all DC lines)

tbus

bus index vector for “to” port (port 2) (all DC lines)

fbus_on

vector of “from” bus indices into online buses (in-service DC lines)

tbus_on

vector of “to” bus indices into online buses (in-service DC lines)

loss0

constant term of loss function (p.u.) (in-service DC lines)

loss1

linear coefficient of loss function (in-service DC lines)

p_fr_start

initial active power (p.u.) at “from” port (in-service DC lines)

p_to_start

initial active power (p.u.) at “to” port (in-service DC lines)

q_fr_start

initial reactive power (p.u.) at “from” port (in-service DC lines)

q_to_start

initial reactive power (p.u.) at “to” port (in-service DC lines)

vm_setpoint_fr

from bus voltage magnitude setpoint (p.u.) (in-service DC lines)

vm_setpoint_to

to bus voltage magnitude setpoint (p.u.) (in-service DC lines)

p_fr_lb

p.u. lower bound on active power flow at “from” port (in-service DC lines)

p_fr_ub

p.u. upper bound on active power flow at “from” port (in-service DC lines)

q_fr_lb

p.u. lower bound on reactive power flow at “from” port (in-service DC lines)

q_fr_ub

p.u. upper bound on reactive power flow at “from” port (in-service DC lines)

q_to_lb

p.u. lower bound on reactive power flow at “to” port (in-service DC lines)

q_to_ub

p.u. upper bound on reactive power flow at “to” port (in-service DC lines)

Method Summary

name()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

export_vars()

export_vars_offline_val()

have_cost()

initialize(*dm*)

update_status(*dm*)

apply_vm_setpoints(*dm*)

build_params(*dm*)

pp_have_section_sum(*mpopt*, *pp_args*)

```

pp_data_sum(dm, rows, out_e, mpopt, fd, pp_args)

pp_get_headers_det(dm, out_e, mpopt, pp_args)

pp_have_section_det(mpop, pp_args)

pp_data_row_det(dm, k, out_e, mpopt, fd, pp_args)

```

mp.dme_legacy_dcline_opf

class mp.dme_legacy_dcline_opf

Bases: [mp.dme_legacy_dcline](#) (page 208), [mp.dme_shared_opf](#) (page 61)

[mp.dme_legacy_dcline_opf](#) (page 210) - Data model element for legacy DC line for OPF.

To parent class [mp.dme_legacy_dcline](#) (page 208), adds costs, shadow prices on active and reactive flow limits, and pretty-printing for **lim** sections.

Adds the following columns in the main data table, found in the `tab` property:

Name	Type	Description
<code>cost_pg</code>	<code>mp.cost_table</code>	cost of active power flow (u/MW) ¹
<code>mu_p_fr_lb</code>	<code>double</code>	shadow price on MW flow lower bound at “from” end (u/MW) ¹
<code>mu_p_fr_ub</code>	<code>double</code>	shadow price on MW flow upper bound at “from” end (u/MW) ¹
<code>mu_q_fr_lb</code>	<code>double</code>	shadow price on lower bound of MVar injection at “from” bus ($u/degree$) ¹
<code>mu_q_fr_ub</code>	<code>double</code>	shadow price on upper bound of MVar injection at “from” bus ($u/degree$) ¹
<code>mu_q_to_lb</code>	<code>double</code>	shadow price on lower bound of MVar injection at “to” bus ($u/degree$) ¹
<code>mu_q_to_ub</code>	<code>double</code>	shadow price on upper bound of MVar injection at “to” bus ($u/degree$) ¹

Method Summary

```

main_table_var_names()

export_vars()

export_vars_offline_val()

have_cost()

build_cost_params(dm)

pretty_print(dm, section, out_e, mpopt, fd, pp_args)

pp_have_section_lim(mpop, pp_args)

pp_binding_rows_lim(dm, out_e, mpopt, pp_args)

```

¹ Here u denotes the units of the objective function, e.g. USD.

```
pp_get_headers_lim(dm, out_e, mpop, pp_args)
pp_data_row_lim(dm, k, out_e, mpop, fd, pp_args)
```

Network model element classes belonging to [mp.xt_legacy_dcline](#) (page 206) extension:

mp.nme_legacy_dcline

class mp.nme_legacy_dcline

Bases: [mp.nm_element](#) (page 110)

[mp.nme_legacy_dcline](#) (page 211) - Network model element abstract base class for legacy DC line.

Implements the network model element for legacy DC line elements, with 2 ports and 2 non-voltage states per DC line.

Method Summary

name()

np()

nz()

mp.nme_legacy_dcline_ac

class mp.nme_legacy_dcline_ac

Bases: [mp.nme_legacy_dcline](#) (page 211)

[mp.nme_legacy_dcline_ac](#) (page 211) - Network model element abstract base class for legacy DC line for AC formulation.

Adds non-voltage state variables Pdcf, Qdcf, Pdct, and Qdct to the network model and builds the parameter N.

Method Summary

add_zvars(nm, dm, idx)

build_params(nm, dm)

mp.nme_legacy_dcline_acc

class mp.nme_legacy_dcline_acc

Bases: [mp.nme_legacy_dcline_ac](#) (page 211), [mp.form_acc](#) (page 85)

[mp.nme_legacy_dcline_acc](#) (page 212) - Network model element for legacy DC line for AC cartesian voltage formulations.

Inherits from [mp.form_acc](#) (page 85).

mp.nme_legacy_dcline_acp

class mp.nme_legacy_dcline_acp

Bases: [mp.nme_legacy_dcline_ac](#) (page 211), [mp.form_acp](#) (page 89)

[mp.nme_legacy_dcline_acp](#) (page 212) - Network model element for legacy DC line for AC polar voltage formulations.

Inherits from [mp.form_acp](#) (page 89).

mp.nme_legacy_dcline_dc

class mp.nme_legacy_dcline_dc

Bases: [mp.nme_legacy_dcline](#) (page 211), [mp.form_dc](#) (page 91)

[mp.nme_legacy_dcline_dc](#) (page 212) - Network model element for legacy DC line for DC formulation.

Adds non-voltage state variables Pdcf and Pdct to the network model and builds the parameter **K**.

Method Summary

add_zvars(nm, dm, idx)

build_params(nm, dm)

Mathematical model element classes belonging to [mp.xt_legacy_dcline](#) (page 206) extension:

mp.mme_legacy_dcline

class mp.mme_legacy_dcline

Bases: [mp.mm_element](#) (page 146)

[mp.mme_legacy_dcline](#) (page 212) - Math model element abstract base class for legacy DC line.

Abstract math model element base class for legacy DC line elements.

Method Summary

name()

mp.mme_legacy_dcline_pf_ac**class** mp.mme_legacy_dcline_pf_acBases: [mp.mme_legacy_dcline](#) (page 212)[mp.mme_legacy_dcline_pf_ac](#) (page 213) - Math model element for legacy DC line for AC power flow.

Math model element class for legacy DC line elements for AC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service DC lines from the math model solution.

Method Summary**data_model_update_on**(mm, nm, dm, mpopt)**mp.mme_legacy_dcline_pf_dc****class** mp.mme_legacy_dcline_pf_dcBases: [mp.mme_legacy_dcline](#) (page 212)[mp.mme_legacy_dcline_pf_dc](#) (page 213) - Math model element for legacy DC line for DC power flow.

Math model element class for legacy DC line elements for DC power flow problems.

Implements method for updating the output data in the corresponding data model element for in-service DC lines from the math model solution.

Method Summary**data_model_update_on**(mm, nm, dm, mpopt)**mp.mme_legacy_dcline_opf****class** mp.mme_legacy_dcline_opfBases: [mp.mme_legacy_dcline](#) (page 212)[mp.mme_legacy_dcline_opf](#) (page 213) - Math model element abstract base class for legacy DC line for OPF.

Math model element abstract base class for legacy DC line elements for OPF problems.

Implements methods to add costs, including piecewise linear cost variables, and to form an interior initial point for cost variables.

Property Summary**cost**struct for [cost](#) (page 213) parameters with fields:

- poly - polynomial costs for active power, struct with fields:
 - have_quad_cost
 - i0, i1, i2, i3
 - k, c, Q

- `pwl` - piecewise linear costs for active power, struct with fields:
 - `n`, `i`, `A`, `b`

Method Summary

`build_cost_params(dm)`

`add_vars(mm, nm, dm, mpopt)`

`add_constraints(mm, nm, dm, mpopt)`

`add_costs(mm, nm, dm, mpopt)`

`interior_x0(mm, nm, dm, x0)`

`mp.mme_legacy_dcline_opf_ac`

`class mp.mme_legacy_dcline_opf_ac`

Bases: [`mp.mme_legacy_dcline_opf`](#) (page 213)

[`mp.mme_legacy_dcline_opf_ac`](#) (page 214) - Math model element for legacy DC line for AC OPF.

Math model element class for legacy DC line elements for AC OPF problems.

Implements method for updating the output data in the corresponding data model element for in-service DC lines from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

`mp.mme_legacy_dcline_opf_dc`

`class mp.mme_legacy_dcline_opf_dc`

Bases: [`mp.mme_legacy_dcline_opf`](#) (page 213)

[`mp.mme_legacy_dcline_opf_dc`](#) (page 214) - Math model element for legacy DC line for DC OPF.

Math model element class for legacy DC line elements for DC OPF problems.

Implements method for updating the output data in the corresponding data model element for in-service DC lines from the math model solution.

Method Summary

`data_model_update_on(mm, nm, dm, mpopt)`

3.7.5 Example User Constraint Extension

For more details, see `howto_add_constraint`.

`mp.xt_oval_cap_curve`

class `mp.xt_oval_cap_curve`

Bases: `mp.extension` (page 174)

`mp.xt_oval_cap_curve` (page 215) - MATPOWER extension for OPF with oval gen PQ capability curves.

For OPF problems, this extension restricts the output of each generator to lie within the half-oval-shaped region centered at (P_{MIN}, Q₀) and passing through (P_{MAX}, Q₀), (P_{MIN}, Q_{MIN}) and (P_{MIN}, Q_{MAX}).

`mp.xt_oval_cap_curve` Methods:

- `mm_element_classes()` (page 215) - replace a class in mathematical model elements

See the `sec_customizing` and `sec_extensions` sections in the *MATPOWER Developer's Manual* for more information, and specifically the `sec_element_classes` section and the `tab_element_class_modifiers` table for details on *element class modifiers*.

See also `mp.extension` (page 174), `mp.mme_gen_opf_ac_oval` (page 215).

Method Summary

`mm_element_classes`(*mm_class*, *task_tag*, *mpopt*)

Replace a class in mathematical model elements.

For 'OPF' tasks, replaces `mp.gen_opf_ac` with `mp.gen_opf_ac_oval`.

Mathematical model element class belonging to `mp.xt_oval_cap_curve` (page 215) extension:

`mp.mme_gen_opf_ac_oval`

class `mp.mme_gen_opf_ac_oval`

Bases: `mp.mme_gen_opf_ac` (page 156)

`mp.mme_gen_opf_ac_oval` (page 215) - Math model element for generator for AC OPF w/oval cap curve.

Math model element class for generator elements for AC OPF problems, implementing an oval, as opposed to rectangular, PQ capability curve.

Method Summary

`add_constraints`(*mm*, *nm*, *dm*, *mpopt*)

Set up the nonlinear constraint for gen oval PQ capability curves.

```
mme.add_constraints(mm, nm, dm, mpopt)
```

`oval_pq_capability_fcn`(*xx*, *idx*, *p0*, *q0*, *a2*, *b2*)

Compute oval PQ capability constraints and Jacobian.

```
h = mme.oval_pq_capability_fcn(xx, idx, p0, q0, a2, b2)
[h, dh] = mme.oval_pq_capability_fcn(xx, idx, p0, q0, a2, b2)
```

Compute constraint function and optionally the Jacobian for oval PQ capability limits.

Inputs

- **xx** (*1 x 2 cell array*) – active power injection in **xx{1}**, reactive injection in **xx{2}**
- **idx** (*integer*) – index of subset of generators of interest to include in constraint; if empty, include all
- **p0** (*double*) – vector of horizontal (p) centers
- **q0** (*double*) – vector of vertical (q) centers
- **a2** (*double*) – vector of squares of horizontal (p) radii
- **b2** (*double*) – vector of squares of vertical (q) radii

Outputs

- **h** (*double*) – constraint function, $h(x)$
- **dh** (*double*) – constraint Jacobian, h_x

Note that the oval specs **p0**, **q0**, **a2**, **b2** are assumed to have dimension corresponding to **idx**.

oval_pq_capability_hess(xx, lam, idx, p0, q0, a2, b2)

Compute oval PQ capability constraint Hessian.

```
d2H = mme.oval_pq_capability_hess(xx, lam, idx, p0, q0, a2, b2)
```

Compute a sparse Hessian matrix for oval PQ capability limits. Rather than a full, 3-dimensional Hessian, it computes the Jacobian of the vector obtained by multiplying the transpose of the constraint Jacobian by a vector μ .

Inputs

- **xx** (*1 x 2 cell array*) – active power injection in **xx{1}**, reactive injection in **xx{2}**
- **lam** (*double*) – vector μ of multipliers
- **idx** (*integer*) – index of subset of generators of interest to include in constraint; if empty, include all
- **p0** (*double*) – vector of horizontal (p) centers
- **q0** (*double*) – vector of vertical (q) centers
- **a2** (*double*) – vector of squares of horizontal (p) radii
- **b2** (*double*) – vector of squares of vertical (q) radii

Output

d2H (*double*) – sparse constraint Hessian matrix

Note that the oval specs **p0**, **q0**, **a2**, **b2** are assumed to have dimension corresponding to **idx**.

4.1 MATPOWER Tests

4.1.1 test_matpower

test_matpower(*verbose*, *exit_on_fail*)

[test_matpower\(\)](#) (page 217) - Run all MATPOWER tests.

```
test_matpower
test_matpower(verbose)
test_matpower(verbose, exit_on_fail)
success = test_matpower(...)
```

Runs all of the MATPOWER tests. If *verbose* is true (*false by default*), it prints the details of the individual tests. If *exit_on_fail* is true (*false by default*), it will exit MATLAB or Octave with a status of 1 unless `t_run_tests()` returns `all_ok` true.

See also `t_run_tests()`.

4.1.2 t_mp_mapped_array

t_mp_mapped_array(*quiet*)

[t_mp_mapped_array\(\)](#) (page 217) - Tests for `mp.mapped_array` (page 169).

4.1.3 t_mp_table

t_mp_table(*quiet*)

t_mp_table() (page 218) - Tests for *mp_table* (page 159) (and *table*).

4.1.4 t_mp_data_model

t_mp_data_model(*quiet*)

t_mp_data_model() (page 218) - Tests for *mp.data_model* (page 30).

4.1.5 t_dmc_element

t_dmc_element(*quiet*)

t_dmc_element() (page 218) - Tests for *mp.dmc_element* (page 65).

4.1.6 t_mp_dm_converter_mpc2

t_mp_dm_converter_mpc2(*quiet*)

t_mp_dm_converter_mpc2() (page 218) - Tests for *mp.dm_converter_mpc2* (page 64).

4.1.7 t_nm_element

t_nm_element(*quiet*, *out_ac*)

t_nm_element() (page 218) - Tests for *mp.nm_element* (page 110).

4.1.8 t_port_inj_current_acc

t_port_inj_current_acc(*quiet*)

t_port_inj_current_acc() (page 218) - Tests of *mp.form_ac.port_inj_current()* (page 79) derivatives wrt cartesian V.

4.1.9 t_port_inj_current_acp

t_port_inj_current_acp(*quiet*)

[t_port_inj_current_acp\(\)](#) (page 219) - Tests of *mp.form_ac.port_inj_current()* (page 79) derivatives wrt polar V.

4.1.10 t_port_inj_power_acc

t_port_inj_power_acc(*quiet*)

[t_port_inj_power_acc\(\)](#) (page 219) - Tests of *mp.form_ac.port_inj_power()* (page 79) derivatives wrt cartesian V.

4.1.11 t_port_inj_power_acp

t_port_inj_power_acp(*quiet*)

[t_port_inj_power_acp\(\)](#) (page 219) - Tests of *mp.form_ac.port_inj_power()* (page 79) derivatives wrt polar V.

4.1.12 t_node_test

t_node_test(*quiet*)

[t_node_test\(\)](#) (page 219) - Tests for network model with multiple node-creating elements.

4.1.13 t_run_mp

t_run_mp(*quiet*)

[t_run_mp\(\)](#) (page 219) - Tests for [run_mp\(\)](#) (page 4) and simple creation and solve of models.

4.1.14 t_run_mp_3p

t_run_mp_3p(*quiet*)

[t_run_mp_3p\(\)](#) (page 219) - Tests for [run_pf\(\)](#) (page 5), [run_cpf\(\)](#) (page 5), [run_opf\(\)](#) (page 6) for 3-phase and hybrid test cases.

4.1.15 `t_run_opf_default`

`t_run_opf_default(quiet)`

`t_run_opf_default()` (page 220) - Tests for AC optimal power flow using `run_opf()` (page 6) w/default solver.

4.1.16 `t_pretty_print`

`t_pretty_print(quiet)`

`t_pretty_print()` (page 220) - Tests for pretty printed output.

4.1.17 `t_mpxt_legacy_dcline`

`t_mpxt_legacy_dcline(quiet)`

`t_mpxt_legacy_dcline()` (page 220) - Tests for legacy DC line extension in `mp.xt_legacy_dcline` (page 206).

4.1.18 `t_mpxt_reserves`

`t_mpxt_reserves(quiet)`

`t_mpxt_reserves()` (page 220) - Tests `mp.xt_reserves` (page 177) extension.

4.2 MATPOWER Test Data

4.2.1 `mp_foo_table`

`class mp_foo_table`

Bases: `mp_table_subclass` (page 163)

`mp_foo_table` (page 220) - Subclass of `mp_table_subclass` (page 163) for testing.

4.2.2 `t_case3p_a`

`t_case3p_a()`

`t_case3p_a()` (page 221) - Four bus, unbalanced 3-phase test case.

This data comes from `4Bus-YY-UnB.DSS`, a modified version (with unbalanced load) of `4Bus-YY-Bal.DSS` [1], the OpenDSS 4 bus IEEE test case with grounded-wye to grounded-wye transformer.

[1] <https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Distrib/IEEETestCases/4Bus-YY-Bal/4Bus-YY-Bal.DSS>

4.2.3 `t_case3p_b`

`t_case3p_b()`

`t_case3p_b()` (page 221) - Six bus hybrid test case, 2 single-phase buses, 4 3-phase buses.

One bus is a hybrid PQ bus. Three phase bus solution should match `t_case3p_a()` (page 221).

4.2.4 `t_case3p_c`

`t_case3p_c()`

`t_case3p_c()` (page 221) - Six bus hybrid test case, 2 single-phase buses, 4 3-phase buses.

One bus is a hybrid PV bus (PV on single-phase side). Three phase bus solution should match `t_case3p_a()` (page 221).

4.2.5 `t_case3p_d`

`t_case3p_d()`

`t_case3p_d()` (page 221) - Six bus hybrid test case, 2 single-phase buses, 4 3-phase buses.

One bus is a hybrid PV bus (PV on 3-phase side). Three phase bus solution should match `t_case3p_a()` (page 221).

4.2.6 `t_case3p_e`

`t_case3p_e()`

`t_case3p_e()` (page 221) - Five bus hybrid test case, 1 single-phase bus, 4 3-phase buses.

One bus is a hybrid REF bus (REF on single-phase side). Three phase bus solution should match `t_case3p_a()` (page 221).

4.2.7 `t_case3p_f`

`t_case3p_f()`

`t_case3p_f()` (page 222) - 21 bus hybrid test case, 9 single-phase buses, 12 3-phase buses.

Three buses are hybrid PQ buses.

4.2.8 `t_case3p_g`

`t_case3p_g()`

`t_case3p_g()` (page 222) - 21 bus hybrid test case, 9 single-phase buses, 12 3-phase buses.

Three buses are hybrid buses, one REF-PQ, one PV-PQ and the other PQ-PQ. Solutions of three-phase portions should match `t_case3p_a()` (page 221).

4.2.9 `t_case3p_h`

`t_case3p_h()`

`t_case3p_h()` (page 222) - 21 bus hybrid test case, 9 single-phase buses, 12 3-phase buses.

Same as `t_case3p_g()` (page 222), except the PV hybrid bus has the PV on the 3-phase side. Three buses are hybrid buses, one REF-PQ, one PQ-PV and the other PQ-PQ. Solutions of three-phase portions should match `t_case3p_a()` (page 221).

4.2.10 `t_case9_gizmo`

`t_case9_gizmo()`

`t_case9_gizmo()` (page 222) - Power flow data for 9 bus, 3 generator case, with gizmo data.

Please see caseformat for details on the case file format.

This section contains reference documentation for the **legacy MATPOWER framework** (see `sec_two_frameworks` in the *MATPOWER Developer's Manual*) and the rest of the legacy codebase inherited from MATPOWER 7 and earlier.

5.1 Legacy Class

5.1.1 `opf_model`

class `opf_model`

Bases: `opt_model`

[*opf_model*](#) (page 223) - Legacy MATPOWER OPF model class.

```
OM = OPF_MODEL(MPC)
```

This **class** implements the OPF model object used to encapsulate a given OPF problem formulation. It allows **for** access to optimization variables, constraints **and** costs in named blocks, keeping track of the ordering **and** indexing of the blocks as variables, constraints **and** costs are added to the problem.

This **class** is a subclass of `OPT_MODEL` that adds the `'mpc'` field **for** storing the MATPOWER **case struct** used to build the object along with the `get_mpc()` method.

It also adds the `'cost'` field **and** the following three **methods for** implementing the legacy user-defined OPF costs:

```
add_legacy_cost
params_legacy_cost
eval_legacy_cost
```

The following is the structure of the data in the OPF model object.

(continues on next page)

(continued from previous page)

```

om
<opt_model fields> - see OPT_MODEL for details
.cost              - data for legacy user-defined costs
  .idx
    .i1 - starting row index within full N matrix
    .iN - ending row index within full N matrix
    .N  - number of rows in this cost block in full N matrix
  .N          - total number of rows in full N matrix
  .NS         - number of cost blocks
  .data       - data for each user-defined cost block
    .N        - see help for ADD_LEGACY_COST for details
    .H        -
    .Cw       -
    .dd       -
    .rr       -
    .kk       -
    .nm       -
    .vs       - cell array of variable sets that define xx for this
                  cost block, where the N for this block multiplies xx
  .order       - struct array of names/indices for cost blocks in the
                  order they appear in the rows of the full N matrix
    .name      - name of the block, e.g. R
    .idx       - indices for name, {2,3} => R(2,3)
.mpc           - MATPOWER case struct used to create this model object
  .baseMVA
  .bus
  .branch
  .gen
  .gencost
  .A (if present, must have l, u)
  .l
  .u
  .N (if present, must have fparm, H, Cw)
  .fparm
  .H
  .Cw

```

See also `opt_model`.

Constructor Summary

opf_model(*mpc*)

Constructor.

```

om = opf_model()
om = opf_model(mpc)

```

Property Summary

cost = []

data for legacy user-defined costs

mpc = struct()

MATPOWER case struct from which om was built

Method Summary

def_set_types(om)

Define set types var, lin, nle, nli, qdc, nlc, cost.

init_set_types(om)

Initialize data structures for each set type.

set_mpc(om, mpc)

[set_mpc\(\)](#) (page 225) - Sets the MATPOWER case struct.

```
OM.SET_MPC(MPC)
```

See also [opt_model](#).

display(om)

[display\(\)](#) (page 225) - Displays the object.

Called when semicolon is omitted at the command-line. Displays the details of the variables, constraints, costs included in the model.

See also [opt_model](#).

get_mpc(om)

[get_mpc\(\)](#) (page 225) - Returns the MATPOWER case struct.

```
MPC = OM.GET_MPC()
```

See also [opt_model](#).

eval_legacy_cost(om, x, name, idx)

[eval_legacy_cost\(\)](#) (page 225) - Evaluates individual or full set of legacy user costs.

```
F = OM.EVAL_LEGACY_COST(X ...)
[F, DF] = OM.EVAL_LEGACY_COST(X ...)
[F, DF, D2F] = OM.EVAL_LEGACY_COST(X ...)
[F, DF, D2F] = OM.EVAL_LEGACY_COST(X, NAME)
[F, DF, D2F] = OM.EVAL_LEGACY_COST(X, NAME, IDX_LIST)
Evaluates an individual named set or the full set of legacy user
costs and their derivatives for a given value of the optimization vector
X, based on costs added by ADD_LEGACY_COST.
```

Example:

```
[f, df, d2f] = om.eval_legacy_cost(x)
[f, df, d2f] = om.eval_legacy_cost(x, name)
[f, df, d2f] = om.eval_legacy_cost(x, name, idx)
```

See also [opt_model](#), [add_legacy_cost\(\)](#) (page 227), [params_legacy_cost\(\)](#) (page 225).

params_legacy_cost(om, name, idx)

[params_legacy_cost\(\)](#) (page 225) - Returns cost parameters for legacy user-defined costs.

```

CP = OM.PARAMS_LEGACY_COST()
CP = OM.PARAMS_LEGACY_COST(NAME)
CP = OM.PARAMS_LEGACY_COST(NAME, IDX_LIST)
[CP, VS] = OM.PARAMS_LEGACY_COST(...)
[CP, VS, I1, IN] = OM.PARAMS_LEGACY_COST(...)

```

With no **input** parameters, it assembles and returns the parameters **for** the aggregate legacy user-defined cost from **all** legacy cost sets added using ADD_LEGACY_COST. The values of these parameters are cached **for** subsequent calls. The parameters are contained in the **struct** CP, described below.

If a NAME is provided then it simply returns parameter **struct** CP **for** the corresponding named **set**. Likewise **for** indexed named sets specified by NAME **and** IDX_LIST.

An optional 2nd output argument VS indicates the variable sets used by this cost **set**. The **size** of CP.N will be consistent with VS.

If NAME is provided, optional 3rd **and** 4th output arguments I1 **and** IN indicate the starting **and** ending row indices of the corresponding cost **set** in the **full** aggregate cost matrix CP.N.

Let X refer to the vector formed by combining the corresponding varsets VS, **and** F_U(X, CP) be the cost at X corresponding to the cost parameters contained in CP, where CP is a **struct** with the following fields:

```

N      - nw x nx sparse matrix (optional, identity matrix by default)
Cw     - nw x 1 vector
H      - nw x nw sparse matrix (optional, all zeros by default)
dd, mm - nw x 1 vectors (optional, all ones by default)
rh, kk - nw x 1 vectors (optional, all zeros by default)

```

These parameters are used as follows to compute F_U(X, CP)

```

R = N*X - rh

      /  kk(i),  R(i) < -kk(i)
K(i) = <  0,    -kk(i) <= R(i) <= kk(i)
      \ -kk(i), R(i) > kk(i)

RR = R + K

U(i) = /  0, -kk(i) <= R(i) <= kk(i)
      \  1, otherwise

DDL(i) = /  1, dd(i) = 1
        \  0, otherwise

DDQ(i) = /  1, dd(i) = 2
        \  0, otherwise

Dl = diag(mm) * diag(U) * diag(DDL)
Dq = diag(mm) * diag(U) * diag(DDQ)

```

(continues on next page)

(continued from previous page)

$$w = (Dl + Dq * \text{diag}(RR)) * RR$$

$$F_U(X, CP) = 1/2 * w' * H * w + Cw' * w$$

See also `opt_model`, `add_legacy_cost()` (page 227), `eval_legacy_cost()` (page 225).

add_named_set(*om, set_type, name, idx, N, varargin*)

`add_named_set()` (page 227) - Adds a named set of variables/constraints/costs to the model.

```
----- PRIVATE METHOD -----

This method is intended to be a private method, used internally by
the public methods ADD_VAR, ADD_LIN_CONSTRAINT, ADD_NLN_CONSTRAINT
ADD_QUAD_COST, ADD_NLN_COST and ADD_LEGACY_COST.

Variable Set
    OM.ADD_NAMED_SET('var', NAME, IDX_LIST, N, V0, VL, VU, VT);

Linear Constraint Set
    OM.ADD_NAMED_SET('lin', NAME, IDX_LIST, N, A, L, U, VARSETS);

Nonlinear Inequality Constraint Set
    OM.ADD_NAMED_SET('nle', NAME, IDX_LIST, N, FCN, HESS, COMPUTED_BY,
    ↪VARSETS);

Nonlinear Inequality Constraint Set
    OM.ADD_NAMED_SET('nli', NAME, IDX_LIST, N, FCN, HESS, COMPUTED_BY,
    ↪VARSETS);

Quadratic Cost Set
    OM.ADD_NAMED_SET('qdc', NAME, IDX_LIST, N, CP, VARSETS);

General Nonlinear Cost Set
    OM.ADD_NAMED_SET('nlc', NAME, IDX_LIST, N, FCN, VARSETS);

Legacy Cost Set
    OM.ADD_NAMED_SET('cost', NAME, IDX_LIST, N, CP, VARSETS);
```

See also `opt_model`, `add_var`, `add_lin_constraint`, `add_nln_constraint`, `add_quad_cost`, `add_nln_cost`, `add_legacy_cost()` (page 227).

add_legacy_cost(*om, name, idx, varargin*)

`add_legacy_cost()` (page 227) - Adds a set of user costs to the model.

```
OM.ADD_LEGACY_COST(NAME, CP);
OM.ADD_LEGACY_COST(NAME, CP, VARSETS);
OM.ADD_LEGACY_COST(NAME, IDX_LIST, CP);
OM.ADD_LEGACY_COST(NAME, IDX_LIST, CP, VARSETS);
```

Adds a named block of user-defined costs to the model. Each **set** is defined by the CP **struct** described below. All user-defined sets of costs are combined together into a **single set** of cost parameters in

(continues on next page)

(continued from previous page)

a **single** CP **struct** by BULD_COST_PARAMS. This **full** aggregate **set** of cost parameters can be retrieved from the model by GET_COST_PARAMS.

Examples:

```
cp1 = struct('N', N1, 'Cw', Cw1);
cp2 = struct('N', N2, 'Cw', Cw2, 'H', H, 'dd', dd, ...
            'rh', rh, 'kk', kk, 'mm', mm);
om.add_legacy_cost('usr1', cp1, {'Pg', 'Qg', 'z'});
om.add_legacy_cost('usr2', cp2, {'Vm', 'Pg', 'Qg', 'z'});

om.init_indexed_name('c', {2, 3});
for i = 1:2
    for j = 1:3
        om.add_legacy_cost('c', {i, j}, cp(i,j), ...);
    end
end
```

Let x refer to the vector formed by combining the specified VARSETS, and $f_u(x, CP)$ be the cost at x corresponding to the cost parameters contained in CP, where CP is a **struct** with the following fields:

N - $nw \times nx$ **sparse** matrix (optional, identity matrix by default)
Cw - $nw \times 1$ vector
H - $nw \times nw$ **sparse** matrix (optional, **all zeros** by default)
dd, mm - $nw \times 1$ vectors (optional, **all ones** by default)
rh, kk - $nw \times 1$ vectors (optional, **all zeros** by default)

These parameters are used as follows to compute $f_u(x, CP)$

```
R = N*x - rh

      /  kk(i),  R(i) < -kk(i)
K(i) = <  0,      -kk(i) <= R(i) <= kk(i)
      \ -kk(i),  R(i) > kk(i)

RR = R + K

U(i) = /  0, -kk(i) <= R(i) <= kk(i)
      \  1, otherwise

DDL(i) = /  1, dd(i) = 1
        \  0, otherwise

DDQ(i) = /  1, dd(i) = 2
        \  0, otherwise

Dl = diag(mm) * diag(U) * diag(DDL)
Dq = diag(mm) * diag(U) * diag(DDQ)

w = (Dl + Dq * diag(RR)) * RR

f_u(x, CP) = 1/2 * w' * H * w + Cw' * w
```

See also `opt_model`, `params_legacy_cost()` (page 225), `eval_legacy_cost()` (page 225).

5.2 Legacy Functions

5.2.1 Top-Level Simulation Functions

runpf

runpf(*casedata*, *mpopt*, *fname*, *solvedcase*)

runpf() - Runs a power flow.

```
[RESULTS, SUCCESS] = RUNPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)
```

Runs a **power** flow (full AC Newton's method by default), optionally returning a RESULTS **struct** and SUCCESS **flag**.

Inputs (all are optional):

CASEDATA : either a MATPOWER **case struct** or a string containing the name of the file with the **case** data (default is 'case9') (see CASEFORMAT and LOADCASE)
 MPOPT : MATPOWER options **struct** to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).
 FNAME : name of a file to which the pretty-printed output will be appended
 SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results **struct**, with the following fields:
 (all fields from the **input** MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)
 order - **info** used in external <-> internal data conversion
 et - elapsed **time** in seconds
 success - success **flag**, 1 = succeeded, 0 = failed
 SUCCESS : the success **flag** can additionally be returned as a second output argument

Calling syntax options:

```
results = runpf;  
results = runpf(casedata);  
results = runpf(casedata, mpopt);  
results = runpf(casedata, mpopt, fname);  
results = runpf(casedata, mpopt, fname, solvedcase);  
[results, success] = runpf(...);
```

Alternatively, **for** compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, branch, success, et] = runpf(...);
```

(continues on next page)

(continued from previous page)

If the `pf.enforce_q_lims` option is **set** to **true** (default is **false**) then, **if** **any** generator reactive **power** limit is violated after running the AC **power** flow, the corresponding bus is converted to a PQ bus, with Q_g at the limit, **and** the **case** is re-run. The voltage magnitude at the bus will deviate from the specified value in order to satisfy the reactive **power** limit. If the reference bus is converted to PQ, the first remaining PV bus will be used as the slack bus **for** the next iteration. This may result in the **real power** output at this generator being slightly off from the specified values.

Examples:

```
results = runpf('case30');
results = runpf('case30', mpopoption('pf.enforce_q_lims', 1));
```

See also `rundcpf()`.

runcpf

runcpf(*basecasedata*, *targetcasedata*, *mpop*, *fname*, *solvedcase*)

`runcpf()` - Runs a full AC continuation power flow

```
[RESULTS, SUCCESS] = RUNCPF(BASECASEDATA, TARGETCASEDATA, ...
                             MPOPT, FNAME, SOLVEDCASE)
```

Runs a **full** AC continuation **power** flow using a normalized tangent predictor **and** selected parameterization scheme, optionally returning a **RESULTS** **struct** **and** **SUCCESS** **flag**. Step **size** can be fixed **or** adaptive.

Inputs (**all** are optional):

BASECASEDATA : either a MATPOWER **case** **struct** **or** a string containing the name of the file with the **case** data defining the base loading **and** generation (default is **'case9'**)
(see **CASEFORMAT** **and** **LOADCASE**)

TARGETCASEDATA : either a MATPOWER **case** **struct** **or** a string containing the name of the file with the **case** data defining the target loading **and** generation (default is **'case9target'**)

MPOPT : MATPOWER options **struct** to override default options can be used to specify the parameterization, output options, termination criteria, **and** more (see **MPOPTION**).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with **'mat'**)

Outputs (**all** are optional):

RESULTS : results **struct**, with the following fields:
(**all** fields from the **input** MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)

(continues on next page)

(continued from previous page)

```

order - info used in external <-> internal data conversion
et - elapsed time in seconds
success - success flag, 1 = succeeded, 0 = failed
cpf - CPF output struct whose content depends on any
      user callback functions, where default contains fields:
      done_msg - string with message describing cause of
                  continuation termination
      iterations - number of continuation steps performed
      lam - (nsteps+1) row vector of lambda values from
              correction steps
      lam_hat - (nsteps+1) row vector of lambda values from
                  prediction steps
      max_lam - maximum value of lambda in RESULTS.cpf.lam
      steps - (nsteps+1) row vector of stepsizes taken at each
                  continuation step
      V - (nb x nsteps+1) complex bus voltages from
              correction steps
      V_hat - (nb x nsteps+1) complex bus voltages from
                  prediction steps
      events - an array of structs of size nevents with the
                  following fields:
                  k - continuation step number at which event was located
                  name - name of event
                  idx - index(es) of critical elements in corresponding
                        event function, e.g. index of generator reaching VAR
                        limit
                  msg - descriptive text detailing the event
SUCCESS : the success flag can additionally be returned as
          a second output argument

```

Calling syntax options:

```

results = runcpf;
results = runcpf(basecasedata, targetcasedata);
results = runcpf(basecasedata, targetcasedata, mpopt);
results = runcpf(basecasedata, targetcasedata, mpopt, fname);
results = runcpf(basecasedata, targetcasedata, mpopt, fname, solvedcase)
[results, success] = runcpf(...);

```

If the 'cpf.enforce_q_lims' option is set to true (default is false) then, if any generator reaches its reactive power limits during the AC continuation power flow,

- the corresponding bus is converted to a PQ bus, and the problem is modified to eliminate further reactive transfer on this bus
- the voltage magnitude at the bus will deviate from the specified setpoint to satisfy the reactive power limit,
- if the reference bus is converted to PQ, further real power transfer for the bus is also eliminated, and the first remaining PV bus is selected as the new slack, resulting in the transfers at both reference buses potentially deviating from the specified values
- if all reference and PV buses are converted to PQ, RUNCPF terminates with an infeasibility message.

(continues on next page)

(continued from previous page)

If the `'cpf.enforce_p_lims'` option is `set` to `true` (default is `false`) then, `if` any generator reaches its maximum active power limit during the AC continuation power flow,

- the problem is modified to eliminate further active transfer by this generator
- `if` the generator was at the reference bus, it is converted to PV `and` the first remaining PV bus is selected as the new slack.

If the `'cpf.enforce_v_lims'` option is `set` to `true` (default is `false`) then the continuation power flow is `set` to terminate `if` any bus voltage magnitude exceeds its minimum `or` maximum limit.

If the `'cpf.enforce_flow_lims'` option is `set` to `true` (default is `false`) then the continuation power flow is `set` to terminate `if` any line MVA flow exceeds its `rateA` limit.

Possible CPF termination modes:

- when `cpf.stop_at == 'NOSE'`
 - Reached steady state loading limit
 - Nose point eliminated by limit induced bifurcation
- when `cpf.stop_at == 'FULL'`
 - Traced `full` continuation curve
- when `cpf.stop_at == <target_lam_val>`
 - Reached desired `lambda`
- when `cpf.enforce_p_lims == true`
 - All generators at PMAX
- when `cpf.enforce_q_lims == true`
 - No REF `or` PV buses remaining
- when `cpf.enforce_v_lims == true`
 - Any bus voltage magnitude limit is reached
- when `cpf.enforce_flow_lims == true`
 - Any branch MVA flow limit is reached
- other
 - Base `case` power flow did `not` converge
 - Base `and` target `case` have identical load `and` generation
 - Corrector did `not` converge
 - Could `not` locate `<event_name>` event
 - Too many rollback steps triggered by callbacks

Examples:

```
results = runcpf('case9', 'case9target');
results = runcpf('case9', 'case9target', ...
    mption('cpf.adapt_step', 1));
results = runcpf('case9', 'case9target', ...
    mption('cpf.enforce_q_lims', 1));
results = runcpf('case9', 'case9target', ...
    mption('cpf.stop_at', 'FULL'));
```

See also `mption()`, `runpf()`.

runopf**runopf**(*casedata*, *mpopt*, *fname*, *solvedcase*)

runopf() - Runs an optimal power flow.

[RESULTS, SUCCESS] = RUNOPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

Runs an optimal **power** flow (AC OPF by default), optionally returning a RESULTS **struct** and SUCCESS **flag**.

Inputs (all are optional):

CASEDATA : either a MATPOWER **case struct** or a string containing the name of the file with the **case** data (default is 'case9') (see CASEFORMAT and LOADCASE)

MPOPT : MATPOWER options **struct** to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results **struct**, with the following fields:
 (all fields from the input MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)
 order - **info** used in external <-> internal data conversion
 et - elapsed **time** in seconds
 success - success **flag**, 1 = succeeded, 0 = failed
 (additional OPF fields, see OPF **for** details)

SUCCESS : the success **flag** can additionally be returned as a second output argument

Calling syntax options:

```
results = runopf;
results = runopf(casedata);
results = runopf(casedata, mpopt);
results = runopf(casedata, mpopt, fname);
results = runopf(casedata, mpopt, fname, solvedcase);
[results, success] = runopf(...);
```

Alternatively, **for** compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, gencost, branch, f, success, et] = runopf(...);
```

Example:

```
results = runopf('case30');
```

See also rundcpf(), runuopf().

runuopf**runuopf**(*casedata*, *mpopt*, *fname*, *solvedcase*)

runuopf() - Runs an optimal power flow with unit-decommitment heuristic.

[RESULTS, SUCCESS] = RUNUOPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

Runs an optimal **power** flow (AC OPF by default) with a heuristic which allows it to shut down "**expensive**" generators, optionally returning a RESULTS **struct** and SUCCESS **flag**.

Inputs (all are optional):

CASEDATA : either a MATPOWER **case struct** or a string containing the name of the file with the **case** data (default is 'case9') (see CASEFORMAT and LOADCASE)

MPOPT : MATPOWER options **struct** to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results **struct**, with the following fields:
 (all fields from the **input** MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)
 order - **info** used in external <-> internal data conversion
 et - elapsed **time** in seconds
 success - success **flag**, 1 = succeeded, 0 = failed
 (additional OPF fields, see OPF **for** details)

SUCCESS : the success **flag** can additionally be returned as a second output argument

Calling syntax options:

```
results = runuopf;
results = runuopf(casedata);
results = runuopf(casedata, mpopt);
results = runuopf(casedata, mpopt, fname);
results = runuopf(casedata, mpopt, fname, solvedcase);
[results, success] = runuopf(...);
```

Alternatively, **for** compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, gencost, branch, f, success, et] = runuopf(...);
```

Example:

```
results = runuopf('case30');
```

See also runopf(), runduopf().

rundcpf**rundcpf**(*casedata*, *mpopt*, *fname*, *solvedcase*)

rundcpf() - Runs a DC power flow.

[RESULTS, SUCCESS] = RUNCDF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

Runs a DC power flow, optionally returning a RESULTS struct and SUCCESS flag.

Inputs (all are optional):

CASEDATA : either a MATPOWER case struct or a string containing the name of the file with the case data (default is 'case9') (see CASEFORMAT and LOADCASE)

MPOPT : MATPOWER options struct to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved case will be saved in MATPOWER case format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results struct, with the following fields:
 (all fields from the input MATPOWER case, i.e. bus, branch, gen, etc., but with solved voltages, power flows, etc.)
 order - info used in external <-> internal data conversion
 et - elapsed time in seconds
 success - success flag, 1 = succeeded, 0 = failed

SUCCESS : the success flag can additionally be returned as a second output argument

Calling syntax options:

```
results = rundcpf;
results = rundcpf(casedata);
results = rundcpf(casedata, mpopt);
results = rundcpf(casedata, mpopt, fname);
results = rundcpf(casedata, mpopt, fname, solvedcase);
[results, success] = rundcpf(...);
```

Alternatively, for compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, branch, success, et] = rundcpf(...);
```

Example:

```
results = rundcpf('case30');
```

See also runpf().

rundcopf**rundcopf**(*casedata, mpopt, fname, solvedcase*)

rundcopf() - Runs a DC optimal power flow.

[RESULTS, SUCCESS] = RUNCOPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

Runs a DC optimal **power** flow, optionally returning a RESULTS **struct** and SUCCESS **flag**.

Inputs (all are optional):

CASEDATA : either a MATPOWER **case struct** or a string containing the name of the file with the **case** data (default is 'case9') (see CASEFORMAT and LOADCASE)

MPOPT : MATPOWER options **struct** to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results **struct**, with the following fields:
 (all fields from the input MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)
 order - **info** used in external <-> internal data conversion
 et - elapsed **time** in seconds
 success - success **flag**, 1 = succeeded, 0 = failed
 (additional OPF fields, see OPF **for** details)

SUCCESS : the success **flag** can additionally be returned as a second output argument

Calling syntax options:

```
results = rundcopf;
results = rundcopf(casedata);
results = rundcopf(casedata, mpopt);
results = rundcopf(casedata, mpopt, fname);
results = rundcopf(casedata, mpopt, fname, solvedcase);
[results, success] = rundcopf(...);
```

Alternatively, **for** compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, gencost, branch, f, success, et] = rundcopf(...);
```

Example:

```
results = rundcopf('case30');
```

See also runopf(), runduopf().

runduopf**runduopf**(*casedata, mpopt, fname, solvedcase*)

runduopf() - Runs a DC optimal power flow with unit-decommitment heuristic.

[RESULTS, SUCCESS] = RUNDUOPF(CASEDATA, MPOPT, FNAME, SOLVEDCASE)

Runs a DC optimal **power** flow with a heuristic which allows it to shut down "**expensive**" generators optionally returning a RESULTS **struct** and SUCCESS **flag**.

Inputs (all are optional):

CASEDATA : either a MATPOWER **case struct** or a string containing the name of the file with the **case** data (default is 'case9') (see CASEFORMAT and LOADCASE)

MPOPT : MATPOWER options **struct** to override default options can be used to specify the solution algorithm, output options termination tolerances, and more (see MPOPTION).

FNAME : name of a file to which the pretty-printed output will be appended

SOLVEDCASE : name of file to which the solved **case** will be saved in MATPOWER **case** format (M-file will be assumed unless the specified name ends with '.mat')

Outputs (all are optional):

RESULTS : results **struct**, with the following fields:
 (all fields from the input MATPOWER **case**, i.e. bus, branch, gen, etc., but with solved voltages, **power** flows, etc.)
 order - **info** used in external <-> internal data conversion
 et - elapsed **time** in seconds
 success - success **flag**, 1 = succeeded, 0 = failed
 (additional OPF fields, see OPF **for** details)

SUCCESS : the success **flag** can additionally be returned as a second output argument

Calling syntax options:

```
results = runduopf;
results = runduopf(casedata);
results = runduopf(casedata, mpopt);
results = runduopf(casedata, mpopt, fname);
results = runduopf(casedata, mpopt, fname, solvedcase);
[results, success] = runduopf(...);
```

Alternatively, **for** compatibility with previous versions of MATPOWER, some of the results can be returned as individual output arguments:

```
[baseMVA, bus, gen, gencost, branch, f, success, et] = runduopf(...);
```

Example:

```
results = runduopf('case30');
```

See also rundcpf(), runuopf().

runopf_w_res

runopf_w_res(*varargin*)

runopf_w_res() (page 238) - Runs an optimal power flow with fixed zonal reserves.

```
RESULTS = RUNOPF_W_RES(CASEDATA, MPOPT, FNAME, SOLVEDCASE)
[RESULTS, SUCCESS] = RUNOPF_W_RES(CASEDATA, MPOPT, FNAME, SOLVEDCASE)
```

Runs an optimal **power** flow with the addition of reserve requirements specified as a **set** of fixed zonal reserves. See RUNOPF **for** a description of the **input and output** arguments, which are the same, with the exception that the **case file or struct** CASEDATA must define a **'reserves'** field, which is a **struct** with the following fields:

zones	nrz x ng, zone(i, j) = 1, if gen j belongs to zone i 0, otherwise
req	nrz x 1, zonal reserve requirement in MW
cost	(ng or ngr) x 1, cost of reserves in \$/MW
qty	(ng or ngr) x 1, max quantity of reserves in MW (optional)

where nrz is the number of reserve zones and ngr is the number of generators belonging to at least one reserve zone and ng is the total number of generators.

In addition to the normal OPF output, the RESULTS struct contains a new 'reserves' field with the following fields, in addition to those provided in the input:

R	- ng x 1, reserves provided by each gen in MW
Rmin	- ng x 1, lower limit on reserves provided by each gen, (MW)
Rmax	- ng x 1, upper limit on reserves provided by each gen, (MW)
mu.l	- ng x 1, shadow price on reserve lower limit, (\$/MW)
mu.u	- ng x 1, shadow price on reserve upper limit, (\$/MW)
mu.Pmax	- ng x 1, shadow price on $P_g + R \leq P_{max}$ constraint, (\$/MW)
prc	- ng x 1, reserve price for each gen equal to maximum of the shadow prices on the zonal requirement constraint for each zone the generator belongs to

See T_CASE30_USERFCNS for an example case file with fixed reserves, and TOGGLE_RESERVES for the implementation.

Calling syntax options:

```
results = runopf_w_res(casedata);
results = runopf_w_res(casedata, mppopt);
results = runopf_w_res(casedata, mppopt, fname);
results = runopf_w_res(casedata, mppopt, fname, solvedcase);
[results, success] = runopf_w_res(...);
```

Example:

```
results = runopf_w_res('t_case30_userfcns');
```

See also `runopf()`, `toggle_reserves()` (page 309), `t_case30_userfcns()` (page 384).

5.2.2 Input/Output Functions

caseformat

caseformat()

caseformat - Defines the MATPOWER case file format.

A MATPOWER **case** file is an M-file or MAT-file that defines or returns a **struct** named `mpc`, referred to as a "MATPOWER case struct". The fields of this **struct** are `baseMVA`, `bus`, `gen`, `branch`, and (optional) `gencost`. With the exception of `baseMVA`, a scalar, each data variable is a matrix, where a row corresponds to a **single** bus, branch, gen, etc. The format of the data is similar to the PTI format described in

<https://labs.ece.uw.edu/pstca/formats/pti.txt>

except where noted. An item marked with (+) indicates that it is included in this data but is **not** part of the PTI format. An item marked with (-) is one that is in the PTI format but is **not** included here. Those marked with (2) were added **for version 2** of the **case** file format. The **columns for** each data matrix are given below.

MATPOWER Case Version Information:

There are two versions of the MATPOWER **case** file format. The current **version** of MATPOWER uses **version 2** of the MATPOWER **case** format internally, and includes a `'version'` field with a value of `'2'` to make the **version** explicit. Earlier versions of MATPOWER used the **version 1 case** format, which defined the data matrices as individual variables, as opposed to fields of a **struct**. Case files in **version 1** format with OPF data also included an (unused) `'areas'` variable. While the **version 1** format has **now** been deprecated, it is still be handled automatically by `LOADCASE` and `SAVECASE` which are able to **load and save case** files in both **version 1 and version 2** formats.

See `IDX_BUS`, `IDX_BRCH`, `IDX_GEN`, `IDX_AREA` and `IDX_COST` regarding constants which can be used as named column indices **for** the data matrices. Also described in the first three are additional results **columns** that are added to the bus, branch and gen matrices by the **power flow and OPF solvers**.

The **case struct** also allows **for** additional fields to be included. The OPF is designed to recognize fields named `A`, `l`, `u`, `H`, `Cw`, `N`, `fparm`, `z0`, `z1` and `zu` as parameters used to directly extend the OPF formulation (see OPF **for** details). Additional standard optional fields include `bus_name`, `gentype` and `genfuel`. Other user-defined fields may also be included and will be automatically loaded by the `LOADCASE function` and, given an appropriate `'savecase'` callback **function** (see `ADD_USERFCN`), saved by the `SAVECASE function`.

Bus Data Format

- 1 bus number (positive integer)
- 2 bus **type**

(continues on next page)

(continued from previous page)

```

        PQ bus      = 1
        PV bus      = 2
        reference bus = 3
        isolated bus = 4
3  Pd, real power demand (MW)
4  Qd, reactive power demand (MVar)
5  Gs, shunt conductance (MW demanded at V = 1.0 p.u.)
6  Bs, shunt susceptance (MVar injected at V = 1.0 p.u.)
7  area number, (positive integer)
8  Vm, voltage magnitude (p.u.)
9  Va, voltage angle (degrees)
(-) (bus name)
10 baseKV, base voltage (kV)
11 zone, loss zone (positive integer)
(+) 12 maxVm, maximum voltage magnitude (p.u.)
(+) 13 minVm, minimum voltage magnitude (p.u.)

Generator Data Format
1  bus number
(-) (machine identifier, 0-9, A-Z)
2  Pg, real power output (MW)
3  Qg, reactive power output (MVar)
4  Qmax, maximum reactive power output (MVar)
5  Qmin, minimum reactive power output (MVar)
6  Vg, voltage magnitude setpoint (p.u.)
(-) (remote controlled bus index)
7  mBase, total MVA base of this machine, defaults to baseMVA
(-) (machine impedance, p.u. on mBase)
(-) (step up transformer impedance, p.u. on mBase)
(-) (step up transformer off nominal turns ratio)
8  status, > 0 - machine in service
        <= 0 - machine out of service
(-) (% of total VAR's to come from this gen in order to hold V at
      remote bus controlled by several generators)
9  Pmax, maximum real power output (MW)
10 Pmin, minimum real power output (MW)
(2) 11 Pc1, lower real power output of PQ capability curve (MW)
(2) 12 Pc2, upper real power output of PQ capability curve (MW)
(2) 13 Qc1min, minimum reactive power output at Pc1 (MVar)
(2) 14 Qc1max, maximum reactive power output at Pc1 (MVar)
(2) 15 Qc2min, minimum reactive power output at Pc2 (MVar)
(2) 16 Qc2max, maximum reactive power output at Pc2 (MVar)
(2) 17 ramp rate for load following/AGC (MW/min)
(2) 18 ramp rate for 10 minute reserves (MW)
(2) 19 ramp rate for 30 minute reserves (MW)
(2) 20 ramp rate for reactive power (2 sec timescale) (MVar/min)
(2) 21 APF, area participation factor

Branch Data Format
1  f, from bus number
2  t, to bus number
(-) (circuit identifier)

```

(continues on next page)

(continued from previous page)

```

3  r, resistance (p.u.)
4  x, reactance (p.u.)
5  b, total line charging susceptance (p.u.)
6  rateA, MVA rating A (long term rating), set to 0 for unlimited
7  rateB, MVA rating B (short term rating), set to 0 for unlimited
8  rateC, MVA rating C (emergency rating), set to 0 for unlimited
9  tap, transformer off nominal turns ratio, if non-zero
   (taps at "from" bus, impedance at "to" bus, i.e. if  $r = x = b = 0$ ,
    then  $\text{tap} = V_f / V_t$ ;  $\text{tap} = 0$  used to indicate transmission
    line rather than transformer, i.e. mathematically equivalent to
    transformer with  $\text{tap} = 1$ )
10 shift, transformer phase shift angle (degrees), positive => delay
(-) (Gf, shunt conductance at from bus p.u.)
(-) (Bf, shunt susceptance at from bus p.u.)
(-) (Gt, shunt conductance at to bus p.u.)
(-) (Bt, shunt susceptance at to bus p.u.)
11 initial branch status, 1 - in service, 0 - out of service
(2) 12 minimum angle difference,  $\text{angle}(V_f) - \text{angle}(V_t)$  (degrees)
(2) 13 maximum angle difference,  $\text{angle}(V_f) - \text{angle}(V_t)$  (degrees)
   (The voltage angle difference is taken to be unbounded below
    if  $\text{ANGMIN} < -360$  and unbounded above if  $\text{ANGMAX} > 360$ .
    If both parameters are zero, it is unconstrained.)

(+) Generator Cost Data Format
NOTE: If gen has ng rows, then the first ng rows of gencost contain
the cost for active power produced by the corresponding generators.
If gencost has 2*ng rows then rows ng+1 to 2*ng contain the reactive
power costs in the same format.
1  model, 1 - piecewise linear, 2 - polynomial
2  startup, startup cost in US dollars
3  shutdown, shutdown cost in US dollars
4  N (= n+1), number of data points to follow defining an n-segment
   piecewise linear cost function, or of cost coefficients defining
   an n-th order polynomial cost function
5  and following, parameters defining total cost function  $f(p)$ ,
   units of  $f$  and  $p$  are $/hr and MW (or MVar), respectively.
   (MODEL = 1) : p1, f1, p2, f2, ..., pN, fN
               where  $p_1 < p_2 < \dots < p_N$  and the cost  $f(p)$  is defined by
               the coordinates  $(p_1, f_1)$ ,  $(p_2, f_2)$ , ...,  $(p_N, f_N)$  of the
               end/break-points of the piecewise linear cost function
   (MODEL = 2) : cn, ..., c1, c0
               N coefficients of an n-th order polynomial cost function,
               starting with highest order, where cost is
                $f(p) = c_n p^n + \dots + c_1 p + c_0$ 

(+) Area Data Format (deprecated)
   (this data is not used by MATPOWER and is no longer necessary for
    version 2 case files with OPF data).
1  i, area number
2  price_ref_bus, reference bus for that area

```

See also `loadcase()`, `savecase()`, `idx_bus()` (page 342), `idx_brch()` (page 341), `idx_gen()` (page 347), `idx_area` `idx_cost()` (page 343).

cdf2mpc

cdf2mpc(cdf_file_name, mpc_name, verbose)

cdf2mpc() - Converts an IEEE CDF data file into a MATPOWER case struct.

```

MPC = CDF2MPC(CDF_FILE_NAME)
MPC = CDF2MPC(CDF_FILE_NAME, VERBOSE)
MPC = CDF2MPC(CDF_FILE_NAME, MPC_NAME)
MPC = CDF2MPC(CDF_FILE_NAME, MPC_NAME, VERBOSE)
[MPC, WARNINGS] = CDF2MPC(CDF_FILE_NAME, ...)

```

Converts an IEEE Common Data Format (CDF) data file into a MATPOWER **case struct**.

Input:

```

CDF_FILE_NAME : name of the IEEE CDF file to be converted
MPC_NAME      : (optional) file name to use to save the resulting
                MATPOWER case
VERBOSE       : 1 (default) to display progress info, 0 otherwise

```

Output(s):

```

MPC      : resulting MATPOWER case struct
WARNINGS : (optional) cell array of strings containing warning
            messages (included by default in comments of MPC_NAME).

```

The IEEE CDF does **not** include some data need to **run** an optimal **power** flow. This script creates default values **for** some of this data as follows:

```

Bus data:
  Vmin = 0.94 p.u.
  Vmax = 1.06 p.u.
Gen data:
  Pmin = 0 MW
  Pmax = Pg + baseMVA
Gen cost data:
  Quadratic costs with:
    c2 = 10 / Pg, c1 = 20, c0 = 0, if Pg is non-zero, and
    c2 = 0.01,    c1 = 40, c0 = 0, if Pg is zero
  This should yield an OPF solution "close" to the
  existing solution (assuming it is a solved case)
  with lambdas near $40/MWh. See 'help caseformat'
for details on the cost curve format.

```

CDF2MPC may modify some of the data which are "**infeasible**" **for** running optimal **power** flow. If so, **warning** information will be printed out on screen.

Note: Since our code can **not** handle transformers with variable tap, you may **not** expect to **get** exactly the same **power** flow solution using converted data. This is the **case** when we converted ieee300.cdf.

loadcase

loadcase(casefile)

loadcase() - Load .m or .mat case files or data struct in MATPOWER format.

```
[BASEMVA, BUS, GEN, BRANCH, AREAS, GENCOST] = LOADCASE(CASEFILE)
[BASEMVA, BUS, GEN, BRANCH, GENCOST] = LOADCASE(CASEFILE)
[BASEMVA, BUS, GEN, BRANCH] = LOADCASE(CASEFILE)
MPC = LOADCASE(CASEFILE)
```

Returns the individual data matrices or a struct containing them as fields.

Here CASEFILE is either (1) a struct containing the fields baseMVA, bus, gen, branch and, optionally, areas and/or gencost, or (2) a string containing the name of the file. If CASEFILE contains the extension '.mat' or '.m', then the explicit file is searched. If CASEFILE contains no extension, then LOADCASE looks for a MAT-file first, then for an M-file. If the file does not exist or doesn't define all required matrices, the routine aborts with an appropriate error message.

If the input data is from an M-file or MAT-file defining individual data matrices, or from a struct with out a 'version' field whose GEN matrix has fewer than 21 columns, then it is assumed to be a MATPOWER case file in version 1 format, and will be converted to version 2 format.

mpoption

mpoption(varargin)

mpoption() - Used to set and retrieve a MATPOWER options struct.

```
OPT = MPOPTION
```

Returns the default options struct.

```
OPT = MPOPTION(OVERRIDES)
```

Returns the default options struct, with some fields overridden by values from OVERRIDES, which can be a struct or the name of a function that returns a struct.

```
OPT = MPOPTION(NAME1, VALUE1, NAME2, VALUE2, ...)
```

Same as previous, except override options are specified by NAME, VALUE pairs. This can be used to set any part of the options struct. The names can be individual fields or multi-level field names with embedded periods. The values can be scalars or structs.

For backward compatibility, the NAMES and VALUES may correspond to old-style MATPOWER option names (elements in the old-style options vector) as well.

(continues on next page)

(continued from previous page)

`OPT = MPOPTION(OPT0)`
 Converts an old-style options vector `OPT0` into the corresponding options `struct`. If `OPT0` is an options `struct` it does nothing.

`OPT = MPOPTION(OPT0, OVERRIDES)`
 Applies overrides to an existing `set` of options, `OPT0`, which can be an old-style options vector `or` an options `struct`.

`OPT = MPOPTION(OPT0, NAME1, VALUE1, NAME2, VALUE2, ...)`
 Same as above except it uses the old-style options vector `OPT0` as a base instead of the old default options vector.

`OPT_VECTOR = MPOPTION(OPT, [])`
 Creates `and` returns an old-style options vector from an options `struct` `OPT`.

Note: The use of old-style MATPOWER options vectors `and` their names `and` values has been deprecated `and` will be removed in a future `version` of MATPOWER. Until then, `all` uppercase option names are `not` permitted `for` new top-level options.

Examples:

```
mpopt = mpooption('pf.alg', 'FDXB', 'pf.tol', 1e-4);
mpopt = mpooption(mpooption, 'opf.dc.solver', 'CPLEX', 'verbose', 2);
```

The currently defined options are as follows:

name	default	description [options]

Model options:		
model	'AC'	AC vs. DC <code>power</code> flow model
['AC' - use nonlinear AC model & corresponding algorithms/options]		
['DC' - use linear DC model & corresponding algorithms/options]		
Power Flow options:		
pf.alg	'NR'	AC <code>power</code> flow algorithm
['NR' - Newton's method (formulation depends on values of]		
[pf.current_balance <code>and</code> pf.v_cartesian options)]		
['NR-SP' - Newton's method (<code>power</code> mismatch, <code>polar</code>)]		
['NR-SC' - Newton's method (<code>power</code> mismatch, cartesian)]		
['NR-SH' - Newton's method (<code>power</code> mismatch, hybrid)]		
['NR-IP' - Newton's method (current mismatch, <code>polar</code>)]		
['NR-IC' - Newton's method (current mismatch, cartesian)]		
['NR-IH' - Newton's method (current mismatch, hybrid)]		
['FDXB' - Fast-Decoupled (XB <code>version</code>)]		
['FDBX' - Fast-Decoupled (BX <code>version</code>)]		
['GS' - Gauss-Seidel]		
['ZG' - Implicit Z-bus Gauss]		
['PQSUM' - Power Summation method (radial networks only)]		
['ISUM' - Current Summation method (radial networks only)]		
['YSUM' - Admittance Summation method (radial networks only)]		

(continues on next page)

(continued from previous page)

```

pf.current_balance      0          type of nodal balance equation
[ 0 - use complex power balance equations ]
[ 1 - use complex current balance equations ]
pf.v_cartesian          0          voltage representation
[ 0 - bus voltage variables represented in polar coordinates ]
[ 1 - bus voltage variables represented in cartesian coordinates ]
[ 2 - hybrid, polar updates computed via modified cartesian Jacobian ]
pf.tol                  1e-8       termination tolerance on per unit
                                   P & Q mismatch
pf.nr.max_it            10         maximum number of iterations for
                                   Newton's method
pf.nr.lin_solver        ''         linear solver passed to MPLINSOLVE to
                                   solve Newton update step
[ '' - default to '\' for small systems, 'LU3' for larger ones ]
[ '\' - built-in backslash operator ]
[ 'LU' - explicit default LU decomposition and back substitution ]
[ 'LU3' - 3 output arg form of LU, Gilbert-Peierls algorithm with ]
[ approximate minimum degree (AMD) reordering ]
[ 'LU4' - 4 output arg form of LU, UMFPACK solver (same as 'LU') ]
[ 'LU5' - 5 output arg form of LU, UMFPACK solver, w/row scaling ]
[ (see MPLINSOLVE for complete list of all options) ]
pf.fd.max_it            30         maximum number of iterations for
                                   fast decoupled method
pf.gs.max_it            1000       maximum number of iterations for
                                   Gauss-Seidel method
pf.zg.max_it            1000       maximum number of iterations for
                                   Implicit Z-bus Gauss method
pf.radial.max_it        20         maximum number of iterations for
                                   radial power flow methods
pf.radial.vcorr         0          perform voltage correction procedure
                                   in distribution power flow
[ 0 - do NOT perform voltage correction ]
[ 1 - perform voltage correction ]
pf.enforce_q_lims       0          enforce gen reactive power limits at
                                   expense of |V|
[ 0 - do NOT enforce limits ]
[ 1 - enforce limits, simultaneous bus type conversion ]
[ 2 - enforce limits, one-at-a-time bus type conversion ]

Continuation Power Flow options:
cpf.parameterization    3          choice of parameterization
[ 1 - natural ]
[ 2 - arc length ]
[ 3 - pseudo arc length ]
cpf.stop_at             'NOSE'     determines stopping criterion
[ 'NOSE' - stop when nose point is reached ]
[ 'FULL' - trace full nose curve ]
[ <lam_stop> - stop upon reaching specified target lambda value ]
cpf.enforce_p_lims      0          enforce gen active power limits
[ 0 - do NOT enforce limits ]
[ 1 - enforce limits, simultaneous bus type conversion ]
cpf.enforce_q_lims      0          enforce gen reactive power limits at

```

(continues on next page)

(continued from previous page)

```

                                expense of |V|
    [ 0 - do NOT enforce limits ]
    [ 1 - enforce limits, simultaneous bus type conversion ]
    cpf.enforce_v_lims      0      enforce bus voltage magnitude limits
    [ 0 - do NOT enforce limits ]
    [ 1 - enforce limits, termination on detection ]
    cpf.enforce_flow_lims   0      enforce branch flow MVA limits
    [ 0 - do NOT enforce limits ]
    [ 1 - enforce limits, termination on detection ]
    cpf.step                0.05   continuation power flow step size
    cpf.adapt_step          0      toggle adaptive step size feature
    [ 0 - adaptive step size disabled ]
    [ 1 - adaptive step size enabled ]
    cpf.step_min            1e-4   minimum allowed step size
    cpf.step_max            0.2    maximum allowed step size
    cpf.adapt_step_damping  0.7    damping factor for adaptive step
                                sizing
    cpf.adapt_step_tol      1e-3   tolerance for adaptive step sizing
    cpf.target_lam_tol      1e-5   tolerance for target lambda detection
    cpf.nose_tol            1e-5   tolerance for nose point detection (pu)
    cpf.p_lims_tol          0.01   tolerance for generator active
                                power limit enforcement (MW)
    cpf.q_lims_tol          0.01   tolerance for generator reactive
                                power limit enforcement (MVAR)
    cpf.v_lims_tol          1e-4   tolerance for bus voltage
                                magnitude enforcement (p.u)
    cpf.flow_lims_tol       0.01   tolerance for line MVA flow
                                enforcement (MVA)
    cpf.plot.level          0      control plotting of nose curve
    [ 0 - do not plot nose curve ]
    [ 1 - plot when completed ]
    [ 2 - plot incrementally at each iteration ]
    [ 3 - same as 2, with 'pause' at each iteration ]
    cpf.plot.bus            <empty> index of bus whose voltage is to be
                                plotted
    cpf.user_callback       <empty> string containing the name of a user
                                callback function, or struct with
                                function name, and optional priority
                                and/or args, or cell array of such
                                strings and/or structs, see
                                'help cpf_default_callback' for details

```

Optimal Power Flow options:

name	default	description [options]
opf.ac.solver	'DEFAULT'	AC optimal power flow solver
['DEFAULT']		- choose default solver, i.e. 'MIPS'
['MIPS']		- MIPS, MATPOWER Interior Point Solver, primal/dual
[]		interior point method (pure MATLAB/Octave)
['FMINCON']		- MATLAB Optimization Toolbox, FMINCON
['IPOPT']		- IPOPT, requires MEX interface to IPOPT solver
[]		available from:

(continues on next page)

(continued from previous page)

```

[                                     https://github.com/coin-or/Ipopt                                     ]
[ 'KNITRO' - Artelys Knitro, requires Artelys Knitro solver,           ]
[                                     available from:https://www.artelys.com/solvers/knitro/           ]
[ 'MINOPF' - MINOPF, MINOS-based solver, requires optional           ]
[                                     MEX-based MINOPF package, available from:           ]
[                                     http://www.pserc.cornell.edu/minopf/                                     ]
[ 'PDIPM' - PDIPM, primal/dual interior point method, requires       ]
[                                     optional MEX-based TSPOPF package, available from:   ]
[                                     http://www.pserc.cornell.edu/tspopf/                                     ]
[ 'SDPOPF' - SDPOPF, solver based on semidefinite relaxation of       ]
[                                     OPF problem, requires optional packages:           ]
[                                     SDP_PF, available in extras/sdp_pf           ]
[                                     YALMIP, available from:           ]
[                                     https://yalmip.github.io                                     ]
[                                     SDP solver such as SeDuMi, available from:         ]
[                                     http://sedumi.ie.lehigh.edu/                                     ]
[ 'TRALM' - TRALM, trust region based augmented Langrangian         ]
[                                     method, requires TSPOPF (see 'PDIPM')           ]
opf.dc.solver      'DEFAULT'      DC optimal power flow solver
[ 'DEFAULT' - choose solver based on availability in the following   ]
[                                     order: 'GUROBI', 'CPLEX', 'MOSEK', 'OT',           ]
[                                     'GLPK' (linear costs only), 'BPMPD', 'MIPS'         ]
[ 'MIPS' - MIPS, MATPOWER Interior Point Solver, primal/dual       ]
[                                     interior point method (pure MATLAB/Octave)         ]
[ 'BPMPD' - BPMPD, requires optional MEX-based BPMPD_MEX package    ]
[                                     available from: http://www.pserc.cornell.edu/bpmpd/                                     ]
[ 'CLP' - CLP, requires interface to COIN-OP LP solver              ]
[                                     available from:https://github.com/coin-or/Clp                                     ]
[ 'CPLEX' - CPLEX, requires CPLEX solver available from:           ]
[                                     https://www.ibm.com/analytics/cplex-optimizer                                     ]
[ 'GLPK' - GLPK, requires interface to GLPK solver                 ]
[                                     available from: https://www.gnu.org/software/glpk/                                     ]
[                                     (GLPK does not work with quadratic cost functions)   ]
[ 'GUROBI' - GUROBI, requires Gurobi optimizer (v. 5+)             ]
[                                     available from: https://www.gurobi.com/                                     ]
[ 'IPOPT' - IPOPT, requires MEX interface to IPOPT solver           ]
[                                     available from:           ]
[                                     https://github.com/coin-or/Ipopt                                     ]
[ 'MOSEK' - MOSEK, requires MATLAB interface to MOSEK solver       ]
[                                     available from: https://www.mosek.com/                                     ]
[ 'OSQP' - OSQP, requires MATLAB interface to OSQP solver          ]
[                                     available from: https://osqp.org/                                     ]
[ 'OT' - MATLAB Optimization Toolbox, QUADPROG, LINPROG            ]
opf.current_balance 0 type of nodal balance equation
[ 0 - use complex power balance equations ]
[ 1 - use complex current balance equations ]
opf.v_cartesian 0 voltage representation
[ 0 - bus voltage variables represented in polar coordinates ]
[ 1 - bus voltage variables represented in cartesian coordinates ]
opf.violation 5e-6 constraint violation tolerance
opf.use_vg 0 respect gen voltage setpt [ 0-1 ]
[ 0 - use specified bus Vmin & Vmax, and ignore gen Vg ]

```

(continues on next page)

(continued from previous page)

```

[ 1 - replace specified bus Vmin & Vmax by corresponding gen Vg ]
[ between 0 and 1 - use a weighted average of the 2 options ]
opf.flow_lim      'S'      quantity limited by branch flow
                        constraints
[ 'S' - apparent power flow (limit in MVA) ]
[ 'P' - active power flow, implemented using P (limit in MW) ]
[ '2' - active power flow, implemented using P^2 (limit in MW) ]
[ 'I' - current magnitude (limit in MVA at 1 p.u. voltage) ]
opf.ignore_angle_lim  0      angle diff limits for branches
[ 0 - include angle difference limits, if specified ]
[ 1 - ignore angle difference limits even if specified ]
opf.softlims.default  1      behavior of OPF soft limits for
                        which parameters are not explicitly
                        provided
[ 0 - do not include softlims if not explicitly specified ]
[ 1 - include softlims w/default values if not explicitly specified ]
opf.start          0      strategy for initializing OPF start pt
[ 0 - default, MATPOWER decides based on solver ]
[ (currently identical to 1) ]
[ 1 - ignore current state in MATPOWER case (only applies to ]
[ fmincon, Ipopt, Knitro and MIPS, which use an interior pt ]
[ estimate; others use current state as with opf.start = 2) ]
[ 2 - use current state in MATPOWER case ]
[ 3 - solve power flow and use resulting state ]
opf.return_raw_der  0      for AC OPF, return constraint and
                        derivative info in results.raw
                        (in fields g, dg, df, d2f) [ 0 or 1 ]

```

Output options:

name	default	description [options]	
verbose	1	amount of progress info printed	
[0 - print no progress info]
[1 - print a little progress info]
[2 - print a lot of progress info]
[3 - print all progress info]
out.all	-1	controls pretty-printing of results	
[-1 - individual flags control what prints]
[0 - do not print anything (overrides individual flags, ignored]
[for files specified as FNAME arg to runpf(), runopf(), etc.)]
[1 - print everything (overrides individual flags)]
out.sys_sum	1	print system summary	[0 or 1]
out.area_sum	0	print area summaries	[0 or 1]
out.bus	1	print bus detail	[0 or 1]
out.branch	1	print branch detail	[0 or 1]
out.gen	0	print generator detail	[0 or 1]
out.lim.all	-1	controls constraint info output	
[-1 - individual flags control what constraint info prints]
[0 - no constraint info (overrides individual flags)]
[1 - binding constraint info (overrides individual flags)]
[2 - all constraint info (overrides individual flags)]
out.lim.v	1	control voltage limit info	

(continues on next page)

(continued from previous page)

```

[ 0 - do not print ]
[ 1 - print binding constraints only ]
[ 2 - print all constraints ]
[ (same options for OUT_LINE_LIM, OUT_PG_LIM, OUT_QG_LIM) ]
out.lim.line      1      control line flow limit info
out.lim.pg        1      control gen active power limit info
out.lim.qg        1      control gen reactive pwr limit info
out.force         0      print results even if success
                        flag = 0 [ 0 or 1 ]
out.suppress_detail -1    suppress all output but system summary
[ -1 - suppress details for large systems (> 500 buses) ]
[ 0 - do not suppress any output specified by other flags ]
[ 1 - suppress all output except system summary section ]
[ (overrides individual flags, but not out.all = 1) ]

```

Solver specific options:

name	default	description [options]
MIPS:		
mips.linsolver	' '	linear system solver
[' ' or '\'	build-in backslash \ operator (e.g. x = A \ b)]	
['PARDISO'	PARDISO solver (if available)]	
mips.feastol	0	feasibility (equality) tolerance (set to opf.violation by default)
mips.gradtol	1e-6	gradient tolerance
mips.comptol	1e-6	complementary condition (inequality) tolerance
mips.costtol	1e-6	optimality tolerance
mips.max_it	150	maximum number of iterations
mips.step_control	0	enable step-size cntrl [0 or 1]
mips.sc.red_it	20	maximum number of reductions per iteration with step control
mips.xi	0.99995	constant used in alpha updates*
mips.sigma	0.1	centering parameter*
mips.z0	1	used to initialize slack variables*
mips.alpha_min	1e-8	returns "Numerically Failed" if either alpha parameter becomes smaller than this value*
mips.rho_min	0.95	lower bound on rho_t*
mips.rho_max	1.05	upper bound on rho_t*
mips.mu_threshold	1e-5	KT multipliers smaller than this value for non-binding constraints are forced to zero
mips.max_stepsize	1e10	returns "Numerically Failed" if the 2-norm of the reduced Newton step exceeds this value*

* See the corresponding Appendix in the manual for details.

CPLEX:

```

cplex.lpmethod      0      solution algorithm for LP problems
[ 0 - automatic: let CPLEX choose ]
[ 1 - primal simplex ]

```

(continues on next page)

(continued from previous page)

```

[ 2 - dual simplex ]
[ 3 - network simplex ]
[ 4 - barrier ]
[ 5 - sifting ]
[ 6 - concurrent (dual, barrier, and primal) ]
cplex.qpmethod      0          solution algorithm for QP problems
[ 0 - automatic: let CPLEX choose ]
[ 1 - primal simplex optimizer ]
[ 2 - dual simplex optimizer ]
[ 3 - network optimizer ]
[ 4 - barrier optimizer ]
cplex.opts          <empty>    see CPLEX_OPTIONS for details
cplex.opt_fname     <empty>    see CPLEX_OPTIONS for details
cplex.opt           0          see CPLEX_OPTIONS for details

FMINCON:
fmincon.alg         4          algorithm used by fmincon() for OPF
                           for Opt Toolbox 4 and later
[ 1 - active-set (not suitable for large problems) ]
[ 2 - interior-point, w/default 'bfgs' Hessian approx ]
[ 3 - interior-point, w/ 'lbfgs' Hessian approx ]
[ 4 - interior-point, w/exact user-supplied Hessian ]
[ 5 - interior-point, w/Hessian via finite differences ]
[ 6 - sqp (not suitable for large problems) ]
fmincon.tol_x       1e-4       termination tol on x
fmincon.tol_f       1e-4       termination tol on f
fmincon.max_it      0          maximum number of iterations
                           [ 0 => default ]

GUROBI:
gurobi.method       0          solution algorithm (Method)
[ -1 - automatic, let Gurobi decide ]
[ 0 - primal simplex ]
[ 1 - dual simplex ]
[ 2 - barrier ]
[ 3 - concurrent (LP only) ]
[ 4 - deterministic concurrent (LP only) ]
[ 5 - deterministic concurrent simplex (LP only) ]
gurobi.timelimit    Inf        maximum time allowed (TimeLimit)
gurobi.threads      0          max number of threads (Threads)
gurobi.opts         <empty>    see GUROBI_OPTIONS for details
gurobi.opt_fname    <empty>    see GUROBI_OPTIONS for details
gurobi.opt          0          see GUROBI_OPTIONS for details

IPOPT:
ipopt.opts          <empty>    see IPOPT_OPTIONS for details
ipopt.opt_fname     <empty>    see IPOPT_OPTIONS for details
ipopt.opt           0          see IPOPT_OPTIONS for details

KNITRO:
knitro.tol_x        1e-4       termination tol on x
knitro.tol_f        1e-4       termination tol on f

```

(continues on next page)

(continued from previous page)

knitro.maxit	0	maximum number of iterations [0 => default]
knitro.opts	<empty>	see KNITRO_OPTIONS for details
knitro.opt_fname	<empty>	name of user-supplied native Knitro options file that overrides all other options
knitro.opt	0	if knitro.opt_fname is empty and knitro.opt is a non-zero integer N then knitro.opt_fname is auto- generated as: 'knitro_user_options_N.txt'
LINPROG:		
linprog	<empty>	LINPROG options passed to OPTIMOPTIONS or OPTIMSET. see LINPROG in the Optimization Toolbox for details
MINOPF:		
minopf.feastol	0 (1e-3)	primal feasibility tolerance (set to opf.violation by default)
minopf.rowtol	0 (1e-3)	row tolerance
minopf.xtol	0 (1e-4)	x tolerance
minopf.majdamp	0 (0.5)	major damping parameter
minopf.mindamp	0 (2.0)	minor damping parameter
minopf.penalty	0 (1.0)	penalty parameter
minopf.major_it	0 (200)	major iterations
minopf.minor_it	0 (2500)	minor iterations
minopf.max_it	0 (2500)	iterations limit
minopf.verbosity	-1	amount of progress info printed [-1 - controlled by 'verbose' option] [0 - print nothing] [1 - print only termination status message] [2 - print termination status and screen progress] [3 - print screen progress, report file (usually fort.9)]
minopf.core	0 (1200*nb + 2*(nb+ng)^2)	memory allocation
minopf.supbasic_lim	0 (2*nb + 2*ng)	superbasics limit
minopf.mult_price	0 (30)	multiple price
MOSEK:		
mosek.lp_alg	0	solution algorithm (MSK_IPAR_OPTIMIZER) for MOSEK 8.x ... (see MOSEK_SYMBCON for a "better way") [0 - automatic: let MOSEK choose] [1 - dual simplex] [2 - automatic: let MOSEK choose] [3 - automatic simplex (MOSEK chooses which simplex method)] [4 - interior point] [6 - primal simplex]
mosek.max_it	0 (400)	interior point max iterations (MSK_IPAR_INTPT_MAX_ITERATIONS)
mosek.gap_tol	0 (1e-8)	interior point relative gap tol

(continues on next page)

(continued from previous page)

mosek.max_time	0 (-1)	(MSK_DPAR_INTPNT_TOL_REL_GAP) maximum time allowed
mosek.num_threads	0 (1)	(MSK_DPAR_OPTIMIZER_MAX_TIME) max number of threads
mosek.opts	<empty>	(MSK_IPAR_INTPNT_NUM_THREADS) see MOSEK_OPTIONS for details
mosek.opt_fname	<empty>	see MOSEK_OPTIONS for details
mosek.opt	0	see MOSEK_OPTIONS for details
OSQP:		
osqp.opts	<empty>	see OSQP_OPTIONS for details
QUADPROG:		
quadprog	<empty>	QUADPROG options passed to OPTIMOPTIONS or OPTIMSET. see QUADPROG in the Optimization Toolbox for details
TSPOPF:		
pdipm.feastol	0	feasibility (equality) tolerance (set to opf.violation by default)
pdipm.gradtol	1e-6	gradient tolerance
pdipm.comptol	1e-6	complementary condition (inequality) tolerance
pdipm.costtol	1e-6	optimality tolerance
pdipm.max_it	150	maximum number of iterations
pdipm.step_control	0	enable step-size cntrl [0 or 1]
pdipm.sc.red_it	20	maximum number of reductions per iteration with step control
pdipm.sc.smooth_ratio	0.04	piecewise linear curve smoothing ratio
tralm.feastol	0	feasibility tolerance (set to opf.violation by default)
tralm.primaltol	5e-4	primal variable tolerance
tralm.dualtol	5e-4	dual variable tolerance
tralm.costtol	1e-5	optimality tolerance
tralm.major_it	40	maximum number of major iterations
tralm.minor_it	40	maximum number of minor iterations
tralm.smooth_ratio	0.04	piecewise linear curve smoothing ratio
Experimental Options:		
exp.use_legacy_core	0	set to 1 to bypass MP-Core and force use of legacy core code for runpf(), runcpf(), runopf().
exp.sys_wide_zip_loads.pw	<empty>	1 x 3 vector of active load fraction to be modeled as constant power, constant current and constant impedance, respectively, where <empty> means use [1 0 0]
exp.sys_wide_zip_loads.qw	<empty>	same for reactive power, where

(continues on next page)

(continued from previous page)

<empty> means use same value as
for 'pw'

printf

printf(baseMVA, bus, gen, branch, f, success, et, fd, mpopt)

printf() - Prints power flow results.

```
PRINTF(RESULTS, FD, MPOPT)
```

```
PRINTF(BASEMVA, BUS, GEN, BRANCH, F, SUCCESS, ET, FD, MPOPT)
```

Prints power flow and optimal power flow results to FD (a file descriptor which defaults to STDOUT), with the details of what gets printed controlled by the optional MPOPT argument, which is a MATPOWER options struct (see MPOPTION for details).

The data can either be supplied in a single RESULTS struct, or in the individual arguments: BASEMVA, BUS, GEN, BRANCH, F, SUCCESS and ET, where F is the OPF objective function value, SUCCESS is true if the solution converged and false otherwise, and ET is the elapsed time for the computation in seconds. If F is given, it is assumed that the output is from an OPF run, otherwise it is assumed to be a simple power flow run.

Examples:

```
mpopt = mpoptions('out.gen', 1, 'out.bus', 0, 'out.branch', 0);
[fd, msg] = fopen(fname, 'at');
results = runopf(mpc);
printf(results);
printf(results, fd);
printf(results, fd, mpopt);
printf(baseMVA, bus, gen, branch, f, success, et);
printf(baseMVA, bus, gen, branch, f, success, et, fd);
printf(baseMVA, bus, gen, branch, f, success, et, fd, mpopt);
fclose(fd);
```

psse2mpc

psse2mpc(rawfile_name, mpc_name, verbose, rev)

psse2mpc() - Converts a PSS/E RAW data file into a MATPOWER case struct.

```
MPC = PSSE2MPC(RAWFILE_NAME)
MPC = PSSE2MPC(RAWFILE_NAME, VERBOSE)
MPC = PSSE2MPC(RAWFILE_NAME, VERBOSE, REV)
MPC = PSSE2MPC(RAWFILE_NAME, MPC_NAME)
MPC = PSSE2MPC(RAWFILE_NAME, MPC_NAME, VERBOSE)
MPC = PSSE2MPC(RAWFILE_NAME, MPC_NAME, VERBOSE, REV)
```

(continues on next page)

(continued from previous page)

```
[MPC, WARNINGS] = PSSE2MPC(RAWFILE_NAME, ...)
```

Converts a PSS/E RAW data file into a MATPOWER **case struct**.

Input:

```
RAWFILE_NAME : name of the PSS/E RAW file to be converted
                (opened directly with FILEREAD)
MPC_NAME      : (optional) file name to use to save the resulting
                MATPOWER case
VERBOSE       : 1 (default) to display progress info, 0 otherwise
REV           : (optional) assume the input file is of this
                PSS/E revision number, attempts to determine
                REV from the file by default
```

Output(s):

```
MPC           : resulting MATPOWER case struct
WARNINGS      : (optional) cell array of strings containing warning
                messages (included by default in comments of MPC_NAME).
```

NOTE: The data sections to be read in the PSS/E raw file includes:
 identification data; bus data; branch data; fixed shunt data;
 generator data; transformer data; switched shunt data; **area** data
 and hvdc **line** data. Other data sections are currently ignored.

save2psse

save2psse(*fname*, *mpc*, *rawver*)

save2psse() - Saves a MATPOWER case to PSS/E RAW format.

```
SAVE2PSSE(FNAME, MPC)
```

```
FNAME = SAVE2PSSE(FNAME, ...)
```

Saves a MATPOWER **case struct** MPC as a PSS/E RAW file. The FNAME parameter is a string containing the name of the file to be created or overwritten. If FNAME does **not** include a file extension, **'.raw'** will be added. Optionally returns the, possibly updated, filename. Currently exports to RAW format Rev **33**.

savecase**savecase**(*fname*, *varargin*)

savecase() - Saves a MATPOWER case file, given a filename and the data.

```

SAVECASE(FNAME, CASESTRUCT)
SAVECASE(FNAME, CASESTRUCT, VERSION)
SAVECASE(FNAME, BASEMVA, BUS, GEN, BRANCH)
SAVECASE(FNAME, BASEMVA, BUS, GEN, BRANCH, GENCOST)
SAVECASE(FNAME, BASEMVA, BUS, GEN, BRANCH, AREAS, GENCOST)
SAVECASE(FNAME, COMMENT, CASESTRUCT)
SAVECASE(FNAME, COMMENT, CASESTRUCT, VERSION)
SAVECASE(FNAME, COMMENT, BASEMVA, BUS, GEN, BRANCH)
SAVECASE(FNAME, COMMENT, BASEMVA, BUS, GEN, BRANCH, GENCOST)
SAVECASE(FNAME, COMMENT, BASEMVA, BUS, GEN, BRANCH, AREAS, GENCOST)

FNAME = SAVECASE(FNAME, ...)

```

Writes a MATPOWER **case** file, given a filename and data **struct** or list of data matrices. The FNAME parameter is the name of the file to be created or overwritten. If FNAME ends with **'*.mat*'** it saves the **case** as a MAT-file **otherwise** it saves it as an M-file. Optionally returns the filename, with extension added **if** necessary. The optional COMMENT argument is either string (**single line** comment) or a **cell** array of strings which are inserted as comments. When using a MATPOWER **case struct**, **if** the optional VERSION argument is **'1'** it will modify the data matrices to **version 1** format before saving.

savechgtab**savechgtab**(*fname*, *chgtab*, *warnings*)

savechgtab() - Save a change table to a file.

```

SAVECHGTAB(FNAME, CHGTAB)
SAVECHGTAB(FNAME, CHGTAB, WARNINGS)
FNAME = SAVECHGTAB(FNAME, ...)

```

Writes a CHGTAB, suitable **for** use with APPLY_CHANGES to a file specified by FNAME. If FNAME ends with **'*.mat*'** it saves CHGTAB and WARNINGS to a MAT-file as the variables **'*chgtab*'** and **'*warnings*'**, respectively. Otherwise, it saves an M-file **function** that returns the CHGTAB, with the optional WARNINGS in comments.

Optionally returns the filename, with extension added **if** necessary.

Input:

```

FNAME : name of the file to be saved
CHGTAB : change table suitable for use with APPLY_CHANGES
WARNINGS : optional cell array of warning messages (to be
            included in comments), such as those returned by

```

(continues on next page)

(continued from previous page)

PSSECON2CHGTAB

Output(s):

FNAME : name of the file, with extension added **if** necessary

5.2.3 Data Conversion Functions

ext2int

ext2int(bus, gen, branch, areas)

ext2int() - Converts external to internal indexing.

This **function** has two forms, (1) the old form that operates on **and** returns individual matrices **and** (2) the new form that operates on **and** returns an entire MATPOWER **case struct**.

1. [I2E, BUS, GEN, BRANCH, AREAS] = EXT2INT(BUS, GEN, BRANCH, AREAS)
[I2E, BUS, GEN, BRANCH] = EXT2INT(BUS, GEN, BRANCH)

If the first argument is a matrix, it simply converts from (possibly non-consecutive) external bus numbers to consecutive internal bus numbers which start at 1. Changes are made to BUS, GEN **and** BRANCH, which are returned along with a vector of indices I2E that can be passed to INT2EXT to perform the reverse conversion, where `EXTERNAL_BUS_NUMBER = I2E(INTERNAL_BUS_NUMBER)`. AREAS is completely ignored **and** is only included here **for** backward compatibility of the API.

Examples:

```
[i2e, bus, gen, branch, areas] = ext2int(bus, gen, branch, areas);
[i2e, bus, gen, branch] = ext2int(bus, gen, branch);
```

2. MPC = EXT2INT(MPC)
MPC = EXT2INT(MPC, MPOPT)

If the **input** is a **single** MATPOWER **case struct**, followed optionally by a MATPOWER options **struct**, then **all** isolated buses, off-line generators **and** branches are removed along with **any** generators **or** branches connected to isolated buses. Then the buses are renumbered consecutively, beginning at 1. Any '**ext2int**' callback routines registered in the **case** are also invoked automatically. All of the related indexing information **and** the original data matrices are stored in an '**order**' field in the **struct** to be used by INT2EXT to perform the reverse conversions. If the **case** is already using internal numbering it is returned unchanged.

Examples:

```
mpc = ext2int(mpc);
```

(continues on next page)

(continued from previous page)

```
mpc = ext2int(mpc, mpopt);
```

The 'order' field of MPC used to store the indexing information needed for subsequent internal to external conversion is structured as:

```
order
  state      'i' | 'e'
  ext | int
    bus
    branch
    gen
    gencost
    A
    N
  bus
    e2i
    i2e
    status
      on
      off
  gen
    e2i
    i2e
    status
      on
      off
  branch
    status
      on
      off
```

See also `int2ext()`, `e2i_field()` (page 258), `e2i_data()` (page 257).

e2i_data

e2i_data(mpc, val, ordering, dim)

`e2i_data()` (page 257) - Converts data from external to internal indexing.

```
VAL = E2I_DATA(MPC, VAL, ORDERING)
VAL = E2I_DATA(MPC, VAL, ORDERING, DIM)
```

When given a **case struct** that has already been converted to internal indexing, this **function** can be used to convert other data structures as well by passing in 2 or 3 extra parameters in addition to the **case struct**. If the value passed in the 2nd argument is a column vector or cell array, it will be converted according to the ORDERING specified by the 3rd argument (described below). If VAL is an n-dimensional matrix or cell array, then the optional 4th argument (DIM, default = 1) can be used to specify

(continues on next page)

(continued from previous page)

which dimension to reorder. The **return** value in this **case** is the value passed in, converted to internal indexing.

The 3rd argument, ORDERING, is used to indicate whether the data corresponds to bus-, gen- or branch-ordered data. It can be one of the following three strings: 'bus', 'gen' or 'branch'. For data structures with multiple blocks of data, ordered by bus, gen or branch, they can be converted with a **single** call by specifying ORDERING as a **cell** array of strings.

Any extra elements, rows, columns, etc. beyond those indicated in ORDERING, are **not** disturbed.

Examples:

```
A_int = e2i_data(mpc, A_ext, {'bus','bus','gen','gen'}, 2);
```

Converts an A matrix **for** user-supplied OPF constraints from external to internal ordering, where the **columns** of the A matrix correspond to bus voltage angles, then voltage magnitudes, then generator **real power** injections and finally generator reactive **power** injections.

```
gencost_int = e2i_data(mpc, gencost_ext, {'gen','gen'}, 1);
```

Converts a GENCOST matrix that has both **real** and reactive **power** costs (in **rows** 1--ng and ng+1--2*ng, respectively).

See also [i2e_data\(\)](#) (page 260), [e2i_field\(\)](#) (page 258), ext2int().

e2i_field

e2i_field(mpc, field, ordering, dim)

[e2i_field\(\)](#) (page 258) - Converts fields of mpc from external to internal indexing.

This **function** performs several different tasks, depending on the arguments passed.

```
MPC = E2I_FIELD(MPC, FIELD, ORDERING)
MPC = E2I_FIELD(MPC, FIELD, ORDERING, DIM)
```

When given a **case struct** that has already been converted to internal indexing, this **function** can be used to convert other data structures as well by passing in 2 or 3 extra parameters in addition to the **case struct**.

The 2nd argument is a string or **cell** array of strings, specifying a field in the **case struct** whose value should be converted by a corresponding call to E2I_DATA. The field can contain either a numeric or a **cell** array. The converted value is stored back in the specified field, the original value is saved **for** later use and the

(continues on next page)

(continued from previous page)

updated **case struct** is returned. If **FIELD** is a **cell** array of strings, they specify nested fields.

The 3rd and optional 4th arguments are simply passed along to the call to **E2I_DATA**.

Examples:

```
mpc = e2i_field(mpc, {'reserves', 'cost'}, 'gen');
```

Reorders **rows** of **mpc.reserves.cost** to match internal generator ordering.

```
mpc = e2i_field(mpc, {'reserves', 'zones'}, 'gen', 2);
```

Reorders **columns** of **mpc.reserves.zones** to match internal generator ordering.

See also *i2e_field()* (page 261), *e2i_data()* (page 257), *ext2int()*.

int2ext

int2ext(*i2e, bus, gen, branch, areas*)

int2ext() - Converts internal to external bus numbering.

This **function** has two forms, (1) the old form that operates on **and** returns individual matrices **and** (2) the new form that operates on **and** returns an entire MATPOWER **case struct**.

1. **[BUS, GEN, BRANCH, AREAS] = INT2EXT(I2E, BUS, GEN, BRANCH, AREAS)**
[BUS, GEN, BRANCH] = INT2EXT(I2E, BUS, GEN, BRANCH)

Converts from the consecutive internal bus numbers back to the originals using the mapping provided by the **I2E** vector returned from **EXT2INT**, where **EXTERNAL_BUS_NUMBER = I2E(INTERNAL_BUS_NUMBER)**.

AREAS is completely ignored **and** is only included here **for** backward compatibility of the API.

Examples:

```
[bus, gen, branch, areas] = int2ext(i2e, bus, gen, branch, areas);
[bus, gen, branch] = int2ext(i2e, bus, gen, branch);
```

2. **MPC = INT2EXT(MPC)**
MPC = INT2EXT(MPC, MPOPT)

If the **input** is a **single** MATPOWER **case struct**, followed optionally by a MATPOWER options **struct**, then it restores **all** buses, generators **and** branches that were removed because of being isolated **or** off-line, **and** reverts to the original generator ordering **and** original bus numbering. This requires that the **'order'** field created by **EXT2INT** be in place.

(continues on next page)

(continued from previous page)

Examples:

```
mpc = int2ext(mpc);
mpc = int2ext(mpc, mpopt);
```

See also `ext2int()`, `i2e_field()` (page 261), `i2e_data()` (page 260).

i2e_data

i2e_data(mpc, val, oldval, ordering, dim)

`i2e_data()` (page 260) - Converts data from internal to external indexing.

```
VAL = I2E_DATA(MPC, VAL, OLDVAL, ORDERING)
VAL = I2E_DATA(MPC, VAL, OLDVAL, ORDERING, DIM)
```

For a **case struct** using internal indexing, this **function** can be used to convert other data structures as well by passing in 3 or 4 extra parameters in addition to the **case struct**. If the value passed in the 2nd argument (VAL) is a column vector or cell array, it will be converted according to the ordering specified by the 4th argument (ORDERING, described below). If VAL is an n-dimensional matrix or cell array, then the optional 5th argument (DIM, default = 1) can be used to specify which dimension to reorder. The 3rd argument (OLDVAL) is used to initialize the **return** value before converting VAL to external indexing. In particular, any data corresponding to off-line gens or branches or isolated buses or any connected gens or branches will be taken from OLDVAL, with VAL supplying the rest of the returned data.

The ORDERING argument is used to indicate whether the data corresponds to bus-, gen- or branch-ordered data. It can be one of the following three strings: 'bus', 'gen' or 'branch'. For data structures with multiple blocks of data, ordered by bus, gen or branch, they can be converted with a single call by specifying ORDERING as a cell array of strings.

Any extra elements, rows, columns, etc. beyond those indicated in ORDERING, are not disturbed.

Examples:

```
A_ext = i2e_data(mpc, A_int, A_orig, {'bus','bus','gen','gen'}, 2);
```

Converts an A matrix for user-supplied OPF constraints from internal to external ordering, where the columns of the A matrix correspond to bus voltage angles, then voltage magnitudes, then generator real power injections and finally generator reactive power injections.

```
gencost_ext = i2e_data(mpc, gencost_int, gencost_orig, {'gen','gen'}, 1);
```

(continues on next page)

(continued from previous page)

Converts a GENCO matrix that has both **real** and reactive **power** costs (in **rows** 1--ng and ng+1--2*ng, respectively).

See also `e2i_data()` (page 257), `i2e_field()` (page 261), `int2ext()`.

i2e_field

i2e_field(mpc, field, ordering, dim)

`i2e_field()` (page 261) - Converts fields of mpc from internal to external bus numbering.

```
MPC = I2E_FIELD(MPC, FIELD, ORDERING)
MPC = I2E_FIELD(MPC, FIELD, ORDERING, DIM)
```

For a **case struct** using internal indexing, this **function** can be used to convert other data structures as well by passing in 2 or 3 extra parameters in addition to the **case struct**.

The 2nd argument is a string or cell array of strings, specifying a field in the **case struct** whose value should be converted by a corresponding call to `I2E_DATA`. The field can contain either a numeric or a cell array. The corresponding OLDVAL is taken from where it was stored by `EXT2INT` in `MPC.ORDER.EXT` and the updated **case struct** is returned. If `FIELD` is a cell array of strings, they specify nested fields.

The 3rd and optional 4th arguments are simply passed along to the call to `I2E_DATA`.

Examples:

```
mpc = i2e_field(mpc, {'reserves', 'cost'}, 'gen');
```

Reorders **rows** of `mpc.reserves.cost` to match external generator ordering.

```
mpc = i2e_field(mpc, {'reserves', 'zones'}, 'gen', 2);
```

Reorders **columns** of `mpc.reserves.zones` to match external generator ordering.

See also `e2i_field()` (page 258), `i2e_data()` (page 260), `int2ext()`.

get_reorder

get_reorder(A, idx, dim)

get_reorder() (page 262) - Returns A with one of its dimensions indexed.

```
B = GET_REORDER(A, IDX, DIM)
```

Returns A(:, ..., :, IDX, :, ..., :), where DIM determines in which dimension to place the IDX.

See also *set_reorder()* (page 262).

set_reorder

set_reorder(A, B, idx, dim)

set_reorder() (page 262) - Assigns B to A with one of the dimensions of A indexed.

```
A = SET_REORDER(A, B, IDX, DIM)
```

Returns A after doing A(:, ..., :, IDX, :, ..., :) = B where DIM determines in which dimension to place the IDX.

If any dimension of B is smaller than the corresponding dimension of A, the "extra" elements in A are untouched. If any dimension of B is larger than the corresponding dimension of A, then A is padded with zeros (if numeric) or empty matrices (if cell array) before performing the assignment.

See also *get_reorder()* (page 262).

5.2.4 Power Flow Functions

calc_v_i_sum

calc_v_i_sum(Vslack, nb, nl, f, Zb, Ybf, Ybt, Yd, Sd, pv, Pg, Vg, mpopt)

calc_v_i_sum() (page 262) - Solves the power flow using the current summation method.

```
[V, Qpv, Sf, St, Sslack, iter, success] = calc_v_i_sum(Vslack, nb, nl, f, Zb, Ybf, Ybt, Yd,
↳ Sd, pv, Pg, Vg, tol, iter_max)
```

Solves for bus voltages, generator reactive power, branch active and reactive power flows and slack bus active and reactive power. The input data consist of slack bus voltage, vector "from bus" indices, branch impedance and shunt admittance, vector of bus shunt admittances and load demand, as well as vectors with indices of PV buses with their specified voltages and active powers. It is assumed that the branches are ordered using the principle of oriented ordering: indices of

(continues on next page)

(continued from previous page)

sending nodes are smaller than the indices of the receiving nodes. The branch `index` is equal to the `index` of their receiving node. Branch admittances are added in `Yd` and treated as constant admittance bus loads. The applied method is current summation taken from:

D. Shirmohammadi, H. W. Hong, A. Semlyen and G. X. Luo, "A compensation-based power flow method for weakly meshed distribution and transmission networks," IEEE Transactions on Power Systems, vol. 3, no. 2, pp. 753-762, May 1988. <https://doi.org/10.1109/59.192932> and G. X. Luo and A. Semlyen, "Efficient load flow for large weakly meshed networks," IEEE Transactions on Power Systems, vol. 5, no. 4, pp. 1309-1316, Nov 1990. <https://doi.org/10.1109/59.99382>

See also `radial_pf()` (page 271).

`calc_v_pq_sum`

`calc_v_pq_sum(Vslack, nb, nl, f, Zb, Ybf, Ybt, Yd, Sd, pv, Pg, Vg, mpopt)`

`calc_v_pq_sum()` (page 263) - Solves the power flow using the power summation method.

```
[V, Qpv, Sf, St, Sslack, iter, success] = calc_v_pq_sum(Vslack, nb, nl, f, Zb, Ybf, Ybt, ,
↪ Yd, Sd, pv, Pg, Vg, tol, iter_max)
```

Solves **for** bus voltages, generator reactive **power**, branch active and reactive **power** flows and slack bus active and reactive **power**. The input data consist of slack bus voltage, vector "from bus" indices, branch impedance and shunt admittance, vector of bus shunt admittances and load demand, as well as vectors with indices of PV buses with their specified voltages and active powers. It is assumed that the branches are ordered using the principle of oriented ordering: indices of sending nodes are smaller than the indices of the receiving nodes. The branch `index` is equal to the `index` of their receiving node. Branch admittances are added in `Yd` and treated as constant admittance bus loads. The applied method is Voltage correction **power** flow (VCPF) taken from:

D. Rajicic, R. Ackovski and R. Taleski, "Voltage correction power flow," IEEE Transactions on Power Delivery, vol. 9, no. 2, pp. 1056-1062, Apr 1994. <https://doi.org/10.1109/61.296308>

See also `radial_pf()` (page 271).

calc_v_y_sum**calc_v_y_sum**(*Vslack, nb, nl, f, Zb, Ybf, Ybt, Yd, Sd, pv, Pg, Vg, mpopt*)*calc_v_y_sum*() (page 264) - Solves the power flow using the admittance summation method.

```
[V, Qpv, Sf, St, Sslack, iter, success] = calc_v_y_sum(Vslack,nb,nl,f,Zb,Ybf,Ybt,Yd,
↪Sd,pv,Pg,Vg,tol,iter_max)
```

Solves **for** bus voltages, generator reactive **power**, branch active **and** reactive **power** flows **and** slack bus active **and** reactive **power**. The **input** data consist of slack bus voltage, vector "**from bus**" indices, branch impedance **and** shunt admittance, vector of bus shunt admittances **and** **load** demand, as well as vectors with indices of PV buses with their specified voltages **and** active powers. It is assumed that the branches are ordered using the principle of oriented ordering: indices of sending nodes are smaller than the indices of the receiving nodes. The branch **index** is equal to the **index** of their receiving node. Branch admittances are added in *Yd* **and** treated as constant admittance bus loads. The applied method is admittance summation taken from: Dragoslav Rajičić, Rubin Taleski, Two novel **methods for** radial **and** weakly meshed network analysis, Electric Power Systems Research, Volume 48, Issue 2, 15 December 1998, Pages 79-87
[https://doi.org/10.1016/S0378-7796\(98\)00067-4](https://doi.org/10.1016/S0378-7796(98)00067-4)

See also *radial_pf*() (page 271).

dcpf**dcpf**(*B, Pbus, Va0, ref, pv, pq*)*dcpf*() - Solves a DC power flow.

[VA, SUCCESS] = DCPF(B, PBUS, VA0, REF, PV, PQ) solves **for** the bus voltage angles at **all** but the reference bus, given the **full system** B matrix **and** the vector of bus **real power** injections, the initial vector of bus voltage angles (in radians), **and** column vectors with the lists of bus indices **for** the swing bus, PV buses, **and** PQ buses, respectively. Returns a vector of bus voltage angles in radians.

See also *rundcpf*(), *runpf*() .

fdpf**fdpf**(*Ybus, Sbus, V0, Bp, Bpp, ref, pv, pq, mpop*)

fdpf() - Solves the power flow using a fast decoupled method.

[V, CONVERGED, I] = FDPF(YBUS, SBUS, V0, BP, BPP, REF, PV, PQ, MPOPT) solves **for** bus voltages given the **full system** admittance matrix (**for all** buses), the **complex** bus **power** injection vector (**for all** buses), the initial vector of **complex** bus voltages, the FDPF matrices B prime and B double prime, and column vectors with the lists of bus indices **for** the swing bus, PV buses, and PQ buses, respectively. The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, and the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes and angles. MPOPT is a MATPOWER options vector which can be used to **set** the termination tolerance, maximum number of iterations, and output options (see MPOPTION **for** details). Uses default options **if** this parameter is **not** given. Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, and the number of iterations performed.

See also runpf().

gausspf**gausspf**(*Ybus, Sbus, V0, ref, pv, pq, mpop*)

gausspf() - Solves the power flow using a Gauss-Seidel method.

[V, CONVERGED, I] = GAUSSPF(YBUS, SBUS, V0, REF, PV, PQ, MPOPT) solves **for** bus voltages given the **full system** admittance matrix (**for all** buses), the **complex** bus **power** injection vector (**for all** buses), the initial vector of **complex** bus voltages, and column vectors with the lists of bus indices **for** the swing bus, PV buses, and PQ buses, respectively. The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, and the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes and angles. MPOPT is a MATPOWER options **struct** which can be used to **set** the termination tolerance, maximum number of iterations, and output options (see MPOPTION **for** details). Uses default options **if** this parameter is **not** given. Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, and the number of iterations performed.

See also runpf().

make_vcorr

make_vcorr(*DD, pv, nb, nl, f, Zb*)

make_vcorr() (page 266) - Voltage Correction used in distribution power flow.

```
V_corr = make_vcorr(DD,pv,nb,nl,f,Zb)
```

Calculates voltage corrections with current generators placed at PV buses. Their currents are calculated with the voltage difference at PV buses **break** points and loop impedances. The slack bus voltage is **set** to zero. Details can be seen in D. Rajicic, R. Ackovski and R. Taleski, "Voltage correction power flow," IEEE Transactions on Power Delivery, vol. 9, no. 2, pp. 1056-1062, Apr 1994. <https://doi.org/10.1109/61.296308>

See also *radial_pf()* (page 271).

make_zpv

make_zpv(*pv, nb, nl, f, Zb, Yd*)

make_zpv() (page 266) - Calculates loop impedances for all PV buses.

```
Zpv = make_zpv(pv,nb,nl,f,Zb,Yd)
```

Loop impedance of a PV bus is defined as impedance of the **path** between the bus and the slack bus. The mutual impedance between two PV buses is the impedance of the joint part of the two **path** going from each of the PV buses to the slack bus. The impedances are calculated as bus voltages in cases when at one of the PV buses we inject current of -1 A. All voltages are calculated with the backward-forward sweep method. The **input** variables are the vector of indicies with "from" buses **for** each branch, the vector of branch impedances and indicies of PV buses.

See also *calc_v_pq_sum()* (page 263).

newtonpf

newtonpf(*Ybus, Sbus, V0, ref, pv, pq, mpopt*)

newtonpf() - Solves power flow using full Newton's method (power/polar).

```
[V, CONVERGED, I] = NEWTONPF(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)
```

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal **power balance** equations and **polar** coordinate representation of voltages, given the following inputs:

- YBUS - **full system** admittance matrix (**for all** buses)
- SBUS - handle to **function** that returns the **complex** bus **power** injection vector (**for all** buses), given the bus voltage magnitude vector (**for all** buses)

(continues on next page)

(continued from previous page)

V0 - initial vector of **complex** bus voltages
REF - bus **index** of reference bus (voltage ang reference & gen slack)
PV - vector of bus indices **for** PV buses
PQ - vector of bus indices **for** PQ buses
MPOPT - (optional) MATPOWER option **struct**, used to **set** the termination tolerance, maximum number of iterations, **and** output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, **and** the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes **and** angles.

Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, **and** the number of iterations performed.

See also `runpf()`, `newtonpf_S_cart()` (page 269), `newtonpf_I_polar()` (page 268), `newtonpf_I_cart()` (page 267).

newtonpf_I_cart

newtonpf_I_cart(*Ybus, Sbus, V0, ref, pv, pq, mpopt*)

`newtonpf_I_cart()` (page 267) - Solves power flow using full Newton's method (current/cartesian).

```
[V, CONVERGED, I] = NEWTONPF_I_CART(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)
```

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal current **balance** equations **and** cartesian coordinate representation of voltages, given the following inputs:

YBUS - **full system** admittance matrix (**for all** buses)
SBUS - handle to **function** that returns the **complex** bus **power** injection vector (**for all** buses), given the bus voltage magnitude vector (**for all** buses)
V0 - initial vector of **complex** bus voltages
REF - bus **index** of reference bus (voltage ang reference & gen slack)
PV - vector of bus indices **for** PV buses
PQ - vector of bus indices **for** PQ buses
MPOPT - (optional) MATPOWER option **struct**, used to **set** the termination tolerance, maximum number of iterations, **and** output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, **and** the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes **and** angles.

Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, **and** the number of iterations performed.

See also `runpf()`, `newtonpf()`, `newtonpf_S_cart()` (page 269), `newtonpf_I_polar()` (page 268).

newtonpf_I_hybrid

newtonpf_I_hybrid(*Ybus, Sbus, V0, ref, pv, pq, mpopt*)

newtonpf_I_hybrid() (page 268) - Solves power flow using full Newton's method (current/hybrid).

```
[V, CONVERGED, I] = NEWTONPF_I_HYBRID(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)
```

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal current **balance** equations **and** a hybrid representation of voltages, where a **polar** update is computed using a cartesian Jacobian, given the following inputs:

- YBUS - **full system** admittance matrix (**for all** buses)
- SBUS - handle to **function** that returns the **complex** bus **power** injection vector (**for all** buses), given the bus voltage magnitude vector (**for all** buses)
- V0 - initial vector of **complex** bus voltages
- REF - bus **index** of reference bus (voltage ang reference & gen slack)
- PV - vector of bus indices **for** PV buses
- PQ - vector of bus indices **for** PQ buses
- MPOPT - (optional) MATPOWER option **struct**, used to **set** the termination tolerance, maximum number of iterations, **and** output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, **and** the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes **and** angles.

Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, **and** the number of iterations performed.

See also `runpf()`, `newtonpf()`, *newtonpf_S_cart()* (page 269), *newtonpf_I_polar()* (page 268).

newtonpf_I_polar

newtonpf_I_polar(*Ybus, Sbus, V0, ref, pv, pq, mpopt*)

newtonpf_I_polar() (page 268) - Solves power flow using full Newton's method (current/cartesian).

```
[V, CONVERGED, I] = NEWTONPF_I_POLAR(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)
```

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal current **balance** equations **and** **polar** coordinate representation of voltages, given the following inputs:

- YBUS - **full system** admittance matrix (**for all** buses)
- SBUS - handle to **function** that returns the **complex** bus **power** injection vector (**for all** buses), given the bus voltage magnitude vector (**for all** buses)
- V0 - initial vector of **complex** bus voltages

(continues on next page)

(continued from previous page)

REF - bus **index** of reference bus (voltage ang reference & gen slack)
 PV - vector of bus indices **for** PV buses
 PQ - vector of bus indices **for** PQ buses
 MPOPT - (optional) MATPOWER option **struct**, used to **set** the
 termination tolerance, maximum number of iterations, **and**
 output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator
 (including ref bus) buses, **and** the reference **angle** of the swing
 bus, as well as an initial guess **for** remaining magnitudes **and**
 angles.

Returns the final **complex** voltages, a **flag** which indicates whether it
 converged **or not**, **and** the number of iterations performed.

See also `runpf()`, `newtonpf()`, `newtonpf_S_cart()` (page 269), `newtonpf_I_cart()` (page 267).

newtonpf_S_cart

newtonpf_S_cart(Ybus, Sbus, V0, ref, pv, pq, mpopt)

`newtonpf_S_cart()` (page 269) - Solves power flow using full Newton's method (power/cartesian).

[V, CONVERGED, I] = NEWTONPF_S_CART(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal
power balance equations **and** cartesian coordinate representation of
 voltages, given the following inputs:

YBUS - **full system** admittance matrix (**for all** buses)
 SBUS - handle to **function** that returns the **complex** bus **power**
 injection vector (**for all** buses), given the bus voltage
 magnitude vector (**for all** buses)
 V0 - initial vector of **complex** bus voltages
 REF - bus **index** of reference bus (voltage ang reference & gen slack)
 PV - vector of bus indices **for** PV buses
 PQ - vector of bus indices **for** PQ buses
 MPOPT - (optional) MATPOWER option **struct**, used to **set** the
 termination tolerance, maximum number of iterations, **and**
 output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator
 (including ref bus) buses, **and** the reference **angle** of the swing
 bus, as well as an initial guess **for** remaining magnitudes **and**
 angles.

Returns the final **complex** voltages, a **flag** which indicates whether it
 converged **or not**, **and** the number of iterations performed.

See also `runpf()`, `newtonpf()`, `newtonpf_I_polar()` (page 268), `newtonpf_I_cart()` (page 267).

newtonpf_S_hybrid

newtonpf_S_hybrid(Ybus, Sbus, V0, ref, pv, pq, mpop)

[newtonpf_S_hybrid\(\)](#) (page 270) - Solves power flow using full Newton's method (power/hybrid).

```
[V, CONVERGED, I] = NEWTONPF_S_HYBRID(YBUS, SBUS, V0, REF, PV, PQ, MPOPT)
```

Solves **for** bus voltages using a **full** Newton-Raphson method, using nodal **power balance** equations **and** a hybrid representation of voltages, where a **polar** update is computed using a cartesian Jacobian, given the following inputs:

- YBUS - **full system** admittance matrix (**for all** buses)
- SBUS - handle to **function** that returns the **complex** bus **power** injection vector (**for all** buses), given the bus voltage magnitude vector (**for all** buses)
- V0 - initial vector of **complex** bus voltages
- REF - bus **index** of reference bus (voltage ang reference & gen slack)
- PV - vector of bus indices **for** PV buses
- PQ - vector of bus indices **for** PQ buses
- MPOPT - (optional) MATPOWER option **struct**, used to **set** the termination tolerance, maximum number of iterations, **and** output options (see MPOPTION **for** details).

The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, **and** the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes **and** angles.

Returns the final **complex** voltages, a **flag** which indicates whether it converged **or not**, **and** the number of iterations performed.

See also [runpf\(\)](#), [newtonpf\(\)](#), [newtonpf_I_polar\(\)](#) (page 268), [newtonpf_I_cart\(\)](#) (page 267).

order_radial

order_radial(mpc)

[order_radial\(\)](#) (page 270) - Performs oriented ordering to buses and branches.

```
mpc = order_radial(mpc)
```

orders the branches by the the principle of oriented ordering: indicies of sending nodes are smaller than the indicies of the receiving nodes. The branch **index** is equal to the **index** of their receiving node. The ordering is taken from:
D. Rajicic, R. Ackovski **and** R. Taleski, "Voltage correction power flow," IEEE Transactions on Power Delivery, vol. 9, no. 2, pp. 1056-1062, Apr 1994.

See also [radial_pf\(\)](#) (page 271).

pfsoln

pfsoln(baseMVA, bus0, gen0, branch0, Ybus, Yf, Yt, V, ref, pv, pq, mpopt)

pfsoln() - Updates bus, gen, branch data structures to match power flow soln.

```
[BUS, GEN, BRANCH] = PFSOLN(BASEMVA, BUS0, GEN0, BRANCH0, ...
                             YBUS, YF, YT, V, REF, PV, PQ, MPOPT)
```

radial_pf

radial_pf(mpc, mpopt)

radial_pf() (page 271) - Solves the power flow using a backward-forward sweep method.

```
[mpc, success, iterations] = radial_pf(mpc,mpopt)
```

Inputs:

mpc : MATPOWER **case struct** with internal bus numbering
 mpopt : MATPOWER options **struct** to override default options
 can be used to specify the solution algorithm, output options
 termination tolerances, **and** more.

Outputs:

mpc : results **struct** with **all** fields from the **input** MATPOWER **case**,
 with solved voltages, active **and** reactive **power** flows
 and generator active **and** reactive **power** output.
 success : success **flag**, 1 = succeeded, 0 = failed
 iterations : number of iterations

See also caseformat, loadcase(), mpoption().

zgausspf

zgausspf(Ybus, Sbus, V0, ref, pv, pq, Bpp, mpopt)

zgausspf() - Solves the power flow using an Implicit Z-bus Gauss method.

```
[V, CONVERGED, I] = ZGAUSSPF(YBUS, SBUS, V0, REF, PV, PQ, BPP, MPOPT)
```

solves **for** bus voltages given the **full system** admittance matrix (**for** **all** buses), the **complex** bus **power** injection vector (**all** buses), the initial vector of **complex** bus voltages, column vectors with the lists of bus indices **for** the swing bus, PV buses, **and** PQ buses, respectively, **and** the fast-decoupled B **double-prime** matrix (**all** buses) **for** Q updates at PV buses. The bus voltage vector contains the **set** point **for** generator (including ref bus) buses, **and** the reference **angle** of the swing bus, as well as an initial guess **for** remaining magnitudes **and** angles. MPOPT is a MATPOWER options **struct** which can be used to **set** the termination tolerance, maximum number of iterations, **and** output options (see MPOPTION **for** details). Uses default options **if** this parameter is **not** given. Returns the final **complex** voltages,

(continues on next page)

(continued from previous page)

a **flag** which indicates whether it converged **or not**, **and** the number of iterations performed.

NOTE: This method does **not** scale well with the number of generators **and** seems to have serious problems with some systems with many PV buses.

See also `runpf()`.

5.2.5 Continuation Power Flow Functions

`cpf_corrector`

cpf_corrector(*Ybus, Sbusb, V_hat, ref, pv, pq, lam_hat, Sbusb, Vprv, lamprv, z, step, parameterization, mpopt*)
cpf_corrector() (page 272) - Solves the corrector step of a continuation power flow.

```
[V, CONVERGED, I, LAM] = CPF_CORRECTOR(YBUS, SBUSB, V_HAT, REF, PV, PQ, ...
                                         LAM_HAT, SBUST, VPRV, LPRV, Z, ...
                                         STEP, PARAMETERIZATION, MPOPT)
```

Computes the corrector step of a continuation **power** flow using a **full** Newton method with selected parameterization scheme.

Inputs:

YBUS : **complex** bus admittance matrix
 SBUSB : handle of **function** returning nb x 1 vector of **complex**
 base **case** injections in p.u. **and** derivatives w.r.t. |V|
 V_HAT : predicted **complex** bus voltage vector
 REF : vector of indices **for** REF buses
 PV : vector of indices of PV buses
 PQ : vector of indices of PQ buses
 LAM_HAT : predicted scalar lambda
 SBUST : handle of **function** returning nb x 1 vector of **complex**
 target **case** injections in p.u. **and** derivatives w.r.t. |V|
 VPRV : **complex** bus voltage vector at previous solution
 LAMPRV : scalar lambda value at previous solution
 STEP : continuation step **length**
 Z : normalized tangent prediction vector
 STEP : continuation step **size**
 PARAMETERIZATION : Value of `cpf.parameterization` option.
 MPOPT : Options **struct**

Outputs:

V : **complex** bus voltage solution vector
 CONVERGED : Newton iteration count
 I : Newton iteration count
 LAM : lambda continuation parameter

See also `runcpf()`.

cpf_current_mpc

cpf_current_mpc(*mpc, mpct, Ybus, Yf, Yt, ref, pv, pq, V, lam, mpopt*)

cpf_current_mpc() (page 273) - Construct *mpc* for current continuation step.

```
MPC = CPF_CURRENT_MPC(MPC_BASE, MPC_TARGET, YBUS, YF, YT, REF, PV, PQ, V, LAM,
↳ MPOPT)
```

Constructs the MATPOWER **case struct** for the current continuation step based on the MPC_BASE and MPC_TARGET cases and the value of LAM.

cpf_default_callback

cpf_default_callback(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

cpf_default_callback() (page 273) - Default callback function for CPF.

```
[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =
CPF_DEFAULT_CALLBACK(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...
CB_DATA, CB_ARGS, RESULTS)
```

Default callback **function** used by RUNCPF that collects the results and optionally, plots the nose curve. Inputs and outputs are defined below, with the RESULTS argument being optional, used only **for** the final call when K is negative.

Inputs:

- K - continuation step iteration count
- NX - next state (corresponding to proposed next step), **struct** with the following fields:
 - lam_hat - value of LAMBDA from predictor
 - V_hat - vector of **complex** bus voltages from predictor
 - lam - value of LAMBDA from corrector
 - V - vector of **complex** bus voltages from corrector
 - z - normalized tangent predictor
 - default_step - default step **size**
 - default_parm - default parameterization
 - this_step - step **size** **for** this step only
 - this_parm - parameterization **for** this step only
 - step - current step **size**
 - parm - current parameterization
 - events** - **struct** array, event **log**
 - cb - user state, **for** callbacks (replaces CB_STATE), the user may add fields containing **any** information the callback **function** would like to pass from one invocation to the next, taking care **not** to step on fields being used by other callbacks, such as the 'default' field used by this default callback
 - ef - **cell** array of event **function** values
- CX - current state (corresponding to most recent successful step)

(continues on next page)

(continued from previous page)

(same structure as NX)

PX - previous state (corresponding to last successful step prior to CX)

DONE - **struct**, with **flag** to indicate CPF termination and reason, with fields:

- flag** - termination **flag**, 1 => terminate, 0 => **continue**
- msg** - string containing reason **for** termination

ROLLBACK - scalar **flag** to indicate that the current step should be rolled back and retried with a different step **size**, etc.

EVNTS - **struct** array listing **any events** detected **for** this step, see CPF_DETECT_EVENTS **for** details

CB_DATA - **struct** containing potentially useful "static" data, with the following fields (all based on internal indexing):

- mpc_base** - MATPOWER **case struct** of base state
- mpc_target** - MATPOWER **case struct** of target state
- Sbusb** - handle of **function** returning nb x 1 vector of **complex** base **case** injections in p.u. and derivatives w.r.t. |V|
- Sbust** - handle of **function** returning nb x 1 vector of **complex** target **case** injections in p.u. and derivatives w.r.t. |V|
- Ybus** - bus admittance matrix
- Yf** - branch admittance matrix, "from" **end** of branches
- Yt** - branch admittance matrix, "to" **end** of branches
- pv** - vector of indices of PV buses
- pq** - vector of indices of PQ buses
- ref** - vector of indices of REF buses
- idx_pmax** - vector of generator indices **for** generators fixed at their PMAX limits
- mpopt** - MATPOWER options **struct**

CB_ARGS - arbitrary data structure containing callback arguments

RESULTS - initial value of output **struct** to be assigned to CPF field of results **struct** returned by RUNCPF

Outputs:

(all are updated versions of the corresponding **input** arguments)

NX - user state ('cb' field) should be updated here **if** ROLLBACK is **false**

CX - may contain updated 'this_step' or 'this_parm' values to be used **if** ROLLBACK is **true**

DONE - callback may have requested termination and **set** the msg field

ROLLBACK - callback can request a rollback step, even **if** it was **not** indicated by an event **function**

EVNTS - msg field **for** a given event may be updated

CB_DATA - this data should only be modified **if** the underlying problem has been changed (e.g. generator limit reached) and should always be followed by a step of zero **length**, i.e. **set** NX.this_step to 0
It is the job of **any** callback modifying CB_DATA to ensure that **all** data in CB_DATA is kept consistent.

RESULTS - updated **version** of RESULTS **input arg**

This **function** is called in three different contexts, distinguished by the value of K, as follows:

- (1) initial - called with K = 0, without RESULTS **input/output** args, after base **power** flow, before 1st CPF step.

(continues on next page)

(continued from previous page)

- (2) iterations - called with $K > 0$, without RESULTS input/output args, at each iteration, after predictor-corrector step
- (3) final - called with $K < 0$, with RESULTS input/output args, after exiting predictor-corrector loop, inputs identical to last iteration call, except K which is negated

User Defined CPF Callback Functions:

The user can define their own callback functions which take the same form and are called in the same contexts as CPF_DEFAULT_CALLBACK. These are specified via the MATPOWER option 'cpf.user_callback'. This option can be a string containing the name of the callback function, or a struct with the following fields, where all but the first are optional:

- 'fcn' - string with name of callback function
- 'priority' - numerical value specifying callback priority (default = 20, see CPF_REGISTER_CALLBACK for details)
- 'args' - arbitrary value (any type) passed to the callback as CB_ARGS each time it is invoked

Multiple user callbacks can be registered by assigning a cell array of such strings and/or structs to the 'cpf.user_callback' option.

See also `runcpf()`, `cpf_register_callback()` (page 281).

cpf_detect_events

cpf_detect_events(*cpf_events, cef, pef, step, verbose*)

cpf_detect_events() (page 275) - Detect events from event function values.

```
[ROLLBACK, CRITICAL_EVENTS, CEF] = CPF_DETECT_EVENTS(CPF_EVENTS, CEF, PEF, STEP, ↳
↳VERBOSE)
```

Inputs:

- CPF_EVENTS : struct containing info about registered CPF event fcns
- CEF : cell array of Current Event Function values
- PEF : cell array of Previous Event Function values
- STEP : current step size
- VERBOSE : 0 = no output, otherwise level of verbose output

Outputs:

- ROLLBACK : flag indicating whether any event has requested a rollback step
- CRITICAL_EVENTS : struct array containing information about any detected events, with fields:
 - eidx : event index, in list of registered events
0 if no event detected
 - name : name of event function, empty if none detected
 - zero : 1 if zero has been detected, 0 otherwise
(interval detected or no event detected)
 - idx : index(es) of critical elements in event function
 - step_scale : linearly interpolated estimate of scaling factor

(continues on next page)

(continued from previous page)

```

    for current step size required to reach event zero
log      : 1 log the event in the results, 0 don't log the event
          (set to 1 for zero events, 0 otherwise, can be
           modified by callbacks)
msg      : event message, set to something generic like
          'ZERO detected for TARGET_LAM event' or
          'INTERVAL detected for QLIM(3) event', but intended
          to be changed/updated by callbacks
CEF : cell array of Current Event Function values

```

cpf_flim_event

cpf_flim_event(*cb_data, cx*)

[cpf_flim_event\(\)](#) (page 276) - Event function to detect branch flow limit (MVA) violations.

```
EF = CPF_FLIM_EVENT(CB_DATA, CX)
```

CPF event **function** to detect branch flow limit (MVA) violations,
i.e. `max(Sf,St) >= SrateA`.

Inputs:

CB_DATA : **struct** of data **for** callback **functions**
CX : **struct** containing **info** about current point (continuation soln)

Outputs:

EF : event **function** value

cpf_flim_event_cb

cpf_flim_event_cb(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

[cpf_flim_event_cb\(\)](#) (page 276) - Callback to handle FLIM events.

```

[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =
  CPF_NOSE_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...
    CB_DATA, CB_ARGS, RESULTS)

```

Callback to handle FLIM (branch flow limit violation) **events**,
triggered by event **function** CPF_FLIM_EVENT to indicate the point at which
a branch flow limit is reached.

All branch flows are expected to be within limits **for** the base **case**,
otherwise the continuation terminates.

This **function** sets the msg field of the event when the flow in **any** branch
reaches its limit, raises the DONE.**flag** and sets the DONE.msg.

For details of the input and output arguments see also [cpf_default_callback\(\)](#) (page 273).

cpf_nose_event

cpf_nose_event(*cb_data, cx*)

[cpf_nose_event\(\)](#) (page 277) - Event function to detect the nose point.

```
EF = CPF_NOSE_EVENT(CB_DATA, CX)
```

CPF event **function** to detect the nose point of the continuation curve, based on the **sign** of the lambda component of the tangent vector.

Inputs:

CB_DATA : **struct** of data **for** callback **functions**

CX : **struct** containing **info** about current point (continuation soln)

Outputs:

EF : event **function** value

cpf_nose_event_cb

cpf_nose_event_cb(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

[cpf_nose_event_cb\(\)](#) (page 277) - Callback to handle NOSE events.

```
[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =  
CPF_NOSE_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...  
CB_DATA, CB_ARGS, RESULTS)
```

Callback to handle NOSE **events**, triggered by event **function** CPF_NOSE_EVENT to indicate the nose point of the continuation curve.

This **function** sets the msg field of the event when the nose point has been found, raises the DONE.**flag** and sets the DONE.msg.

For details of the input and output arguments see also [cpf_default_callback\(\)](#) (page 273).

cpf_p

cpf_p(*parameterization, step, z, V, lam, Vprv, lamprv, pv, pq*)

[cpf_p\(\)](#) (page 277) m - Computes the value of the CPF parameterization function.

```
P = CPF_P(PARAMETERIZATION, STEP, Z, V, LAM, VPRV, LAMPRV, PV, PQ)
```

Computes the value of the parameterization **function** at the current solution point.

Inputs:

(continues on next page)

(continued from previous page)

PARAMETERIZATION : Value of `cpf.parameterization` option
 STEP : continuation step **size**
 Z : normalized tangent prediction vector from previous step
 V : **complex** bus voltage vector at current solution
 LAM : scalar lambda value at current solution
 VPRV : **complex** bus voltage vector at previous solution
 LAMPRV : scalar lambda value at previous solution
 PV : vector of indices of PV buses
 PQ : vector of indices of PQ buses

Outputs:

P : value of the parameterization **function** at the current point

See also [`cpf_predictor\(\)`](#) (page 279), [`cpf_corrector\(\)`](#) (page 272).

cpf_p_jac

cpf_p_jac(*parameterization, z, V, lam, Vprv, lamprv, pv, pq*)

[`cpf_p_jac\(\)`](#) (page 278) - Computes partial derivatives of CPF parameterization function.

```
[DP_DV, DP_DLAM] = CPF_P_JAC(PARAMETERIZATION, Z, V, LAM, ...
                              VPRV, LAMPRV, PV, PQ)
```

Computes the partial derivatives of the continuation **power** flow parameterization **function** w.r.t. bus voltages **and** the continuation parameter lambda.

Inputs:

PARAMETERIZATION : Value of `cpf.parameterization` option.
 Z : normalized tangent prediction vector from previous step
 V : **complex** bus voltage vector at current solution
 LAM : scalar lambda value at current solution
 VPRV : **complex** bus voltage vector at previous solution
 LAMPRV : scalar lambda value at previous solution
 PV : vector of indices of PV buses
 PQ : vector of indices of PQ buses

Outputs:

DP_DV : partial of parameterization **function** w.r.t. voltages
 DP_DLAM : partial of parameterization **function** w.r.t. lambda

See also [`cpf_predictor\(\)`](#) (page 279), [`cpf_corrector\(\)`](#) (page 272).

cpf_plim_event

cpf_plim_event(*cb_data, cx*)

[cpf_plim_event\(\)](#) (page 279) - Event function to detect gen active power limit violations.

```
EF = CPF_PLIM_EVENT(CB_DATA, CX)
```

CPF event **function** to detect generator active **power** limit violations, i.e. $P_g \geq P_{max}$.

Inputs:

CB_DATA : **struct** of data **for** callback **functions**
CX : **struct** containing **info** about current point (continuation soln)

Outputs:

EF : event **function** value

cpf_plim_event_cb

cpf_plim_event_cb(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

[cpf_plim_event_cb\(\)](#) (page 279) - Callback to handle PLIM events.

```
[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =  
CPF_PLIM_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...  
CB_DATA, CB_ARGS, RESULTS)
```

Callback to handle PLIM (generator active **power** limit violation) **events**, triggered by event **function** CPF_PLIM_EVENT to indicate the point at which an **upper** active **power** output limit is reached **for** a generator.

When an active **power** limit is encountered, this **function** **zeros** out subsequent transfers from that generator, chooses a new reference bus **if** necessary, **and** updates the CB_DATA accordingly, setting the next step **size** to zero. The event msg is updated with the details of the changes. It also requests termination **if** all generators reach PMAX.

For details of the input and output arguments see also [cpf_default_callback\(\)](#) (page 273).

cpf_predictor

cpf_predictor(*V, lam, z, step, pv, pq*)

[cpf_predictor\(\)](#) (page 279) - Performs the predictor step for the continuation power flow.

```
[V_HAT, LAM_HAT] = CPF_PREDICTOR(V, LAM, Z, STEP, PV, PQ)
```

Computes a prediction (approximation) to the next solution of the continuation **power** flow using a normalized tangent predictor.

(continues on next page)

(continued from previous page)

Inputs:

V : **complex** bus voltage vector at current solution
 LAM : scalar lambda value at current solution
 Z : normalized tangent prediction vector from previous step
 STEP : continuation step **length**
 PV : vector of indices of PV buses
 PQ : vector of indices of PQ buses

Outputs:

V_HAT : predicted **complex** bus voltage vector
 LAM_HAT : predicted lambda continuation parameter

cpf_qlim_event**cpf_qlim_event**(*cb_data, cx*)*cpf_qlim_event()* (page 280) - Event function to detect gen reactive power limit violations.

EF = CPF_QLIM_EVENT(CB_DATA, CX)

CPF event **function** to detect generator reactive **power** limit violations,
 i.e. $Q_g \leq Q_{min}$ or $Q_g \geq Q_{max}$.

Inputs:

CB_DATA : **struct** of data **for** callback **functions**
 CX : **struct** containing **info** about current point (continuation soln)

Outputs:

EF : event **function** value

cpf_qlim_event_cb**cpf_qlim_event_cb**(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)*cpf_qlim_event_cb()* (page 280) - Callback to handle QLIM events.

```
[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =
  CPF_QLIM_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...
    CB_DATA, CB_ARGS, RESULTS)
```

Callback to handle QLIM (generator reactive **power** limit violation) **events**,
 triggered by event **function** CPF_QLIM_EVENT to indicate the point at which
 an **upper** or **lower** reactive **power** output limit is reached **for** a generator.

When a reactive **power** limit is encountered, this **function** **zeros** out
 subsequent transfers from that generator, changes it's bus **type** to PQ,
 chooses a new reference bus **if** necessary, **and** updates the CB_DATA
 accordingly, setting the next step **size** to zero. The event msg is updated

(continues on next page)

(continued from previous page)

with the details of the changes. It also requests termination **if** no more PV or REF buses remain.

For details of the input and output arguments see also `cpf_default_callback()` (page 273).

cpf_register_callback

cpf_register_callback(*cpf_callbacks*, *fcn*, *priority*, *args*)

`cpf_register_callback()` (page 281) - Register CPF callback functions.

```
CPF_CALLBACKS = CPF_REGISTER_CALLBACK(CPF_CALLBACKS, FCN, PRIORITY)
```

Registers a CPF callback **function** to be called by RUNCPF.

Inputs:

CPF_CALLBACKS : **struct** containing **info** about registered CPF callback fcns

FCN : string containing name of callback **function**

PRIORITY : number that determines order of execution **for** multiple callback **functions**, where higher numbers **run** first, default priority is **20**, where the standard callbacks are called with the following priority:

cpf_flim_event_cb	53
cpf_vlim_event_cb	52
cpf_nose_event_cb	51
cpf_target_lam_event_cb	50
cpf_qlim_event_cb	41
cpf_plim_event_cb	40
cpf_default_callback	0

ARGS : arguments to be passed to the callback each **time** it is invoked

Outputs:

CPF_CALLBACKS : updated **struct** containing **info** about registered CPF callback fcns

User Defined CPF Callback Functions:

The user can define their own callback **functions** which take the same form **and** are called in the same contexts as CPF_DEFAULT_CALLBACK. These are specified via the MATPOWER option '`cpf.user_callback`'. This option can be a string containing the name of the callback **function**, or a **struct** with the following fields, where **all** but the first are optional:

- 'fcn' - string with name of callback **function**
- 'priority' - numerical value specifying callback priority (default = **20**, see CPF_REGISTER_CALLBACK **for** details)
- 'args' - arbitrary value (**any type**) passed to the callback as CB_ARGS each **time** it is invoked

Multiple user callbacks can be registered by assigning a **cell** array of such strings **and/or** structs to the '`cpf.user_callback`' option.

See also `runcpf()`, `cpf_default_callback()` (page 273).

cpf_register_event

cpf_register_event(*cpf_events, name, fcn, tol, locate*)

cpf_register_event() (page 282) - Register event functions.

```
CPF_EVENTS = CPF_REGISTER_EVENT(CPF_EVENTS, NAME, FCN, TOL, LOCATE)
```

Registers a CPF event **function** to be called by RUNCPF.

Inputs:

CPF_EVENTS : **struct** containing **info** about registered CPF event fcns

NAME : string containing event name

FCN : string containing name of event **function**, returning numerical scalar **or** vector value that changes **sign** at location of the event

TOL : scalar **or** vector of same dimension as event **function return** value of tolerance **for** detecting the event, i.e. **abs**(val) <= tol

LOCATE : **flag** indicating whether the event requests a rollback step to locate the event **function** zero

Outputs:

CPF_EVENTS : updated **struct** containing **info** about registered CPF event fcns

cpf_tangent

cpf_tangent(*V, lam, Ybus, Sbusb, Sbust, pv, pq, zprv, Vprv, lamprv, parameterization, direction*)

cpf_tangent() (page 282) - Computes normalized tangent predictor for continuation power flow.

```
Z = CPF_TANGENT(V, LAM, YBUS, SBUSB, SBUST, PV, PQ, ...
                ZPRV, VPRV, LAMPRV, PARAMETERIZATION, DIRECTION)
```

Computes a normalized tangent predictor **for** the continuation **power** flow.

Inputs:

V : **complex** bus voltage vector at current solution

LAM : scalar lambda value at current solution

YBUS : **complex** bus admittance matrix

SBUSB : handle of **function** returning nb x 1 vector of **complex** base **case** injections in p.u. **and** derivatives w.r.t. |V|

SBUST : handle of **function** returning nb x 1 vector of **complex** target **case** injections in p.u. **and** derivatives w.r.t. |V|

PV : vector of indices of PV buses

PQ : vector of indices of PQ buses

ZPRV : normalized tangent prediction vector from previous step

VPRV : **complex** bus voltage vector at previous solution

LAMPRV : scalar lambda value at previous solution

PARAMETERIZATION : value of cpf.parameterization option.

DIRECTION: continuation direction (+1 **for** postive lambda increase, -1 **otherwise**)

(continues on next page)

(continued from previous page)

Outputs:

Z : the normalized tangent prediction vector

cpf_target_lam_event**cpf_target_lam_event**(*cb_data, cx*)*cpf_target_lam_event*() (page 283) - Event function to detect a target lambda value.

EF = CPF_TARGET_LAM_EVENT(CB_DATA, CX)

CPF event **function** to detect the completion of the continuation curve
or another target value of lambda.**Inputs:**CB_DATA : **struct** of data **for** callback **functions**CX : **struct** containing **info** about current point (continuation soln)**Outputs:**EF : event **function** value**cpf_target_lam_event_cb****cpf_target_lam_event_cb**(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)*cpf_target_lam_event_cb*() (page 283) - Callback to handle TARGET_LAM events.[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =
CPF_TARGET_LAM_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...
CB_DATA, CB_ARGS, RESULTS)Callback to handle TARGET_LAM **events**, triggered by event **function**
CPF_TARGET_LAM_EVENT to indicate that a target lambda value has been
reached or that the **full** continuation curve has been traced.This **function** sets the msg field of the event when the target lambda has
been found, raises the DONE.**flag** and sets the DONE.msg. If the current
or predicted next step overshoot the target lambda, it adjusts the step
size to be exactly **what** is needed to reach the target, and sets the
parameterization **for** that step to be the natural parameterization.For details of the input and output arguments see also *cpf_default_callback*() (page 273).

cpf_vlim_event

cpf_vlim_event(*cb_data, cx*)

[cpf_vlim_event\(\)](#) (page 284) - Event function to detect bus voltage limit violations.

```
EF = CPF_VLIM_EVENT(CB_DATA, CX)
```

CPF event **function** to detect bus voltage limits violations,
i.e. $V_m \leq V_{min}$ or $V_m \geq V_{max}$.

Inputs:

CB_DATA : **struct** of data **for** callback **functions**
CX : **struct** containing **info** about current point (continuation soln)

Outputs:

EF : event **function** value

cpf_vlim_event_cb

cpf_vlim_event_cb(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

[cpf_vlim_event_cb\(\)](#) (page 284) - Callback to handle VLIM events.

```
[NX, CX, DONE, ROLLBACK, EVNTS, CB_DATA, RESULTS] =  
CPF_VLIM_EVENT_CB(K, NX, CX, PX, DONE, ROLLBACK, EVNTS, ...  
CB_DATA, CB_ARGS, RESULTS)
```

Callback to handle VLIM (bus voltage magnitude limit violation) **events**,
triggered by event **function** CPF_VLIM_EVENT to indicate the point at which
an **upper or lower** voltage magnitude limit is reached **for** a bus.

All bus voltages are expected to be within limits **for** the base **case**,
otherwise the continuation terminates.

This **function** sets the msg field of the event when the voltage magnitude
at **any** bus reaches its **upper or lower** limit, raises the DONE.flag and sets
the DONE.msg.

For details of the input and output arguments see also [cpf_default_callback\(\)](#) (page 273).

5.2.6 OPF and Wrapper Functions

opf**opf**(varargin)

opf() - Solves an optimal power flow.

[RESULTS, SUCCESS] = OPF(MPC, MPOPT)

Returns either a RESULTS **struct** and an optional SUCCESS **flag**, or individual data matrices, the objective **function** value and a SUCCESS **flag**. In the latter **case**, there are additional optional **return** values. See Examples below **for** the possible calling syntax options.

Examples:

Output argument options:

```
results = opf(...)
[results, success] = opf(...)
[bus, gen, branch, f, success] = opf(...)
[bus, gen, branch, f, success, info, et, g, jac, xr, pimul] = opf(...)
```

Input arguments options:

```
opf(mpc)
opf(mpc, mpopt)
opf(mpc, userfcn, mpopt)
opf(mpc, A, l, u)
opf(mpc, A, l, u, mpopt)
opf(mpc, A, l, u, mpopt, N, fparm, H, Cw)
opf(mpc, A, l, u, mpopt, N, fparm, H, Cw, z0, zl, zu)

opf(baseMVA, bus, gen, branch, areas, gencost)
opf(baseMVA, bus, gen, branch, areas, gencost, mpopt)
opf(baseMVA, bus, gen, branch, areas, gencost, userfcn, mpopt)
opf(baseMVA, bus, gen, branch, areas, gencost, A, l, u)
opf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, mpopt)
opf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
    mpopt, N, fparm, H, Cw)
opf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
    mpopt, N, fparm, H, Cw, z0, zl, zu)
```

The data **for** the problem can be specified in one of three ways:

- (1) a string (mpc) containing the file name of a MATPOWER **case** which defines the data matrices baseMVA, bus, gen, branch, and gencost (areas is **not** used at **all**, it is only included **for** backward compatibility of the API).
- (2) a **struct** (mpc) containing the data matrices as fields.
- (3) the individual data matrices themselves.

The optional user parameters **for** user constraints (A, l, u), user costs (N, fparm, H, Cw), user variable initializer (z0), and user variable limits (zl, zu) can also be specified as fields in a **case struct**, either passed in directly or defined in a **case** file referenced by name.

(continues on next page)

(continued from previous page)

When specified, A , l , u represent additional linear constraints on the optimization variables, $l \leq A[x; z] \leq u$. If the user specifies an A matrix that has more columns than the number of "x" (OPF) variables, then there are extra linearly constrained "z" variables. For an explanation of the formulation used and instructions for forming the A matrix, see the manual.

A generalized cost on all variables can be applied if input arguments N , $fparm$, H and Cw are specified. First, a linear transformation of the optimization variables is defined by means of $r = N * [x; z]$. Then, to each element of r a function is applied as encoded in the $fparm$ matrix (see manual). If the resulting vector is named w , then H and Cw define a quadratic cost on w : $(1/2)*w'*H*w + Cw * w$. H and N should be sparse matrices and H should also be symmetric.

The optional `mpopt` vector specifies MATPOWER options. If the OPF algorithm is not explicitly set in the options MATPOWER will use the default solver, based on a primal-dual interior point method. For the AC OPF this is `opf.ac.solver = 'MIPS'`, unless the TSPOPF optional package is installed, in which case the default is `'PDIPM'`. For the DC OPF, the default is `opf.dc.solver = 'MIPS'`. See `MPOPTION` for more details on the available OPF solvers and other OPF options and their default values.

The solved case is returned either in a single results struct (described below) or in the individual data matrices, `bus`, `gen` and `branch`. Also returned are the final objective function value (`f`) and a flag which is true if the algorithm was successful in finding a solution (success). Additional optional return values are an algorithm specific return status (`info`), elapsed time in seconds (`et`), the constraint vector (`g`), the Jacobian matrix (`jac`), and the vector of variables (`xr`) as well as the constraint multipliers (`pimul`).

The single results struct is a MATPOWER case struct (`mpc`) with the usual `baseMVA`, `bus`, `branch`, `gen`, `gencost` fields, along with the following additional fields:

```
.order      see 'help ext2int' for details of this field
.et         elapsed time in seconds for solving OPF
.success    1 if solver converged successfully, 0 otherwise
.om         OPF model object, see 'help opf_model'
.x          final value of optimization variables (internal order)
.f          final objective function value
.mu         shadow prices on ...
  .var
    .l      lower bounds on variables
    .u      upper bounds on variables
  .nlb
    .l      lower bounds on nonlinear constraints
    .u      upper bounds on nonlinear constraints
  .lin
    .l      lower bounds on linear constraints
```

(continues on next page)

(continued from previous page)

```

        .u  upper bounds on linear constraints
.raw      raw solver output in form returned by MINOS, and more
.xr       final value of optimization variables
.pimul    constraint multipliers
.info     solver specific termination code
.output   solver specific output information
        .alg algorithm code of solver used
.g        (optional) constraint values
.dg       (optional) constraint 1st derivatives
.df       (optional) obj fun 1st derivatives (not yet implemented)
.d2f      (optional) obj fun 2nd derivatives (not yet implemented)
.var
        .val optimization variable values, by named block
            .Va voltage angles
            .Vm voltage magnitudes (AC only)
            .Pg real power injections
            .Qg reactive power injections (AC only)
            .y constrained cost variable (only if have pwl costs)
            (other) any user defined variable blocks
        .mu variable bound shadow prices, by named block
            .l lower bound shadow prices
                .Va, Vm, Pg, Qg, y, (other)
            .u upper bound shadow prices
                .Va, Vm, Pg, Qg, y, (other)
        .nle (AC only)
            .lambda shadow prices on nonlinear equality constraints,
                by named block
                .Pmis real power mismatch equations
                .Qmis reactive power mismatch equations
                (other) use defined constraints
        .nli (AC only)
            .mu shadow prices on nonlinear inequality constraints,
                by named block
                .Sf flow limits at "from" end of branches
                .St flow limits at "to" end of branches
                (other) use defined constraints
        .lin
            .mu shadow prices on linear constraints, by named block
                .l lower bounds
                .Pmis real power mismatch equations (DC only)
                .Pf flow limits at "from" end of branches (DC only)
                .Pt flow limits at "to" end of branches (DC only)
                .PQh upper portion of gen PQ-capability curve (AC only)
                .PQl lower portion of gen PQ-capability curve (AC only)
                .vl constant power factor constraint for loads (AC only)
                .ycon basin constraints for CCV for pwl costs
                (other) any user defined constraint blocks
            .u upper bounds
                .Pmis, Pf, Pt, PQh, PQl, vl, ycon, (other)
        .cost user defined cost values, by named block

```

See also `runopf()`, `dcopf()`, `uopf()`, `caseformat`.

dcopf

dcopf(*varargin*)

dcopf() - Solves a DC optimal power flow.

This is a simple wrapper function around **opf**() that sets the `model` option to 'DC' before calling **opf**(). See **opf**() for the details of input and output arguments.

See also **rundcopf**().

fmincopf

fmincopf(*varargin*)

fmincopf() - Solves an AC optimal power flow using FMINCON (Opt Tbx 2.x & later).

Uses algorithm 520. Please see **opf**() for the details of input and output arguments.

uopf

uopf(*varargin*)

uopf() - Solves combined unit decommitment / optimal power flow.

```
[RESULTS, SUCCESS] = UOPF(MPC, MPOPT)
```

Returns either a **RESULTS** struct and an optional **SUCCESS** flag, or individual data matrices, the objective **function** value and a **SUCCESS** flag. In the latter case, there are additional optional **return** values. See Examples below for the possible calling syntax options.

Examples:

Output argument options:

```
results = uopf(...)
```

```
[results, success] = uopf(...)
```

```
[bus, gen, branch, f, success] = uopf(...)
```

```
[bus, gen, branch, f, success, info, et, g, jac, xr, pimul] = uopf(...)
```

Input arguments options:

```
uopf(mpc)
```

```
uopf(mpc, mpopt)
```

```
uopf(mpc, userfcn, mpopt)
```

```
uopf(mpc, A, l, u)
```

```
uopf(mpc, A, l, u, mpopt)
```

```
uopf(mpc, A, l, u, mpopt, N, fparm, H, Cw)
```

```
uopf(mpc, A, l, u, mpopt, N, fparm, H, Cw, z0, zl, zu)
```

(continues on next page)

(continued from previous page)

```

uopf(baseMVA, bus, gen, branch, areas, gencost)
uopf(baseMVA, bus, gen, branch, areas, gencost, mpopt)
uopf(baseMVA, bus, gen, branch, areas, gencost, userfcn, mpopt)
uopf(baseMVA, bus, gen, branch, areas, gencost, A, l, u)
uopf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, mpopt)
uopf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
      mpopt, N, fparm, H, Cw)
uopf(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
      mpopt, N, fparm, H, Cw, z0, z1, zu)

```

See OPF **for** more information on **input** and **output** arguments.

Solves a combined unit decommitment **and** optimal **power** flow **for** a **single** **time** period. Uses an algorithm similar to dynamic programming. It proceeds through a sequence of stages, where stage N has N generators shut down, starting with N=0. In each stage, it forms a list of candidates (gens at their Pmin limits) **and** computes the cost with each one of them shut down. It selects the least cost **case** as the starting point **for** the next stage, continuing **until** there are no more candidates to be shut down **or** no more improvement can be gained by shutting something down. If MPOPT.verbose (see MPOPTION) is **true**, it prints progress **info**, **if** it is > 1 it prints the output of each individual opf.

See also opf(), runuopf().

5.2.7 Other OPF Functions

dcopf_solver

dcopf_solver(om, mpopt)

dcopf_solver() (page 289) - Solves a DC optimal power flow.

```
[RESULTS, SUCCESS, RAW] = DCOPF_SOLVER(OM, MPOPT)
```

Inputs are an OPF model object **and** a MATPOWER options **struct**.

Outputs are a RESULTS **struct**, SUCCESS **flag** **and** RAW output **struct**.

RESULTS is a MATPOWER **case struct** (mpc) with the usual baseMVA, bus branch, gen, gencost fields, along with the following additional fields:

.order	see 'help ext2int' for details of this field
.x	final value of optimization variables (internal order)
.f	final objective function value
.mu	shadow prices on ...
.var	
.l	lower bounds on variables
.u	upper bounds on variables

(continues on next page)

(continued from previous page)

```

        .lin
            .l lower bounds on linear constraints
            .u upper bounds on linear constraints

SUCCESS    1 if solver converged successfully, 0 otherwise

RAW        raw output in form returned by MINOS
        .xr    final value of optimization variables
        .pimul constraint multipliers
        .info  solver specific termination code
        .output solver specific output information

```

See also `opf()`, `opt_model.solve()`.

nlpopf_solver

nlpopf_solver(*om*, *mpopt*)

nlpopf_solver() (page 290) - Solves AC optimal power flow using MP-Opt-Model.

```

[RESULTS, SUCCESS, RAW] = NLPF_SOLVER(OM, MPOPT)

Inputs are an OPF model object and a MATPOWER options struct.

Outputs are a RESULTS struct, SUCCESS flag and RAW output struct.

RESULTS is a MATPOWER case struct (mpc) with the usual baseMVA, bus
branch, gen, gencost fields, along with the following additional
fields:
    .order    see 'help ext2int' for details of this field
    .x        final value of optimization variables (internal order)
    .f        final objective function value
    .mu       shadow prices on ...
    .var
        .l lower bounds on variables
        .u upper bounds on variables
    .nln      (deprecated) 2*nb+2*nl - Pmis, Qmis, Sf, St
        .l lower bounds on nonlinear constraints
        .u upper bounds on nonlinear constraints
    .nle      nonlinear equality constraints
    .nli      nonlinear inequality constraints
    .lin
        .l lower bounds on linear constraints
        .u upper bounds on linear constraints

SUCCESS    1 if solver converged successfully, 0 otherwise

RAW        raw output in form returned by MINOS
        .xr    final value of optimization variables
        .pimul constraint multipliers

```

(continues on next page)

(continued from previous page)

```
.info solver specific termination code
.output solver specific output information
```

See also `opf()`, `mips()`.

makeAang

makeAang(*baseMVA*, *branch*, *nb*, *mpopt*)

makeAang() (page 291) - Construct constraints for branch angle difference limits.

```
[AANG, LANG, UANG, IANG] = MAKEAANG(BASEMVA, BRANCH, NB, MPOPT)
```

Constructs the parameters **for** the following linear constraint limiting the voltage **angle** differences across branches, where *Va* is the vector of bus voltage angles. NB is the number of buses.

$$LANG \leq AANG * Va \leq UANG$$

IAANG is the vector of indices of branches with **angle** difference limits. The limits are given in the ANGMIN and ANGMAX columns of the branch matrix. Voltage **angle** differences are taken to be unbounded below **if** ANGMIN < -360 and unbounded above **if** ANGMAX > 360. If both ANGMIN and ANGMAX are zero, the **angle** difference is assumed to be unconstrained.

Example:

```
[Aang, lang, uang, iang] = makeAang(baseMVA, branch, nb, mpopt);
```

makeApq

makeApq(*baseMVA*, *gen*)

makeApq() (page 291) - Construct linear constraints for generator capability curves.

```
[APQH, UBPQH, APQL, UBPQL, DATA] = MAKEAPQ(BASEMVA, GEN)
```

Constructs the parameters **for** the following linear constraints implementing trapezoidal generator capability curves, where *Pg* and *Qg* are the **real** and **reactive** generator injections.

$$APQH * [Pg; Qg] \leq UBPQH$$

$$APQL * [Pg; Qg] \leq UBPQL$$

DATA contains additional information as shown below.

Example:

```
[Apqh, ubpqh, Apql, ubpql, data] = makeApq(baseMVA, gen);
```

```
data.h      [QC1MAX-QC2MAX, PC2-PC1]
```

(continues on next page)

(continued from previous page)

data.l	[QC2MIN-QC1MIN, PC1-PC2]
data.ipqh	indices of gens with general PQ cap curves (upper)
data.ipql	indices of gens with general PQ cap curves (lower)

makeAvl

makeAvl(baseMVA, gen)*makeAvl*() (page 292) - Construct linear constraints for constant power factor var loads.

```
[AVL, LVL, UVL, IVL] = MAKEAVL(MPC)
[AVL, LVL, UVL, IVL] = MAKEAVL(BASEMVA, GEN) (deprecated)
```

Constructs parameters **for** the following linear constraint enforcing a constant **power factor** constraint **for** dispatchable loads.

$$LVL \leq AVL * [Pg; Qg] \leq UVL$$

IVL is the vector of indices of generators representing variable loads.

Example:

```
[Avl, lvl, uvl, ivl] = makeAvl(mpc);
[Avl, lvl, uvl, ivl] = makeAvl(baseMVA, gen); %% deprecated
```

makeAy

makeAy(baseMVA, ng, gencost, pgbas, qgbas, ybas)*makeAy*() (page 292) - Make the A matrix and RHS for the CCV formulation.

```
[AY, BY] = MAKEAY(BASEMVA, NG, GENCOST, PGBAS, QGBAS, YBAS)
```

Constructs the parameters **for** linear "**basin constraints**" on Pg, Qg **and** Y used by the CCV cost formulation, expressed as

$$AY * X \leq BY$$

where X is the vector of optimization variables. The starting **index** within the X vector **for** the active, reactive sources **and** the Y variables should be provided in arguments PGBAS, QGBAS, YBAS. The number of generators is NG.

Assumptions: All generators are in-service. Filter **any** generators that are offline from the GENCOST matrix before calling MAKEAY. Efficiency depends on Qg variables being after Pg variables, **and** the Y variables must be the last variables within the vector X **for** the dimensions of the resulting AY to be conformable with X.

(continues on next page)

(continued from previous page)

Example:

```
[Ay, by] = makeAy(baseMVA, ng, gencost, pgbas, qgbas, ybas);
```

margcost

margcost(gencost, Pg)

margcost() - Computes marginal cost for generators at given output level.

MARGINALCOST = MARGCOST(GENCOST, PG) computes marginal cost **for** generators given a matrix in gencost format **and** a column vector of generation levels. The **return** value has the same dimensions as PG. Each row of GENCOST is used to evaluate the cost at the points specified in the corresponding row of PG.

opf_args

opf_args(baseMVA, bus, gen, branch, areas, gencost, Au, lbu, ubu, mpopt, N, fparm, H, Cw, z0, zl, zu)

opf_args() (page 293) - Parses and initializes OPF input arguments.

```
[MPC, MPOPT] = OPF_ARGS( ... )
[BASEMVA, BUS, GEN, BRANCH, GENCOST, A, L, U, MPOPT, ...
 N, FPARM, H, CW, Z0, ZL, ZU, USERFCN] = OPF_ARGS( ... )
```

Returns the **full set** of initialized OPF **input** arguments, filling in default values **for** missing arguments. See Examples below **for** the possible calling syntax options.

Examples:

Output argument options:

```
[mpc, mpopt] = opf_args( ... )
[baseMVA, bus, gen, branch, gencost, A, l, u, mpopt, ...
 N, fparm, H, Cw, z0, zl, zu, userfcn] = opf_args( ... )
```

Input arguments options:

```
opf_args(mpc)
opf_args(mpc, mpopt)
opf_args(mpc, userfcn, mpopt)
opf_args(mpc, A, l, u)
opf_args(mpc, A, l, u, mpopt)
opf_args(mpc, A, l, u, mpopt, N, fparm, H, Cw)
opf_args(mpc, A, l, u, mpopt, N, fparm, H, Cw, z0, zl, zu)

opf_args(baseMVA, bus, gen, branch, areas, gencost)
opf_args(baseMVA, bus, gen, branch, areas, gencost, mpopt)
opf_args(baseMVA, bus, gen, branch, areas, gencost, userfcn, mpopt)
opf_args(baseMVA, bus, gen, branch, areas, gencost, A, l, u)
```

(continues on next page)

(continued from previous page)

```

opf_args(baseMVA, bus, gen, branch, areas, gencost, A, l, u, mpopt)
opf_args(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
        mpopt, N, fparm, H, Cw)
opf_args(baseMVA, bus, gen, branch, areas, gencost, A, l, u, ...
        mpopt, N, fparm, H, Cw, z0, zl, zu)

```

The data **for** the problem can be specified in one of three ways:

- (1) a string (*mpc*) containing the file name of a MATPOWER **case** which defines the data matrices *baseMVA*, *bus*, *gen*, *branch*, and *gencost* (*areas* is **not** used at **all**, it is only included **for** backward compatibility of the API).
- (2) a **struct** (*mpc*) containing the data matrices as fields.
- (3) the individual data matrices themselves.

The optional user parameters **for** user constraints (*A*, *l*, *u*), user costs (*N*, *fparm*, *H*, *Cw*), user variable initializer (*z0*), and user variable limits (*zl*, *zu*) can also be specified as fields in a **case struct**, either passed in directly or defined in a **case** file referenced by name.

When specified, *A*, *l*, *u* represent additional linear constraints on the optimization variables, $l \leq A[x; z] \leq u$. If the user specifies an *A* matrix that has more **columns** than the number of "*x*" (OPF) variables, then there are extra linearly constrained "*z*" variables. For an explanation of the formulation used and instructions **for** forming the *A* matrix, see the manual.

A generalized cost on **all** variables can be applied **if** input arguments *N*, *fparm*, *H* and *Cw* are specified. First, a linear transformation of the optimization variables is defined by means of $r = N * [x; z]$. Then, to each element of *r* a **function** is applied as encoded in the *fparm* matrix (see manual). If the resulting vector is named *w*, then *H* and *Cw* define a quadratic cost on *w*: $(1/2)*w'*H*w + Cw * w$. *H* and *N* should be **sparse** matrices and *H* should also be symmetric.

The optional *mpopt* vector specifies MATPOWER options. See *MPOPTION* **for** details and default values.

opf_setup

opf_setup(*mpc*, *mpopt*)

opf_setup() (page 294) - Constructs an OPF model object from a MATPOWER case struct.

```
OM = OPF_SETUP(MPC, MPOPT)
```

Assumes that *MPC* is a MATPOWER **case struct** with internal indexing, **all** equipment in-service, etc.

See also *opf*(), *ext2int*(), *opf_execute*() (page 295).

opf_execute

opf_execute(*om*, *mpopt*)

[opf_execute\(\)](#) (page 295) - Executes the OPF specified by an OPF model object.

```
[RESULTS, SUCCESS, RAW] = OPF_EXECUTE(OM, MPOPT)
```

RESULTS are returned with internal indexing, **all** equipment in-service, etc.

See also [opf\(\)](#), [opf_setup\(\)](#) (page 294).

opf_branch_ang_fcn

opf_branch_ang_fcn(*x*, *Aang*, *lang*, *uang*)

[opf_branch_ang_fcn\(\)](#) (page 295) - Evaluates branch angle difference constraints and gradients.

```
[VADIF, DVADIF] = OPF_BRANCH_ANG_FCN(X, AANG, LANG, UANG);
```

Computes the **lower** and **upper** constraints on branch **angle** differences **for** voltages in cartesian coordinates. Computes constraint vectors **and** their gradients. The constraints are of the form:

$Aang * Va \geq lang$

$Aang * Va \leq uang$

where Va is the voltage **angle**, a non-linear **function** of the Vr **and** Vi .

Inputs:

X : optimization vector

AANG : constraint matrix, see MAKEAANG

LANG : **lower** bound vector, see MAKEAANG

UANG : **upper** bound vector, see MAKEAANG

Outputs:

VADIF : constraint vector [$lang - Aang * Va$; $Aang * Va - uang$]

DVADIF : (optional) constraint gradients

Examples:

VaDif = opf_branch_ang_fcn(x, Aang, lang, uang);

[VaDif, dVaDif] = opf_branch_ang_fcn(x, Aang, lang, uang);

See also [opf_branch_ang_hess\(\)](#) (page 296).

opf_branch_ang_hess**opf_branch_ang_hess**(*x, lambda, Aang, lang, uang*)*opf_branch_ang_hess*() (page 296) - Evaluates Hessian of branch angle difference constraints.

D2VADIF = OPF_BRANCH_ANG_HESS(X, LAMBDA, AANG, LANG, UANG)

Hessian evaluation **function for** branch **angle** difference constraints
for voltages in cartesian coordinates.

Inputs:

X : optimization vector
 LAMBDA : column vector of Lagrange multipliers on branch **angle**
 difference constraints, **lower**, then **upper**
 AANG : constraint matrix, see MAKEAANG
 LANG : **lower** bound vector, see MAKEAANG
 UANG : **upper** bound vector, see MAKEAANG

Outputs:

D2VADIF : Hessian of branch **angle** difference constraints.

Example:

d2VaDif = opf_branch_ang_hess(x, lambda, Aang, lang, uang);

See also *opf_branch_ang_fcn*() (page 295).**opf_branch_flow_fcn****opf_branch_flow_fcn**(*x, mpc, Yf, Yt, il, mpopt*)*opf_branch_flow_fcn*() (page 296) - Evaluates AC branch flow constraints and Jacobian.

[H, DH] = OPF_BRANCH_FLOW_FCN(X, OM, YF, YT, IL, MPOPT)

Branch flow constraints **for** AC optimal **power** flow.
Computes constraint vectors **and** their gradients.

Inputs:

X : optimization vector
 MPC : MATPOWER **case struct**
 YF : admittance matrix **for "from" end** of constrained branches
 YT : admittance matrix **for "to" end** of constrained branches
 IL : vector of branch indices corresponding to branches with
 flow limits (**all** others are assumed to be unconstrained).
 YF **and** YT contain only the **rows** corresponding to IL.
 MPOPT : MATPOWER options **struct**

Outputs:

H : vector of inequality constraint values (flow limits)
 where the flow can be apparent **power**, **real power**, or
 current, depending on the value of opf.flow_lim in MPOPT
 (only **for** constrained lines), normally expressed as

(continues on next page)

(continued from previous page)

($\text{limit}^2 - \text{flow}^2$), except when `opf.flow_lim == 'P'`, in which **case** it is simply ($\text{limit} - \text{flow}$).
 DH : (optional) inequality constraint gradients, column j is **gradient** of $H(j)$

Examples:

```
h = opf_branch_flow_fcn(x, mpc, Yf, Yt, il, mpopt);
[h, dh] = opf_branch_flow_fcn(x, mpc, Yf, Yt, il, mpopt);
```

See also [opf_branch_flow_hess\(\)](#) (page 297).

opf_branch_flow_hess

opf_branch_flow_hess(*x, lambda, mpc, Yf, Yt, il, mpopt*)

[opf_branch_flow_hess\(\)](#) (page 297) - Evaluates Hessian of branch flow constraints.

```
D2H = OPF_BRANCH_FLOW_HESS(X, LAMBDA, OM, YF, YT, IL, MPOPT)
```

Hessian evaluation **function for** AC branch flow constraints.

Inputs:

X : optimization vector
 LAMBDA : column vector of Kuhn-Tucker multipliers on constrained branch flows
 MPC : MATPOWER **case struct**
 YF : admittance matrix **for "from" end** of constrained branches
 YT : admittance matrix **for "to" end** of constrained branches
 IL : vector of branch indices corresponding to branches with flow limits (**all** others are assumed to be unconstrained).
 YF and YT contain only the **rows** corresponding to IL.
 MPOPT : MATPOWER options **struct**

Outputs:

D2H : Hessian of AC branch flow constraints.

Example:

```
d2H = opf_branch_flow_hess(x, lambda, mpc, Yf, Yt, il, mpopt);
```

See also [opf_branch_flow_fcn\(\)](#) (page 296).

opf_current_balance_fcn

opf_current_balance_fcn(*x, mpc, Ybus, mpopt*)

[opf_current_balance_fcn\(\)](#) (page 298) - Evaluates AC current balance constraints and their gradients.

```
[G, DG] = OPF_CURRENT_BALANCE_FCN(X, OM, YBUS, MPOPT)
```

Computes the **real** or imaginary current **balance** equality constraints **for** AC optimal **power** flow. Computes constraint vectors **and** their gradients.

Inputs:

X : optimization vector
MPC : MATPOWER **case struct**
YBUS : bus admittance matrix
MPOPT : MATPOWER options **struct**

Outputs:

G : vector of equality constraint values (**real**/imaginary current balances)
DG : (optional) equality constraint gradients

Examples:

```
g = opf_current_balance_fcn(x, mpc, Ybus, mpopt);  
[g, dg] = opf_current_balance_fcn(x, mpc, Ybus, mpopt);
```

See also [opf_power_balance_hess\(\)](#) (page 300).

opf_current_balance_hess

opf_current_balance_hess(*x, lambda, mpc, Ybus, mpopt*)

[opf_current_balance_hess\(\)](#) (page 298) - Evaluates Hessian of current balance constraints.

```
D2G = OPF_CURRENT_BALANCE_HESS(X, LAMBDA, OM, YBUS, MPOPT)
```

Hessian evaluation **function for** AC **real and** imaginary current **balance** constraints.

Inputs:

X : optimization vector
LAMBDA : column vector of Lagrange multipliers on **real and** imaginary current **balance** constraints
MPC : MATPOWER **case struct**
YBUS : bus admittance matrix
MPOPT : MATPOWER options **struct**

Outputs:

D2G : Hessian of current **balance** constraints.

Example:

```
d2G = opf_current_balance_hess(x, lambda, mpc, Ybus, mpopt);
```

See also [opf_current_balance_fcn\(\)](#) (page 298).

opf_gen_cost_fcn

opf_gen_cost_fcn(*x*, *baseMVA*, *gencost*, *ig*)

[opf_gen_cost_fcn\(\)](#) (page 299) - Evaluates polynomial generator costs and derivatives.

```
[F, DF, D2F] = OPF_GEN_COST_FCN(X, BASEMVA, COST)
```

Evaluates the polynomial generator costs and derivatives.

Inputs:

X : single-element cell array with vector of (active or reactive) dispatches (in per unit)
BASEMVA : system per unit base
GENCOST : standard gencost matrix corresponding to dispatch (active or reactive) provided in X
IG : vector of generator indices of interest
MPOPT : MATPOWER options struct

Outputs:

F : sum of generator polynomial costs
DF : (optional) gradient (column vector) of polynomial costs
D2F : (optional) Hessian of polynomial costs

Examples:

```
f = opf_gen_cost_fcn(x, baseMVA, gencost, ig);  
[f, df] = opf_gen_cost_fcn(x, baseMVA, gencost, ig);  
[f, df, d2f] = opf_gen_cost_fcn(x, baseMVA, gencost, ig);
```

opf_legacy_user_cost_fcn

opf_legacy_user_cost_fcn(*x*, *cp*)

[opf_legacy_user_cost_fcn\(\)](#) (page 299) - Evaluates legacy user costs and derivatives.

```
[F, DF, D2F] = OPF_LEGACY_USER_COST_FCN(X, CP)
```

Evaluates the legacy user-defined costs and derivatives.

Inputs:

X : cell array with vectors of optimization variables
CP : legacy user-defined cost parameter struct such as returned by OPT_MODEL.GET_COST_PARAMS

Outputs:

F : sum of generator polynomial costs
DF : (optional) gradient (column vector) of polynomial costs
D2F : (optional) Hessian of polynomial costs

Examples:

(continues on next page)

(continued from previous page)

```
f = opf_legacy_user_cost_fcn(x, cp);
[f, df] = opf_legacy_user_cost_fcn(x, cp);
[f, df, d2f] = opf_legacy_user_cost_fcn(x, cp);
```

opf_power_balance_fcn

opf_power_balance_fcn(*x, mpc, Ybus, mpopt*)

[opf_power_balance_fcn\(\)](#) (page 300) - Evaluates AC power balance constraints and their gradients.

```
[G, DG] = OPF_POWER_BALANCE_FCN(X, OM, YBUS, MPOPT)
```

Computes the active **or** reactive **power balance** equality constraints **for** AC optimal **power** flow. Computes constraint vectors **and** their gradients.

Inputs:

X : optimization vector
 MPC : MATPOWER **case struct**
 YBUS : bus admittance matrix
 MPOPT : MATPOWER options **struct**

Outputs:

G : vector of equality constraint values (active/reactive **power** balances)
 DG : (optional) equality constraint gradients

Examples:

```
g = opf_power_balance_fcn(x, mpc, Ybus, mpopt);
[g, dg] = opf_power_balance_fcn(x, mpc, Ybus, mpopt);
```

See also [opf_power_balance_hess\(\)](#) (page 300).

opf_power_balance_hess

opf_power_balance_hess(*x, lambda, mpc, Ybus, mpopt*)

[opf_power_balance_hess\(\)](#) (page 300) - Evaluates Hessian of power balance constraints.

```
D2G = OPF_POWER_BALANCE_HESS(X, LAMBDA, OM, YBUS, MPOPT)
```

Hessian evaluation **function for** AC active **and** reactive **power balance** constraints.

Inputs:

X : optimization vector
 LAMBDA : column vector of Lagrange multipliers on active **and** reactive **power balance** constraints
 MPC : MATPOWER **case struct**
 YBUS : bus admittance matrix
 MPOPT : MATPOWER options **struct**

(continues on next page)

(continued from previous page)

Outputs:

D2G : Hessian of power balance constraints.

Example:

```
d2G = opf_power_balance_hess(x, lambda, mpc, Ybus, mpopt);
```

See also [opf_power_balance_fcn\(\)](#) (page 300).

opf_veq_fcn

opf_veq_fcn(*x, mpc, idx, mpopt*)

[opf_veq_fcn\(\)](#) (page 301) - Evaluates voltage magnitude equality constraint and gradients.

```
[Veq, dVeq] = OPF_VEQ_FCN(X, MPC, IDX, MPOPT)
```

Computes the voltage magnitudes using real and imaginary part of complex voltage for AC optimal power flow. Computes constraint vectors and their gradients.

Inputs:

X : optimization vector

MPC : MATPOWER case struct

IDX : index of buses whose voltage magnitudes should be fixed

MPOPT : MATPOWER options struct

Outputs:

VEQ : vector of voltage magnitudes

DVEQ : (optional) magnitude gradients

Examples:

```
Veq = opf_veq_fcn(x, mpc, mpopt);
```

```
[Veq, dVeq] = opf_veq_fcn(x, mpc, idx, mpopt);
```

See also [opf_veq_hess\(\)](#) (page 301).

opf_veq_hess

opf_veq_hess(*x, lambda, mpc, idx, mpopt*)

[opf_veq_hess\(\)](#) (page 301) - Evaluates Hessian of voltage magnitude equality constraint.

```
D2VEQ = OPF_VEQ_HESS(X, LAMBDA, MPC, IDX, MPOPT)
```

Hessian evaluation function for voltage magnitudes.

Inputs:

X : optimization vector

LAMBDA : column vector of Lagrange multipliers on active and reactive

(continues on next page)

(continued from previous page)

```

    power balance constraints
MPC : MATPOWER case struct
IDX : index of buses whose voltage magnitudes should be fixed
MPOPT : MATPOWER options struct

Outputs:
    D2VEQ : Hessian of voltage magnitudes.

Example:
    d2Veq = opf_veq_hess(x, lambda, mpc, idx, mpopt);

```

See also [opf_veq_fcn\(\)](#) (page 301).

opf_vlim_fcn

opf_vlim_fcn(*x*, *mpc*, *idx*, *mpopt*)

[opf_vlim_fcn\(\)](#) (page 302) - Evaluates voltage magnitudes and their gradients.

```

[Vlims, dVlims] = OPF_VLIM_FCN(X, MPC, IDX, MPOPT)

Computes the voltage magnitudes using real and imaginary part of complex voltage for
AC optimal power flow. Computes constraint vectors and their gradients.

Inputs:
    X : optimization vector
    MPC : MATPOWER case struct
    IDX : index of buses whose voltage magnitudes should be fixed
    MPOPT : MATPOWER options struct

Outputs:
    VLIMS : vector of voltage magnitudes
    DVLIMS : (optional) magnitude gradients

Examples:
    Vlims = opf_vlim_fcn(x, mpc, mpopt);
    [Vlims, dVlims] = opf_vlim_fcn(x, mpc, idx, mpopt);

```

See also [opf_vlim_hess\(\)](#) (page 303).

opf_vlim_hess**opf_vlim_hess**(*x, lambda, mpc, idx, mpopt*)*opf_vlim_hess*() (page 303) - Evaluates Hessian of voltage magnitudes.

D2VLIMS = OPF_VLIM_HESS(X, LAMBDA, MPC, IDX, MPOPT)

Hessian evaluation **function for** voltage magnitudes.

Inputs:

X : optimization vector
 LAMBDA : column vector of Lagrange multipliers on active **and** reactive
 power balance constraints
 MPC : MATPOWER **case struct**
 IDX : **index** of buses whose voltage magnitudes should be fixed
 MPOPT : MATPOWER options **struct**

Outputs:

D2VLIMS : Hessian of voltage magnitudes.

Example:

d2Vlims = opf_vlim_hess(x, lambda, mpc, idx, mpopt);

See also *opf_vlim_fcn*() (page 302).**opf_vref_fcn****opf_vref_fcn**(*x, mpc, refs, mpopt*)*opf_vref_fcn*() (page 303) - Evaluates voltage angle reference and their gradients.

[Vref, dVref] = OPF_VREF_FCN(X, mpc, ref, MPOPT)

Computes the voltage **angle** reference using **real and** imaginary part of **complex**
 ↪ **voltage for**
 AC optimal **power** flow. Computes constraint vectors **and** their gradients.

Inputs:

X : optimization vector
 MPC : MATPOWER **case struct**
 REFS : reference vector
 MPOPT : MATPOWER options **struct**

Outputs:

VREF : vector of voltage **angle** reference
 DVREF : (optional) **angle** reference gradients

Examples:

Vref = opf_vref_fcn(x, mpc, refs, mpopt);
 [Vref, dVref] = opf_vref_fcn(x, mpc, refs, mpopt);

See also *opf_vref_hess*() (page 304).

opf_vref_hess

opf_vref_hess(*x, lam, mpc, refs, mpopt*)

[opf_vref_hess\(\)](#) (page 304) - Evaluates Hessian of voltage angle reference.

```
D2VREF = OPF_VREF_HESS(X, LAMBDA, MPC, REFS, MPOPT)
```

Hessian evaluation **function for** voltage **angle** reference.

Inputs:

- X : optimization vector
- LAMBDA : column vector of Lagrange multipliers on active **and** reactive **power balance** constraints
- MPC : MATPOWER **case struct**
- REFS : reference vector
- MPOPT : MATPOWER options **struct**

Outputs:

- D2VREF : Hessian of voltage **angle** reference.

Example:

```
d2Vref = opf_vref_hess(x, lambda, mpc, refs, mpopt);
```

See also [opf_vref_fcn\(\)](#) (page 303).

totcost

totcost(*gencost, Pg*)

[totcost\(\)](#) - Computes total cost for generators at given output level.

```
TOTALCOST = TOTCOST(GENCOST, PG)
```

computes total cost **for** generators given a matrix in gencost format **and** a column vector **or** matrix of generation levels. The **return** value has the same dimensions as PG. Each row of GENCOST is used to evaluate the cost at the points specified in the corresponding row of PG.

update_mupq

update_mupq(*baseMVA, gen, mu_PQh, mu_PQL, data*)

[update_mupq\(\)](#) (page 304) - Updates values of generator limit shadow prices.

```
GEN = UPDATE_MUPQ(BASEMVA, GEN, MU_PQH, MU_PQL, DATA)
```

Updates the values of MU_PMIN, MU_PMAX, MU_QMIN, MU_QMAX based on **any** shadow prices on the sloped portions of the generator

(continues on next page)

(continued from previous page)

capability curve constraints.

MU_PQH - shadow prices on **upper** sloped portion of capability curves
 MU_PQL - shadow prices on **lower** sloped portion of capability curves
 DATA - **"data" struct** returned by MAKEAPQ

See also *makeApq()* (page 291).

5.2.8 OPF User Callback Functions

add_userfcn

add_userfcn(mpc, stage, fcn, args, allow_multiple)

add_userfcn() (page 305) - Appends a userfcn to the list to be called for a case.

```
MPC = ADD_USERFCN(MPC, STAGE, FCN)
MPC = ADD_USERFCN(MPC, STAGE, FCN, ARGS)
MPC = ADD_USERFCN(MPC, STAGE, FCN, ARGS, ALLOW_MULTIPLE)
```

A userfcn is a callback **function** that can be called automatically by MATPOWER at one of various stages in a simulation.

MPC : the **case struct**
 STAGE : the name of the stage at which this **function** should be called: ext2int, formulation, int2ext, printpf, savecase
 FCN : the name of the userfcn
 ARGS : (optional) the value to be passed as an argument to the userfcn (typically a **struct**)
 ALLOW_MULTIPLE : (optional) **if** TRUE, allows the same **function** to be added more than once.

Currently there are 5 different callback stages defined. Each stage has a name, **and** by convention, the name of a user-defined callback **function** ends with the name of the stage. The following is a description of each stage, when it is called **and** the **input and** output arguments which vary depending on the stage. The reserves **example** (see RUNOPF_W_RES) is used to illustrate how these callback userfcns might be used.

1. ext2int

Called from EXT2INT immediately after the **case** is converted from external to internal indexing. Inputs are a MATPOWER **case struct** (MPC), freshly converted to internal indexing **and** any (optional) ARGS value supplied via ADD_USERFCN. Output is the (presumably updated) MPC. This is typically used to reorder **any input** arguments that may be needed in internal ordering by the formulation stage.

E.g. mpc = userfcn_reserves_ext2int(mpc, mpopt, args)

(continues on next page)

(continued from previous page)

2. formulation

Called from OPF after the OPF Model (OM) object has been initialized with the standard OPF formulation, but before calling the solver. Inputs are the OM object and any (optional) ARGS supplied via ADD_USERFCN. Output is the OM object. This is the ideal place to add any additional vars, constraints or costs to the OPF formulation.

E.g. `om = userfcn_reserves_formulation(om, mpopt, args)`

3. int2ext

Called from INT2EXT immediately before the resulting case is converted from internal back to external indexing. Inputs are the RESULTS struct and any (optional) ARGS supplied via ADD_USERFCN. Output is the RESULTS struct. This is typically used to convert any results to external indexing and populate any corresponding fields in the RESULTS struct.

E.g. `results = userfcn_reserves_int2ext(results, mpopt, args)`

4. printpf

Called from PRINTPF after the pretty-printing of the standard OPF output. Inputs are the RESULTS struct, the file descriptor to write to, a MATPOWER options struct, and any (optional) ARGS supplied via ADD_USERFCN. Output is the RESULTS struct. This is typically used for any additional pretty-printing of results.

E.g. `results = userfcn_reserves_printpf(results, fd, mpopt, args)`

5. savecase

Called from SAVECASE when saving a case struct to an M-file after printing all of the other data to the file. Inputs are the case struct, the file descriptor to write to, the variable prefix (typically 'mpc.') and any (optional) ARGS supplied via ADD_USERFCN. Output is the case struct. This is typically used to write any non-standard case struct fields to the case file.

E.g. `mpc = userfcn_reserves_printpf(mpc, fd, prefix, args)`

See also `run_userfcn()` (page 307), `remove_userfcn()` (page 307), `toggle_reserves()` (page 309), `toggle_iflims()` (page 308), `toggle_dcline()` (page 307), `toggle_softlims()` (page 310), `runopf_w_res()` (page 238).

remove_userfcn

remove_userfcn(*mpc, stage, fcn*)

[remove_userfcn\(\)](#) (page 307) - Removes a userfcn from the list to be called for a case.

```
MPC = REMOVE_USERFCN(MPC, STAGE, FCN)
```

A userfcn is a callback **function** that can be called automatically by MATPOWER at one of various stages in a simulation. This **function** removes the last instance of the userfcn **for** the given STAGE with the **function** handle specified by FCN.

See also [add_userfcn\(\)](#) (page 305), [run_userfcn\(\)](#) (page 307), [toggle_reserves\(\)](#) (page 309), [toggle_iflims\(\)](#) (page 308), [runopf_w_res\(\)](#) (page 238).

run_userfcn

run_userfcn(*userfcn, stage, varargin*)

[run_userfcn\(\)](#) (page 307) - Runs the userfcn callbacks for a given stage.

```
RV = RUN_USERFCN(USERFCN, STAGE, VARARGIN)
```

USERFCN : the 'userfcn' field of mpc, populated by ADD_USERFCN
STAGE : the name of the callback stage being executed
(additional arguments) some stages require additional arguments.

Example:

```
mpc = om.get_mpc();  
om = run_userfcn(mpc.userfcn, 'formulation', om);
```

See also [add_userfcn\(\)](#) (page 305), [remove_userfcn\(\)](#) (page 307), [toggle_reserves\(\)](#) (page 309), [toggle_iflims\(\)](#) (page 308), [runopf_w_res\(\)](#) (page 238).

toggle_dcline

toggle_dcline(*mpc, on_off*)

[toggle_dcline\(\)](#) (page 307) - Enable, disable or check status of DC line modeling.

```
MPC = TOGGLE_DCLINE(MPC, 'on')  
MPC = TOGGLE_DCLINE(MPC, 'off')  
T_F = TOGGLE_DCLINE(MPC, 'status')
```

Enables, disables **or** checks the status of a **set** of OPF userfcn callbacks to implement DC lines as a pair of linked generators. While it uses the OPF extension mechanism, this implementation works **for** simple **power** flow as well as OPF problems.

These callbacks expect to **find** a 'dcline' field in the **input** MPC, where MPC.dcline is an ndc x 17 matrix with **columns** as defined

(continues on next page)

(continued from previous page)

in `IDX_DCLINE`, where `ndc` is the number of DC lines.

The `'int2ext'` callback also packages up flow results and stores them in appropriate `columns` of `MPC.dcline`.

NOTE: Because of the way this extension modifies the number of `rows` in the `gen` and `gencost` matrices, caution must be taken when using it with other extensions that deal with generators.

Examples:

```
mpc = loadcase('t_case9_dcline');
mpc = toggle_dcline(mpc, 'on');
results1 = runpf(mpc);
results2 = runopf(mpc);
```

See also `idx_dcline()` (page 346), `add_userfcn()` (page 305), `remove_userfcn()` (page 307), `run_userfcn()` (page 307).

toggle_iflims

toggle_iflims(*mpc, on_off*)

`toggle_iflims()` (page 308) - Enable, disable or check status of set of interface flow limits.

```
MPC = TOGGLE_IFLIMS(MPC, 'on')
MPC = TOGGLE_IFLIMS(MPC, 'off')
T_F = TOGGLE_IFLIMS(MPC, 'status')
```

Enables, disables or checks the status of a `set` of OPF `userfcn` callbacks to implement interface flow limits based on a DC flow model.

These callbacks expect to find an `'if'` field in the `input` `MPC`, where `MPC.if` is a `struct` with the following fields:

map	n x 2, defines each interface in terms of a <code>set</code> of branch indices and directions. Interface <code>I</code> is defined by the <code>set</code> of <code>rows</code> whose 1st col is equal to <code>I</code> . The 2nd column is a branch <code>index</code> multiplied by 1 or -1 respectively <code>for</code> lines whose orientation is the same as or opposite to that of the interface.
lims	nif x 3, defines the DC model flow limits in MW <code>for</code> specified interfaces. The first column is the <code>index</code> of the interface, the 2nd and 3rd <code>columns</code> specify the <code>lower</code> and <code>upper</code> limits on the (DC model) flow across the interface, respectively. Normally, the <code>lower</code> limit is negative, indicating a flow in the opposite direction.

The `'int2ext'` callback also packages up results and stores them in the following output fields of `results.if`:

P	- nif x 1, actual flow across each interface in MW
---	--

(continues on next page)

(continued from previous page)

```
mu.l    - nif x 1, shadow price on lower flow limit, ($/MW)
mu.u    - nif x 1, shadow price on upper flow limit, ($/MW)
```

See also [add_userfcn\(\)](#) (page 305), [remove_userfcn\(\)](#) (page 307), [run_userfcn\(\)](#) (page 307), [t_case30_userfcns\(\)](#) (page 384).

toggle_reserves

toggle_reserves(*mpc, on_off*)

[toggle_reserves\(\)](#) (page 309) - Enable, disable or check status of fixed reserve requirements.

```
MPC = TOGGLE_RESERVES(MPC, 'on')
MPC = TOGGLE_RESERVES(MPC, 'off')
T_F = TOGGLE_RESERVES(MPC, 'status')
```

Enables, disables or checks the status of a set of OPF userfcn callbacks to implement co-optimization of reserves with fixed zonal reserve requirements.

These callbacks expect to find a 'reserves' field in the input MPC, where MPC.reserves is a struct with the following fields:

```
zones    nrz x ng, zone(i, j) = 1, if gen j belongs to zone i
          0, otherwise
req       nrz x 1, zonal reserve requirement in MW
cost      (ng or ngr) x 1, cost of reserves in $/MW
qty       (ng or ngr) x 1, max quantity of reserves in MW (optional)
```

where nrz is the number of reserve zones and ngr is the number of generators belonging to at least one reserve zone and ng is the total number of generators.

The 'int2ext' callback also packages up results and stores them in the following output fields of results.reserves:

```
R         - ng x 1, reserves provided by each gen in MW
Rmin      - ng x 1, lower limit on reserves provided by each gen, (MW)
Rmax      - ng x 1, upper limit on reserves provided by each gen, (MW)
mu.l      - ng x 1, shadow price on reserve lower limit, ($/MW)
mu.u      - ng x 1, shadow price on reserve upper limit, ($/MW)
mu.Pmax   - ng x 1, shadow price on Pg + R <= Pmax constraint, ($/MW)
prc       - ng x 1, reserve price for each gen equal to maximum of the
            shadow prices on the zonal requirement constraint
            for each zone the generator belongs to
```

See also [runopf_w_res\(\)](#) (page 238), [add_userfcn\(\)](#) (page 305), [remove_userfcn\(\)](#) (page 307), [run_userfcn\(\)](#) (page 307), [t_case30_userfcns\(\)](#) (page 384).

toggle_softlims**toggle_softlims**(*mpc, on_off*)*toggle_softlims()* (page 310) - Relax DC optimal power flow branch limits.

```

MPC = TOGGLE_SOFTLIMS(MPC, 'on')
MPC = TOGGLE_SOFTLIMS(MPC, 'off')
T_F = TOGGLE_SOFTLIMS(MPC, 'status')

```

Enables, disables or checks the status of a **set** of OPF userfcn callbacks to implement relaxed inequality constraints **for** an OPF model.

These callbacks expect to **find** a 'softlims' field in the **input** MPC, where MPC.softlims is a **struct** with fields corresponding to the possible limits, namely:

VMIN, VMAX, RATE_A, PMIN, PMAX, QMIN, QMAX, ANGMAX, ANGMIN

Each of these is itself a **struct** with the following fields, **all** of which are optional:

idx	index of affected buses, branches, or generators. These are row indices into the respective matrices. The default is to include all online elements for which the constraint in question is not unbounded, except for generators, which also exclude those used to model dispatchable loads (i.e. those for which isload(gen) is true).
busnum	for bus constraints, such as VMIN and VMAX, the affected buses can be specified by a vector of external bus numbers in the 'busnum' field instead of bus row indices in the 'idx' field. If both are present, 'idx' overrides 'busnum'.
cost	linear marginal cost of exceeding the original limit The defaults are set as: base_cost x 100 \$/pu for VMAX and VMIN base_cost \$/MW for RATE_A, PMAX, and PMIN base_cost \$/MVar for QMAX, QMIN base_cost \$/deg for ANGMAX, ANGMIN where base_cost is the maximum of \$1000 and twice the maximum generator cost of all online generators.
hl_mod	type of modification to hard limit, hl: 'none' : do *not* add soft limit, no change to original hard limit 'remove' : add soft limit, relax hard limit by removing it completely 'replace' : add soft limit, relax hard limit by replacing original with value specified in hl_val 'scale' : add soft limit, relax hard limit by scaling original by value specified in hl_val 'shift' : add soft limit, relax hard limit by shifting original by value specified in hl_val
hl_val	value used to modify hard limit according to hl_mod. Ignored for 'none' and 'remove', required for 'replace', and optional, with the following defaults, for 'scale' and 'shift': 'scale' : 2 for positive upper limits or negative lower limits, 0.5 otherwise 'shift' : 0.25 for VMAX and VMIN, 10 otherwise

(continues on next page)

(continued from previous page)

For limits that are left unspecified in the structure, the default behavior is determined by the value of the `mpopt.opf.softlims.default` option. If `mpopt.opf.softlims.default = 0`, then the unspecified softlims are ignored (`hl_mod = 'none'`, i.e. original hard limits left in place). If `mpopt.opf.softlims.default = 1` (default), then the unspecified softlims are enabled with default values, which specify to `'remove'` the hard limit, except in the case of VMIN and PMIN, whose defaults are `set` as follows:

```
.VMIN
    .hl_mod = 'replace'
    .hl_val = 0
.PMIN
    .hl_mod = 'replace'
    .hl_val = 0    for normal generators (PMIN > 0)
    .hl_val = -Inf for generators with PMIN < 0 AND PMAX > 0
```

With `mpopt.opf.softlims.default = 0`, it is still possible to enable a softlim with default values by setting that specification to an empty `struct`. E.g. `mpc.softlims.VMAX = struct()` would enable a default softlim on VMAX.

The `'int2ext'` callback also packages up results and stores them in the following output fields of `results.softlims.(lim)`, where `lim` is one of the above mentioned limits:

```
overload - amount of overload, i.e. violation of hard-limit.
ovl_cost  - total cost of overload in $/hr
```

The shadow prices on the soft limit constraints are also returned in the relevant columns of the respective matrices (`MU_SF`, `MU_ST` for `RATE_A`, `MU_VMAX` for `VMAX`, etc.)

Note: These shadow prices are equal to the corresponding hard limit shadow prices when the soft limits are `not` violated. When violated, the shadow price on a soft limit constraint is equal to the user-specified soft limit violation cost + the shadow price on any binding remaining hard limit.

See also `add_userfcn()` (page 305), `remove_userfcn()` (page 307), `run_userfcn()` (page 307), `t_opf_softlims()` (page 381).

5.2.9 Power Flow Derivative Functions

dIbr_dV

dIbr_dV(branch, Yf, Yt, V, vcart)

dIbr_dV() (page 312) - Computes partial derivatives of branch currents w.r.t. voltage.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the 5th argument.

```
[DIF_DVA, DIF_DVM, DIT_DVA, DIT_DVM, IF, IT] = DIBR_DV(BRANCH, YF, YT, V)
[DIF_DVA, DIF_DVM, DIT_DVA, DIT_DVM, IF, IT] = DIBR_DV(BRANCH, YF, YT, V, 0)
```

Returns four matrices containing partial derivatives of the **complex** branch currents at "from" and "to" ends of each branch w.r.t voltage magnitude and voltage **angle**, respectively (**for all** buses).

```
[DIF_DVR, DIF_DVI, DIT_DVR, DIT_DVI, IF, IT] = DIBR_DV(BRANCH, YF, YT, V, 1)
```

Returns four matrices containing partial derivatives of the **complex** branch currents at "from" and "to" ends of each branch w.r.t **real** and **imaginary** parts of voltage, respectively (**for all** buses).

If YF is a **sparse** matrix, the partial derivative matrices will be as well. Optionally returns vectors containing the currents themselves. The following explains the expressions used to form the matrices:

```
If = Yf * V;
```

Polar coordinates:

Partials of V, Vf & If w.r.t. voltage angles

```
dV/dVa = j * diag(V)
dVf/dVa = sparse(1:nl, f, j * V(f)) = j * sparse(1:nl, f, V(f))
dIf/dVa = Yf * dV/dVa = Yf * j * diag(V)
```

Partials of V, Vf & If w.r.t. voltage magnitudes

```
dV/dVm = diag(V./abs(V))
dVf/dVm = sparse(1:nl, f, V(f)./abs(V(f))
dIf/dVm = Yf * dV/dVm = Yf * diag(V./abs(V))
```

Cartesian coordinates:

Partials of V, Vf & If w.r.t. **real** part of **complex** voltage

```
dV/dVr = diag(ones(n,1))
dVf/dVr = Cf
dIf/dVr = Yf
```

where Cf is the connection matrix **for line** & from buses

Partials of V, Vf & If w.r.t. **imaginary** part of **complex** voltage

```
dV/dVi = j * diag(ones(n,1))
dVf/dVi = j * Cf
```

(continues on next page)

(continued from previous page)

```
dIf/dVi = j * Yf
```

Derivations for "to" bus are similar.

Example:

```
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[dIf_dVa, dIf_dVm, dIt_dVa, dIt_dVm, If, It] = ...
    dIbr_dV(branch, Yf, Yt, V);
[dIf_dVr, dIf_dVi, dIt_dVr, dIt_dVi, If, It] = ...
    dIbr_dV(branch, Yf, Yt, V, 1);
```

For more details on the derivations behind the derivative code used in MATPOWER information, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: 10.5281/zenodo.3237866
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: 10.5281/zenodo.3237909

dSbr_dV

dSbr_dV(branch, Yf, Yt, V, vcart)

dSbr_dV() (page 313) - Computes partial derivatives of branch power flows w.r.t. voltage.

The derivatives can be taken with respect to polar or cartesian coordinates of voltage, depending on the 5th argument.

```
[DSF_DVA, DSF_DVM, DST_DVA, DST_DVM, SF, ST] = DSBR_DV(BRANCH, YF, YT, V)
[DSF_DVA, DSF_DVM, DST_DVA, DST_DVM, SF, ST] = DSBR_DV(BRANCH, YF, YT, V, 0)
```

Returns four matrices containing partial derivatives of the complex branch power flows at "from" and "to" ends of each branch w.r.t voltage magnitude and voltage angle, respectively (for all buses).

```
[DSF_DVR, DSF_DVI, DST_DVR, DST_DVI, SF, ST] = DSBR_DV(BRANCH, YF, YT, V, 1)
```

Returns four matrices containing partial derivatives of the complex branch power flows at "from" and "to" ends of each branch w.r.t real and imaginary parts of voltage, respectively (for all buses).

If YF is a sparse matrix, the partial derivative matrices will be as well. Optionally returns vectors containing the power flows themselves. The following explains the expressions used to form the matrices:

(continues on next page)

(continued from previous page)

```

If = Yf * V;
Sf = diag(Vf) * conj(If) = diag(conj(If)) * Vf

Polar coordinates:
Partials of V, Vf & If w.r.t. voltage angles
dV/dVa = j * diag(V)
dVf/dVa = sparse(1:nl, f, j * V(f)) = j * sparse(1:nl, f, V(f))
dIf/dVa = Yf * dV/dVa = Yf * j * diag(V)

Partials of V, Vf & If w.r.t. voltage magnitudes
dV/dVm = diag(V./abs(V))
dVf/dVm = sparse(1:nl, f, V(f)./abs(V(f)))
dIf/dVm = Yf * dV/dVm = Yf * diag(V./abs(V))

Partials of Sf w.r.t. voltage angles
dSf/dVa = diag(Vf) * conj(dIf/dVa)
          + diag(conj(If)) * dVf/dVa
        = diag(Vf) * conj(Yf * j * diag(V))
          + conj(diag(If)) * j * sparse(1:nl, f, V(f))
        = -j * diag(Vf) * conj(Yf * diag(V))
          + j * conj(diag(If)) * sparse(1:nl, f, V(f))
        = j * (conj(diag(If)) * sparse(1:nl, f, V(f))
          - diag(Vf) * conj(Yf * diag(V)))

Partials of Sf w.r.t. voltage magnitudes
dSf/dVm = diag(Vf) * conj(dIf/dVm)
          + diag(conj(If)) * dVf/dVm
        = diag(Vf) * conj(Yf * diag(V./abs(V)))
          + conj(diag(If)) * sparse(1:nl, f, V(f)./abs(V(f)))

Cartesian coordinates:
Partials of V, Vf & If w.r.t. real part of complex voltage
dV/dVr = diag(ones(n,1))
dVf/dVr = Cf
dIf/dVr = Yf
where Cf is the connection matrix for line & from buses

Partials of V, Vf & If w.r.t. imaginary part of complex voltage
dV/dVi = j * diag(ones(n,1))
dVf/dVi = j * Cf
dIf/dVi = j * Yf

Partials of Sf w.r.t. real part of complex voltage
dSf/dVr = conj(diag(If)) * Cf + diag(Vf) * conj(Yf)

Partials of Sf w.r.t. imaginary part of complex voltage
dSf/dVi = j * (conj(diag(If)) * Cf - diag(Vf) * conj(Yf))

Derivations for "to" bus are similar.

```

Examples:

(continues on next page)

(continued from previous page)

```
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[dSf_dVa, dSf_dVm, dSt_dVa, dSt_dVm, Sf, St] = ...
    dSbr_dV(branch, Yf, Yt, V);
[dSf_dVr, dSf_dVi, dSt_dVr, dSt_dVi, Sf, St] = ...
    dSbr_dV(branch, Yf, Yt, V, 1);
```

For more details on the derivations behind the derivative code used in MATPOWER, see:

[TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)

[TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

dAbr_dV

dAbr_dV(dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft)

dAbr_dV() (page 315) - Partial derivatives of squared flow magnitudes w.r.t voltage.

```
[DAF_DV1, DAF_DV2, DAT_DV1, DAT_DV2] = ...
    DABR_DV(DFF_DV1, DFF_DV2, DFT_DV1, DFT_DV2, FF, FT)
```

returns four matrices containing partial derivatives of the square of the branch flow magnitudes at "from" & "to" ends of each branch w.r.t voltage components (either **angle** and magnitude, respectively, **if polar**, or **real** and imaginary, respectively, **if cartesian**) **for all** buses, given the flows and flow sensitivities. Flows could be **complex current** or **complex or real power**. Notation below is based on **complex power**. The following explains the expressions used to form the matrices:

Let Af refer to the square of the apparent **power** at the "from" end of each branch,

```
Af = abs(Sf).^2
    = Sf .* conj(Sf)
    = Pf.^2 + Qf.^2
```

then ...

Partial w.r.t **real power**,
 $dA_f/dP_f = 2 * \text{diag}(P_f)$

Partial w.r.t **reactive power**,
 $dA_f/dQ_f = 2 * \text{diag}(Q_f)$

(continues on next page)

(continued from previous page)

Partial w.r.t V1 & V2 (e.g. Va and Vm, or Vr and Vi)

$$dAf/dV1 = dAf/dPf * dPf/dV1 + dAf/dQf * dQf/dV1$$

$$dAf/dV2 = dAf/dPf * dPf/dV2 + dAf/dQf * dQf/dV2$$

Derivations for "to" bus are similar.

Examples:

%% squared current magnitude

```
[dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft] = ...
```

```
    dIbr_dV(branch(il,:), Yf, Yt, V);
```

```
[dAf_dV1, dAf_dV2, dAt_dV1, dAt_dV2] = ...
```

```
    dAbr_dV(dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft);
```

%% squared apparent power flow

```
[dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft] = ...
```

```
    dSbr_dV(branch(il,:), Yf, Yt, V);
```

```
[dAf_dV1, dAf_dV2, dAt_dV1, dAt_dV2] = ...
```

```
    dAbr_dV(dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft);
```

%% squared real power flow

```
[dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft] = ...
```

```
    dSbr_dV(branch(il,:), Yf, Yt, V);
```

```
dFf_dV1 = real(dFf_dV1);
```

```
dFf_dV2 = real(dFf_dV2);
```

```
dFt_dV1 = real(dFt_dV1);
```

```
dFt_dV2 = real(dFt_dV2);
```

```
[dAf_dV1, dAf_dV2, dAt_dV1, dAt_dV2] = ...
```

```
    dAbr_dV(dFf_dV1, dFf_dV2, dFt_dV1, dFt_dV2, Ff, Ft);
```

See also *dIbr_dV()* (page 312), *dSbr_dV()* (page 313).

For more details on the derivations behind the derivative code used in MATPOWER information, see:

[TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: 10.5281/zenodo.3237866

dImis_dV

dImis_dV(Sbus, Ybus, V, vcart)

dImis_dV() (page 316) - Computes partial derivatives of current balance w.r.t. voltage.

The derivatives can be take with respect to polar or cartesian coordinates of voltage, depending on the 3rd argument.

```
[DIMIS_DVM, DIMIS_DVA] = DIMIS_DV(SBUS, YBUS, V)
```

```
[DIMIS_DVM, DIMIS_DVA] = DIMIS_DV(SBUS, YBUS, V, 0)
```

(continues on next page)

(continued from previous page)

Returns two matrices containing partial derivatives of the **complex** bus current **balance** w.r.t voltage magnitude and voltage **angle**, respectively (**for all** buses).

```
[DIMIS_DVR, DIMIS_DVI] = DIMIS_DV(SBUS, YBUS, V, 1)
```

Returns two matrices containing partial derivatives of the **complex** bus current **balance** w.r.t the **real** and imaginary parts of voltage, respectively (**for all** buses).

If YBUS is a **sparse** matrix, the **return** values will be also. The following explains the expressions used to form the matrices:

$$I_{mis} = I_{bus} + I_{dg} = Y_{bus} * V - \text{conj}(S_{bus}./V)$$

Polar coordinates:

Partials of V & I_{bus} w.r.t. voltage angles

$$dV/dV_a = j * \text{diag}(V)$$

$$dI/dV_a = Y_{bus} * dV/dV_a = Y_{bus} * j * \text{diag}(V)$$

Partials of V & I_{bus} w.r.t. voltage magnitudes

$$dV/dV_m = \text{diag}(V./\text{abs}(V))$$

$$dI/dV_m = Y_{bus} * dV/dV_m = Y_{bus} * \text{diag}(V./\text{abs}(V))$$

Partials of I_{mis} w.r.t. voltage angles

$$dI_{mis}/dV_a = j * (Y_{bus} * \text{diag}(V) - \text{diag}(\text{conj}(S_{bus}./V)))$$

Partials of I_{mis} w.r.t. voltage magnitudes

$$dI_{mis}/dV_m = Y_{bus} * \text{diag}(V./\text{abs}(V)) + \text{diag}(\text{conj}(S_{bus}./(V * \text{abs}(V))))$$

Cartesian coordinates:

Partials of V & I_{bus} w.r.t. **real** part of **complex** voltage

$$dV/dV_r = \text{diag}(\text{ones}(n,1))$$

$$dI/dV_r = Y_{bus} * dV/dV_r = Y_{bus}$$

Partials of V & I_{bus} w.r.t. imaginary part of **complex** voltage

$$dV/dV_i = j * \text{diag}(\text{ones}(n,1))$$

$$dI/dV_i = Y_{bus} * dV/dV_i = Y_{bus} * j$$

Partials of I_{mis} w.r.t. **real** part of **complex** voltage

$$dI_{mis}/dV_r = Y_{bus} + \text{conj}(\text{diag}(S_{bus}./(V.^2)))$$

Partials of S w.r.t. imaginary part of **complex** voltage

$$dI_{mis}/dV_i = j * (Y_{bus} - \text{diag}(\text{conj}(S_{bus}./(V.^2))))$$

Examples:

```
Sbus = makeSbus(baseMVA, bus, gen);
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[dImis_dVm, dImis_dVa] = dImis_dV(Sbus, Ybus, V);
[dImis_dVr, dImis_dVi] = dImis_dV(Sbus, Ybus, V, 1);
```

For more details on the derivations behind the derivative code used in MATPOWER information, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)
- [TN3] B. Sereeter and R. D. Zimmerman, "Addendum to AC Power Flows and their Derivatives using Complex Matrix Notation: Nodal Current Balance," MATPOWER Technical Note 3, April 2018. [Online]. Available: <https://matpower.org/docs/TN3-More-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237900](https://doi.org/10.5281/zenodo.3237900)
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

dSbus_dV

dSbus_dV(Ybus, V, vcart)

dSbus_dV() (page 318) - Computes partial derivatives of power injection w.r.t. voltage.

The derivatives can be taken with respect to **polar or** cartesian coordinates of voltage, depending on the 3rd argument.

```
[DSBUS_DVA, DSBUS_DVM] = DSBUS_DV(YBUS, V)
[DSBUS_DVA, DSBUS_DVM] = DSBUS_DV(YBUS, V, 0)
```

Returns two matrices containing partial derivatives of the **complex** bus power injections w.r.t voltage **angle and** voltage magnitude, respectively (**for all** buses).

```
[DSBUS_DVR, DSBUS_DVI] = DSBUS_DV(YBUS, V, 1)
```

Returns two matrices containing partial derivatives of the **complex** bus power injections w.r.t the **real and** imaginary parts of voltage, respectively (**for all** buses).

If YBUS is a **sparse** matrix, the **return** values will be also. The following explains the expressions used to form the matrices:

$$S = \text{diag}(V) * \text{conj}(I_{\text{bus}}) = \text{diag}(\text{conj}(I_{\text{bus}})) * V$$

Polar coordinates:

Partials of V & Ibus w.r.t. voltage magnitudes
 $dV/dV_m = \text{diag}(V./\text{abs}(V))$
 $dI/dV_m = Y_{\text{bus}} * dV/dV_m = Y_{\text{bus}} * \text{diag}(V./\text{abs}(V))$

Partials of V & Ibus w.r.t. voltage angles
 $dV/dV_a = j * \text{diag}(V)$

(continues on next page)

(continued from previous page)

```

dI/dVa = Ybus * dV/dVa = Ybus * j * diag(V)

Partials of S w.r.t. voltage magnitudes
dS/dVm = diag(V) * conj(dI/dVm) + diag(conj(Ibus)) * dV/dVm
        = diag(V) * conj(Ybus * diag(V./abs(V)))
          + conj(diag(Ibus)) * diag(V./abs(V))

Partials of S w.r.t. voltage angles
dS/dVa = diag(V) * conj(dI/dVa) + diag(conj(Ibus)) * dV/dVa
        = diag(V) * conj(Ybus * j * diag(V))
          + conj(diag(Ibus)) * j * diag(V)
        = -j * diag(V) * conj(Ybus * diag(V))
          + conj(diag(Ibus)) * j * diag(V)
        = j * diag(V) * conj(diag(Ibus) - Ybus * diag(V))

Cartesian coordinates:
Partials of V & Ibus w.r.t. real part of complex voltage
dV/dVr = diag(ones(n,1))
dI/dVr = Ybus * dV/dVr = Ybus

Partials of V & Ibus w.r.t. imaginary part of complex voltage
dV/dVi = j * diag(ones(n,1))
dI/dVi = Ybus * dV/dVi = Ybus * j

Partials of S w.r.t. real part of complex voltage
dS/dVr = diag(V) * conj(dI/dVr) + diag(conj(Ibus)) * dV/dVr
        = diag(V) * conj(Ybus) + conj(diag(Ibus))

Partials of S w.r.t. imaginary part of complex voltage
dS/dVi = diag(V) * conj(dI/dVi) + diag(conj(Ibus)) * dV/dVi
        = j * (conj(diag(Ibus)) - diag(V) conj(Ybus))

Examples:
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[dSbus_dVa, dSbus_dVm] = dSbus_dV(Ybus, V);
[dSbus_dVr, dSbus_dVi] = dSbus_dV(Ybus, V, 1);

```

For more details on the derivations behind the derivative code used in MATPOWER information, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: 10.5281/zenodo.3237866
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: 10.5281/zenodo.3237909

d2Ibr_dV2**d2Ibr_dV2**(Ybr, V, mu, vcart)*d2Ibr_dV2*() (page 320) - Computes 2nd derivatives of complex branch current w.r.t. voltage.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the 4th argument.

```
[HAA, HAV, HVA, HVV] = D2IBR_DV2(YBR, V, MU)
[HAA, HAV, HVA, HVV] = D2IBR_DV2(YBR, V, MU, 0)
```

Returns 4 matrices containing the partial derivatives w.r.t. voltage **angle** and magnitude of the product of a vector MU with the 1st partial derivatives of the **complex** branch currents.

```
[HRR, HRI, HIR, HII] = D2IBR_DV2(YBR, V, MU, 1)
```

Returns 4 matrices (**all zeros**) containing the partial derivatives w.r.t. **real** and imaginary part of **complex** voltage of the product of a vector MU with the 1st partial derivatives of the **complex** branch currents.

Takes **sparse** branch admittance matrix YBR, voltage vector V and nl x 1 vector of multipliers MU. Output matrices are **sparse**.

Examples:

```
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
Ybr = Yf;
[Haa, Hav, Hva, Hvv] = d2Ibr_dV2(Ybr, V, mu);
```

Here the output matrices correspond to:

```
Haa = d/dVa (dIbr_dVa.' * mu)
Hav = d/dVm (dIbr_dVa.' * mu)
Hva = d/dVa (dIbr_dVm.' * mu)
Hvv = d/dVm (dIbr_dVm.' * mu)
```

```
[Hrr, Hri, Hir, Hii] = d2Ibr_dV2(Ybr, V, mu, 1);
```

Here the output matrices correspond to:

```
Hrr = d/dVr (dIbr_dVr.' * mu)
Hri = d/dVi (dIbr_dVr.' * mu)
Hir = d/dVr (dIbr_dVi.' * mu)
Hii = d/dVi (dIbr_dVi.' * mu)
```

For more details on the derivations behind the derivative code used in MATPOWER information, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf> doi: 10.5281/zenodo.3237866
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018.

(continues on next page)

(continued from previous page)

→pdf [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian>.
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

d2Sbr_dV2

d2Sbr_dV2(Cbr, Ybr, V, mu, vcart)

d2Sbr_dV2() (page 321) - Computes 2nd derivatives of complex brch power flow w.r.t. voltage.

The derivatives can be take with respect to **polar or** cartesian coordinates of voltage, depending on the 5th argument.

```
[HAA, HAV, HVA, HVV] = D2SBR_DV2(CBR, YBR, V, MU)
[HAA, HAV, HVA, HVV] = D2SBR_DV2(CBR, YBR, V, MU, 0)
```

Returns 4 matrices containing the partial derivatives w.r.t. voltage **angle and** magnitude of the product of a vector MU with the 1st partial derivatives of the **complex** branch **power** flows.

```
[HRR, HRI, HIR, HII] = d2Sbr_dV2(CBR, YBR, V, MU, 1)
```

Returns 4 matrices containing the partial derivatives w.r.t. **real and** imaginary part of **complex** voltage of the product of a vector MU with the 1st partial derivatives of the **complex** branch **power** flows.

Takes **sparse** connection matrix CBR, **sparse** branch admittance matrix YBR, voltage vector V and nl x 1 vector of multipliers MU. Output matrices are **sparse**.

Examples:

```
f = branch(:, F_BUS);
Cf = sparse(1:nl, f, ones(nl, 1), nl, nb);
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
Cbr = Cf;
Ybr = Yf;
[Haa, Hav, Hva, Hvv] = d2Sbr_dV2(Cbr, Ybr, V, mu);
```

Here the output matrices correspond to:

```
Haa = d/dVa (dSbr_dVa.' * mu)
Hav = d/dVm (dSbr_dVa.' * mu)
Hva = d/dVa (dSbr_dVm.' * mu)
Hvv = d/dVm (dSbr_dVm.' * mu)
```

```
[Hrr, Hri, Hir, Hii] = d2Sbr_dV2(Cbr, Ybr, V, mu, 1);
```

Here the output matrices correspond to:

```
Hrr = d/dVr (dSbr_dVr.' * mu)
Hri = d/dVi (dSbr_dVr.' * mu)
Hir = d/dVr (dSbr_dVi.' * mu)
Hii = d/dVi (dSbr_dVi.' * mu)
```

For more details on the derivations behind the derivative code used in MATPOWER information, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

d2Abr_dV2

d2Abr_dV2(d2F_dV2, dF_dV1, dF_dV2, F, V, mu)

d2Abr_dV2() (page 322) - Computes 2nd derivatives of |branch flow|^2 w.r.t. V.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the first 3 arguments. Flows could be **complex** current or **complex** or **real** power. Notation below is based on **complex** power.

```
[H11, H12, H21, H22] = D2ABR_DV2(D2F_DV2, DF_DV1, DF_DV2, F, V, MU)
```

Returns 4 matrices containing the partial derivatives w.r.t. voltage components (**angle**, magnitude or **real**, imaginary) of the product of a vector MU with the 1st partial derivatives of the square of the magnitude of branch flows.

Takes as inputs a handle to a **function** that evaluates the 2nd derivatives of the flows (with args V and mu only), **sparse** first derivative matrices of flow, flow vector, voltage vector V and nl x 1 vector of multipliers MU. Output matrices are **sparse**.

Example:

```
f = branch(:, F_BUS);
Cf = sparse(1:nl, f, ones(nl, 1), nl, nb);
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[dSf_dV1, dSf_dV2, dSt_dV1, dSt_dV2, Sf, St] = ...
    dSbr_dV(branch, Yf, Yt, V);
dF_dV1 = dSf_dV1;
dF_dV2 = dSf_dV2;
F = Sf;
d2F_dV2 = @(V, mu)d2Sbr_dV2(Cf, Yf, V, mu, 0);
[H11, H12, H21, H22] = ...
    d2Abr_dV2(d2F_dV2, dF_dV1, dF_dV2, F, V, mu);
```

Here the output matrices correspond to:

```
H11 = d/dV1 (dAF_dV1.' * mu)
H12 = d/dV2 (dAF_dV1.' * mu)
```

(continues on next page)

(continued from previous page)

```
H21 = d/dV1 (dAF_dV2.' * mu)
H22 = d/dV2 (dAF_dV2.' * mu)
```

See also [dAbr_dV\(\)](#) (page 315), [dIbr_dV\(\)](#) (page 312), [dSbr_dV\(\)](#) (page 313).

For more details on the derivations behind the derivative code used in MATPOWER information, see:

[TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)

d2Imis_dV2

d2Imis_dV2(Sbus, Ybus, V, lam, vcart)

[d2Imis_dV2\(\)](#) (page 323) - Computes 2nd derivatives of current balance w.r.t. voltage.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the 5th argument.

```
[GAA, GAV, GVA, GVV] = D2IMIS_DV2(SBUS, YBUS, V, LAM)
[GAA, GAV, GVA, GVV] = D2IMIS_DV2(SBUS, YBUS, V, LAM, 0)
```

Returns 4 matrices containing the partial derivatives w.r.t. voltage **angle** and magnitude of the product of a vector LAM with the 1st partial derivatives of the **complex** bus current **balance**.

```
[GRR, GIR, GRI, GII] = D2IMIS_DV2(SBUS, YBUS, V, LAM, 1)
```

Returns 4 matrices containing the partial derivatives w.r.t. **real** and **imaginary** parts of voltage of the product of a vector LAM with the 1st partial derivatives of the **complex** bus current **balance**.

Takes bus **complex power** injection (gen-load) vector, **sparse** bus admittance matrix YBUS, voltage vector V and nb x 1 vector of multipliers LAM. Output matrices are **sparse**.

Examples:

```
Sbus = makeSbus(baseMVA, bus, gen);
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[Gaa, Gav, Gva, Gvv] = d2Imis_dV2(Sbus, Ybus, V, lam);
```

Here the output matrices correspond to:

```
Gaa = d/dVa (dImis_dVa.' * lam)
Gav = d/dVm (dImis_dVa.' * lam)
Gva = d/dVa (dImis_dVm.' * lam)
Gvv = d/dVm (dImis_dVm.' * lam)
```

```
[Grr, Gri, Gir, Gii] = d2Imis_dV2(Sbus, Ybus, V, lam, 1);
```

(continues on next page)

(continued from previous page)

Here the output matrices correspond to:

```
Grr = d/dVr (dImis_dVr.' * lam)
Gri = d/dVi (dImis_dVr.' * lam)
Gir = d/dVr (dImis_dVi.' * lam)
Gii = d/dVi (dImis_dVi.' * lam)
```

For more details on the derivations behind the derivative code used in MATPOWER, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)
- [TN3] B. Sereeter and R. D. Zimmerman, "Addendum to AC Power Flows and their Derivatives using Complex Matrix Notation: Nodal Current Balance," MATPOWER Technical Note 3, April 2018. [Online]. Available: <https://matpower.org/docs/TN3-More-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237900](https://doi.org/10.5281/zenodo.3237900)
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

d2Imis_dVdSg

d2Imis_dVdSg(*Cg, V, lam, vcart*)

d2Imis_dVdSg() (page 324) - Computes 2nd derivatives of current balance w.r.t. V and Sg.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the 4th argument.

```
GSV = D2IMIS_DVDSG(CG, V, LAM)
GSV = D2IMIS_DVDSG(CG, V, LAM, 0)
```

Returns a matrix containing the partial derivatives w.r.t. voltage **angle** and magnitude of the product of a vector LAM with the 1st partial derivatives of the **real** and reactive **power** generation.

```
GSV = D2IMIS_DVDSG(CG, V, LAM, 1)
```

Returns a matrix containing the partial derivatives w.r.t. **real** and imaginary parts of voltage of the product of a vector LAM with the 1st partial derivatives of the **real** and reactive **power** generation.

Takes the generator connection matrix, **complex** voltage vector V and nb x 1 vector of multipliers LAM. Output matrices are **sparse**.

(continues on next page)

(continued from previous page)

Examples:

```
Cg = sparse(gen(:, GEN_BUS), 1:ng, -, nb, ng);
Gsv = d2Imis_dVdSg(Cg, V, lam);
```

Here the output matrix corresponds to:

```
Gsv = [ Gpa Gpv;
        Gqa Gqv ];
Gpa = d/dVa (dImis_dPg.' * lam)
Gpv = d/dVm (dImis_dPg.' * lam)
Gqa = d/dVa (dImis_dQg.' * lam)
Gqv = d/dVm (dImis_dQg.' * lam)
```

```
[Grr, Gri, Gir, Gii] = d2Imis_dVdSg(Cg, V, lam, 1);
```

Here the output matrices correspond to:

```
Gsv = [ Gpr Gpi;
        Gqr Gqi ];
Gpr = d/dVr (dImis_dPg.' * lam)
Gpi = d/dVi (dImis_dPg.' * lam)
Gqr = d/dVr (dImis_dQg.' * lam)
Gqi = d/dVi (dImis_dQg.' * lam)
```

For more details on the derivations behind the derivative code used in MATPOWER, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237866](https://doi.org/10.5281/zenodo.3237866)
- [TN3] B. Sereeter and R. D. Zimmerman, "Addendum to AC Power Flows and their Derivatives using Complex Matrix Notation: Nodal Current Balance," MATPOWER Technical Note 3, April 2018. [Online]. Available: <https://matpower.org/docs/TN3-More-OPF-Derivatives.pdf>
doi: [10.5281/zenodo.3237900](https://doi.org/10.5281/zenodo.3237900)
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>
doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

d2Sbus_dV2**d2Sbus_dV2**(Ybus, V, lam, vcart)*d2Sbus_dV2*() (page 326) - Computes 2nd derivatives of power injection w.r.t. voltage.

The derivatives can be take with respect to **polar** or cartesian coordinates of voltage, depending on the 4th argument.

```
[GAA, GAV, GVA, GVV] = D2SBUS_DV2(YBUS, V, LAM)
[GAA, GAV, GVA, GVV] = D2SBUS_DV2(YBUS, V, LAM, 0)
```

Returns 4 matrices containing the partial derivatives w.r.t. voltage **angle** and magnitude of the product of a vector LAM with the 1st partial derivatives of the **complex** bus **power** injections.

```
[GRR, GIR, GRI, GII] = D2SBUS_DV2(YBUS, V, LAM, 1)
```

Returns 4 matrices containing the partial derivatives w.r.t. **real** and imaginary parts of voltage of the product of a vector LAM with the 1st partial derivatives of the **complex** bus **power** injections.

Takes **sparse** bus admittance matrix YBUS, voltage vector V and nb x 1 vector of multipliers LAM. Output matrices are **sparse**.

Examples:

```
[Ybus, Yf, Yt] = makeYbus(baseMVA, bus, branch);
[Gaa, Gav, Gva, Gvv] = d2Sbus_dV2(Ybus, V, lam);
```

Here the output matrices correspond to:

```
Gaa = d/dVa (dSbus_dVa.' * lam)
Gav = d/dVm (dSbus_dVa.' * lam)
Gva = d/dVa (dSbus_dVm.' * lam)
Gvv = d/dVm (dSbus_dVm.' * lam)
```

```
[Grr, Gri, Gir, Gii] = d2Sbus_dV2(Ybus, V, lam, 1);
```

Here the output matrices correspond to:

```
Grr = d/dVr (dSbus_dVr.' * lam)
Gri = d/dVi (dSbus_dVr.' * lam)
Gir = d/dVr (dSbus_dVi.' * lam)
Gii = d/dVi (dSbus_dVi.' * lam)
```

For more details on the derivations behind the derivative code used in MATPOWER, see:

- [TN2] R. D. Zimmerman, "AC Power Flows, Generalized OPF Costs and their Derivatives using Complex Matrix Notation", MATPOWER Technical Note 2, February 2010. [Online]. Available: <https://matpower.org/docs/TN2-OPF-Derivatives.pdf> doi: 10.5281/zenodo.3237866
- [TN4] B. Sereeter and R. D. Zimmerman, "AC Power Flows and their Derivatives using Complex Matrix Notation and Cartesian Coordinate Voltages," MATPOWER Technical Note 4, April 2018. [Online]. Available: <https://matpower.org/docs/TN4-OPF-Derivatives-Cartesian.pdf>

(continues on next page)

(continued from previous page)

[↪pdf](#)doi: [10.5281/zenodo.3237909](https://doi.org/10.5281/zenodo.3237909)

5.2.10 LP, QP, MILP & MIQP Solver Functions

miqps_matpower

miqps_matpower(varargin)*miqps_matpower()* (page 327) - Deprecated, please use *miqps_master()* instead.

qps_matpower

qps_matpower(varargin)*qps_matpower()* (page 327) - Deprecated, please use *qps_master()* instead.

5.2.11 Matrix Building Functions

makeB

makeB(baseMVA, bus, branch, alg)*makeB()* (page 327) - Builds the FDPF matrices, B prime and B double prime.

```
[BP, BPP] = MAKEB(MPC, ALG)
[BP, BPP] = MAKEB(BASEMVA, BUS, BRANCH, ALG)
```

Returns the two matrices B prime and B double prime used in the fast decoupled power flow. Does appropriate conversions to p.u. ALG is either 'FDXB' or 'FDBX', the corresponding value of MPOPT.pf.alg option specifying the power flow algorithm. Bus numbers must be consecutive beginning at 1 (i.e. internal ordering).

Note: For backward compatibility, ALG can also take on a value of 2 or 3, corresponding to values of the old PF_ALG option. This usage is deprecated and will be removed in a future version.

Example:

```
[Bp, Bpp] = makeB(baseMVA, bus, branch, 'FDXB');
```

See also *fdpf()*.

makeBdc

makeBdc(baseMVA, bus, branch)

[makeBdc\(\)](#) (page 328) - Builds the B matrices and phase shift injections for DC power flow.

```
[BBUS, BF, PBUSINJ, PFINJ] = MAKEBDC(MPC)
```

```
[BBUS, BF, PBUSINJ, PFINJ] = MAKEBDC(BASEMVA, BUS, BRANCH)
```

Returns the B matrices and phase shift injection vectors needed for a DC power flow. 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

$$Pf = BF * Va + PFINJ$$

Does appropriate conversions to p.u.

Bus numbers must be consecutive beginning at 1 (i.e. internal ordering).

Example:

```
[Bbus, Bf, Pbusinj, Pfinj] = makeBdc(baseMVA, bus, branch);
```

See also [dcpf\(\)](#).

makeJac

makeJac(baseMVA, bus, branch, gen, fullJac)

[makeJac\(\)](#) (page 328) - Forms the power flow Jacobian.

```
J = MAKEJAC(MPC)
```

```
J = MAKEJAC(MPC, FULLJAC)
```

```
J = MAKEJAC(BASEMVA, BUS, BRANCH, GEN)
```

```
J = MAKEJAC(BASEMVA, BUS, BRANCH, GEN, FULLJAC)
```

```
[J, YBUS, YF, YT] = MAKEJAC(MPC)
```

Returns the power flow Jacobian and, optionally, the system admittance matrices. Inputs can be a MATPOWER case struct or individual BASEMVA, BUS, BRANCH and GEN values. Bus numbers must be consecutive beginning at 1 (i.e. internal ordering). If the FULLJAC argument is present and true, it returns the full Jacobian (sensitivities of all bus injections w.r.t all voltage angles/magnitudes) as opposed to the reduced version used in the Newton power flow updates. The units for all quantities are in per unit with radians for voltage angles.

Note: This function builds the Jacobian from scratch, rebuilding the YBUS matrix in the process. You probably don't want to use this in performance critical code.

See also [makeYbus\(\)](#) (page 331), [ext2int\(\)](#).

makeLODF

makeLODF(*branch*, *PTDF*, *mask_bridge*)

makeLODF() (page 329) - Builds the line outage distribution factor matrix.

```
LODF = MAKELODF(BRANCH, PTDF)
LODF = MAKELODF(MPC, PTDF)
LODF = MAKELODF(MPC, PTDF, MASK_BRIDGE)
```

Returns the DC model **line** outage distribution **factor** matrix corresponding to a given PTDF matrix. The LODF matrix is *nbr* x *nbr*, where *nbr* is the number of branches. If the optional MASK_BRIDGE argument is **true**, **columns** corresponding to bridge branches (those whose removal result in islanding) are replaced with **NaN**.

Example:

```
H = makePTDF(mpc);
LODF = makeLODF(branch, H);
LODF = makeLODF(mpc, H);

% mask bridge branches in LODF
makeLODF(mpc, H, 1);
```

See also *makePTDF()* (page 329), *find_bridges()* (page 337).

makePTDF

makePTDF(*baseMVA*, *bus*, *branch*, *slack*, *bus_idx*)

makeLODF() (page 329) - Builds the DC PTDF matrix for a given choice of slack.

```
H = MAKEPTDF(MPC)
H = MAKEPTDF(MPC, SLACK)
H = MAKEPTDF(MPC, SLACK, TXFR)
H = MAKEPTDF(MPC, SLACK, BUS_IDX)
H = MAKEPTDF(BASEMVA, BUS, BRANCH)
H = MAKEPTDF(BASEMVA, BUS, BRANCH, SLACK)
H = MAKEPTDF(BASEMVA, BUS, BRANCH, SLACK, TXFR)
H = MAKEPTDF(BASEMVA, BUS, BRANCH, SLACK, BUS_IDX)
```

Returns the DC PTDF matrix **for** a given choice of slack. The matrix is *nbr* x *nb*, where *nbr* is the number of branches **and** *nb* is the number of buses. The SLACK can be a scalar (**single** slack bus) **or** an *nb* x 1 column vector of weights specifying the proportion of the slack taken up at each bus. If the SLACK is **not** specified the reference bus is used by default. Bus numbers must be consecutive beginning at 1 (*i.e.* internal ordering).

For convenience, SLACK can also be an *nb* x *nb* matrix, where each column specifies how the slack should be handled **for** injections at that bus. This option only applies when computing the **full** PTDF matrix (*i.e.* when TXFR **and** BUS_IDX are **not** provided.)

(continues on next page)

(continued from previous page)

If TXFR is supplied it must be a matrix (or vector) with nb rows whose columns each sum to zero, where each column defines a specific (slack independent) transfer. E.g. if k-th transfer is from bus i to bus j, TXFR(i, k) = 1 and TXFR(j, k) = -1. In this case H has the same number of columns as TXFR.

If BUS_IDX is supplied, it contains a column vector of bus indices. The columns of H correspond to these indices, but they are computed individually rather than computing the full PTDF matrix and selecting the desired columns.

Examples:

```
H = makePTDF(mpc);
H = makePTDF(baseMVA, bus, branch, 1);
slack = rand(size(bus, 1), 1);
H = makePTDF(mpc, slack);

% for transfer from bus i to bus j
txfr = zeros(nb, 1); txfr(i) = 1; txfr(j) = -1;
H = makePTDF(mpc, slack, txfr);

% for buses i and j only
H = makePTDF(mpc, slack, [i;j]);
```

See also [makeLODF\(\)](#) (page 329).

makeSbus

makeSbus(baseMVA, bus, gen, mpopt, Vm, Sg)

[makeSbus\(\)](#) (page 330) - Builds the vector of complex bus power injections.

```
SBUS = MAKESBUS(BASEMVA, BUS, GEN)
SBUS = MAKESBUS(BASEMVA, BUS, GEN, MPOPT, VM)
SBUS = MAKESBUS(BASEMVA, BUS, GEN, MPOPT, VM, SG)
```

returns the vector of complex bus power injections, that is, generation minus load. Power is expressed in per unit. If the MPOPT and VM arguments are present it evaluates any ZIP loads based on the provided voltage magnitude vector. If VM is empty, it assumes nominal voltage. If SG is provided, it is a complex ng x 1 vector of generator power injections in p.u., and overrides the PG and QG columns in GEN, using GEN only for connectivity information.

```
[SBUS, DSBUS_DVM] = MAKESBUS(BASEMVA, BUS, GEN, MPOPT, VM)
```

With two output arguments, it computes the partial derivative of the bus injections with respect to voltage magnitude, leaving the first return value SBUS empty. If VM is empty, it assumes no voltage dependence and returns a sparse zero matrix.

See also [makeYbus\(\)](#) (page 331).

makeSdzip

makeSdzip(baseMVA, bus, mpopt)

[makeSdzip\(\)](#) (page 331) - Builds vectors of nominal complex bus power demands for ZIP loads.

SD = MAKESDZIP(BASEMVA, BUS, MPOPT) returns a **struct** with three fields, each an nb x 1 vectors. The fields 'z', 'i' and 'p' correspond to the nominal p.u. **complex power** (at 1 p.u. voltage magnitude) of the constant impedance, constant current, and constant **power** portions, respectively of the ZIP load model.

Example:

```
Sd = makeSdzip(baseMVA, bus, mpopt);
```

makeYbus

makeYbus(baseMVA, bus, branch)

[makeYbus\(\)](#) (page 331) - Builds the bus admittance matrix and branch admittance matrices.

```
[YBUS, YF, YT] = MAKEYBUS(MPC)
[YBUS, YF, YT] = MAKEYBUS(BASEMVA, BUS, BRANCH)
```

Returns the **full** bus admittance matrix (i.e. **for all** buses) and the matrices YF and YT which, when multiplied by a **complex** voltage vector, yield the vector currents injected into each **line** from the "from" and "to" buses respectively of each **line**. Does appropriate conversions to p.u. Inputs can be a MATPOWER **case struct** or individual BASEMVA, BUS and BRANCH values. Bus numbers must be consecutive beginning at 1 (i.e. internal ordering).

See also [makeJac\(\)](#) (page 328), [makeSbus\(\)](#) (page 330), ext2int().

5.2.12 Utility Functions

apply_changes

apply_changes(label, mpc, chgtab)

[apply_changes\(\)](#) (page 331) - Applies a set of changes to a MATPOWER case

```
mpc_modified = apply_changes(label, mpc_original, chgtab)
```

Applies the **set** of changes identified by LABEL to the **case** in MPC, where the change sets are specified in CHGTAB.

LABEL is an integer which identifies the **set** of changes of interest

MPC is a MATPOWER **case struct** with at least fields bus, gen and branch

(continues on next page)

(continued from previous page)

CHGTAB is the **table** of changes that lists the individual changes required by each change **set**, one "change" per row (multiple **rows** or changes allowed **for** each change **set**). Type "**help idx_ct**" **for** more complete information about the format.

1st column: change **set** label, integer > 0

2nd column: change **set** probability

3rd column: **table** to be modified (1:bus, 2:gen, 3:branch) **or** 4: bus **area** changes, apply to **all** gens/buses/branches in a given **area**; 5: gen **table area** changes apply to **all** generators in a given **area**; **or** 6: branch **area** changes apply to **all** branches connected to buses in a given **area**.

4th column: row of **table** to be modified (**if** 3rd col is 1-3), **or area** number **for** overall changes (**if** 3rd column is 4-6).

5th column: column of **table** to be modified. It is best to use the named column **index** defined in the corresponding **idx_bus**, **idx_gen**, **idx_branch** **or** **idx_cost** M-files in MATPOWER.

6th column: **type** of change: 1: absolute (replace)
2: relative (multiply by **factor**)
3: additive (add to value)

7th column: new value **or** multiplicative **or** additive **factor**

Examples:

```
chgtab = [ ...
    1  0.1  CT_TGEN      2  GEN_STATUS  CT_REP  0;
    2  0.05 CT_TGEN      3  PMAX         CT_REP 100;
    3  0.2  CT_TBRCH     2  BR_STATUS   CT_REP  0;
    4  0.1  CT_TAREALOAD 2  CT_LOAD_ALL_P CT_REL  1.1;
];
```

Description of each change **set**:

1. Turn off generator 2, 10% *probability*.
2. Set generator 3's **max** output to 100 MW, 5% *probability*.
3. Take branch 2 out of service, 20% *probability*.
4. Scale **all** loads in **area** 2 (**real** & reactive, fixed **and** dispatchable) by a **factor** of 1.1, 10% *probability*.

See also `idx_ct()` (page 344).

bustypes

bustypes(*bus*, *gen*)

bustypes() - Builds index lists for each type of bus (REF, PV, PQ).

[REF, PV, PQ] = **BUSTYPES**(BUS, GEN)

Generators with "out-of-service" status are treated as PQ buses with zero generation (regardless of Pg/Qg values in *gen*). Expects **BUS** **and** **GEN** have been converted to use internal consecutive bus numbering.

calc_branch_angle

calc_branch_angle(*mpc*)

[calc_branch_angle\(\)](#) (page 333) - Calculate branch angle differences across active branches

```
DELTA = CALC_BRANCH_ANGLE(MPC)
```

Calculates the **angle** difference (in degrees) across **all** active branches in the MATPOWER **case**. Angles are calculated as the difference between the FROM bus **and** the TO bus.

Input:

MPC - MATPOWER **case struct** (can have external bus numbering)

Output:

DELTA - $n_l \times 1$ vector of branch **angle** differences $A_f - A_t$, where A_f **and** A_t are vectors of voltage angles at "from" **and** "to" ends of each **line** respectively. DELTA is 0 **for** out-of-service branches.

See also [toggle_softlims\(\)](#) (page 310).

case_info

case_info(*mpc*, *fd*)

[case_info\(\)](#) (page 333) - Prints information about islands in a network.

```
CASE_INFO(MPC)
CASE_INFO(MPC, FD)
[GROUPS, ISOLATED] = CASE_INFO(...)
```

Prints out detailed information about a MATPOWER **case**. Optionally prints to an open file, whose file identifier, as returned by FOPEN, is specified in the optional second parameter FD. Optional **return** arguments include GROUPS **and** ISOLATED buses, as returned by FIND_ISLANDS.

compare_case

compare_case(*mpc1*, *mpc2*)

[compare_case\(\)](#) (page 333) - Compares the bus, gen, branch matrices of 2 MATPOWER cases.

```
COMPARE_CASE(MPC1, MPC2)
Compares the bus, branch and gen matrices of two MATPOWER cases and prints a summary of the differences. For each column of the matrix it prints the maximum of any non-zero differences.
```


define_constants

define_constants()

define_constants - Defines useful constants for indexing data, etc.

This is simply a convenience script that defines the constants listed below, consisting primarily of named indices **for** the **columns** of the data matrices: bus, branch, gen **and** gencost. This includes **input columns** defined in caseformat as well as **columns** that are added in the **power** flow **and** OPF output. It also defines constants **for** the change tables used by apply_changes().

bus:

PQ, PV, REF, NONE, BUS_I, BUS_TYPE, PD, QD, GS, BS, BUS_AREA, VM,
VA, BASE_KV, ZONE, VMAX, VMIN, LAM_P, LAM_Q, MU_VMAX, MU_VMIN

branch:

F_BUS, T_BUS, BR_R, BR_X, BR_B, RATE_A, RATE_B, RATE_C,
TAP, SHIFT, BR_STATUS, PF, QF, PT, QT, MU_SF, MU_ST,
ANGMIN, ANGMAX, MU_ANGMIN, MU_ANGMAX

gen:

GEN_BUS, PG, QG, QMAX, QMIN, VG, MBASE, GEN_STATUS, PMAX, PMIN,
MU_PMAX, MU_PMIN, MU_QMAX, MU_QMIN, PC1, PC2, QC1MIN, QC1MAX,
QC2MIN, QC2MAX, RAMP_AGC, RAMP_10, RAMP_30, RAMP_Q, APF

gencost:

PW_LINEAR, POLYNOMIAL, MODEL, STARTUP, SHUTDOWN, NCOST, COST

change tables:

CT_LABEL, CT_PROB, CT_TABLE, CT_TBUS, CT_TGEN, CT_TBRCH, CT_TAREABUS,
CT_TAREAGEN, CT_TAREABRCH, CT_ROW, CT_COL, CT_CHGTYPE, CT_REP,
CT_REL, CT_ADD, CT_NEWVAL, CT_TLOAD, CT_TAREALOAD, CT_LOAD_ALL_PQ,
CT_LOAD_FIX_PQ, CT_LOAD_DIS_PQ, CT_LOAD_ALL_P, CT_LOAD_FIX_P,
CT_LOAD_DIS_P, CT_TGENCOST, CT_TAREAGENCOST, CT_MODCOST_F,
CT_MODCOST_X

See CASEFORMAT, IDX_BUS, IDX_BRCH, IDX_GEN, IDX_COST **and** IDX_CT **for** details on the meaning of these constants. Internally DEFINE_CONSTANTS calls IDX_BUS, IDX_BRCH, IDX_GEN, IDX_COST **and** IDX_CT. In performance sensitive code, such as internal MATPOWER **functions** that are called frequently, it is preferred to call these **functions** directly rather than using the DEFINE_CONSTANTS script, which is less efficient.

This script is included **for** convenience **for** interactive use **or** **for** high-level code where maximum performance is **not** a concern.

extract_islands

extract_islands(*mpc*, *varargin*)

extract_islands() (page 335) - Extracts each island in a network with islands.

```

MPC_ARRAY = EXTRACT_ISLANDS(MPC)
MPC_ARRAY = EXTRACT_ISLANDS(MPC, GROUPS)
MPC_K = EXTRACT_ISLANDS(MPC, K)
MPC_K = EXTRACT_ISLANDS(MPC, GROUPS, K)
MPC_K = EXTRACT_ISLANDS(MPC, K, CUSTOM)
MPC_K = EXTRACT_ISLANDS(MPC, GROUPS, K, CUSTOM)

```

Returns a **cell** array of MATPOWER **case** structs **for** each island in the **input case struct**. If the optional second argument is a **cell** array GROUPS it is assumed to be a **cell** array of vectors of bus indices **for** each island (as returned by FIND_ISLANDS). Providing the GROUPS avoids the need **for** another traversal of the network connectivity **and** can save a significant amount of **time** on very large systems. If an additional argument K is included, it indicates which island(s) to **return** **and** the **return** value is a **single case struct**, rather than a **cell** array. If K is a scalar **or** vector, it specifies the **index**(indices) of the island(s) to include in the resulting **case** file. K can also be the string '**all**' which will include **all** islands. This is the same as simply eliminating **all** isolated buses.

A final optional argument CUSTOM is a **struct** that can be used to indicate custom fields of MPC from which to extract data corresponding to buses generators, branches **or** DC lines. It has the following structure:

```
CUSTOM.<ORDERING>{DIM} = FIELDS
```

<ORDERING> is either '**bus**', '**gen**', '**branch**' **or** '**dcline**' **and** indicates that dimension DIM of FIELDS has dimensions corresponding to this <ORDERING> **and** should have the appropriate dimension extracted as well. FIELDS is a **cell** array, where each element is either a **single** string (field name of MPC) **or** a **cell** array of strings (nested fields of MPC).

Examples:

Extract each island into it's own **case struct**:

```
mpc_list = extract_islands(mpc);
```

Extract the 2nd (that is, 2nd largest) island:

```
mpc2 = extract_islands(mpc, 2);
```

Extract the first **and** 3rd islands without a re-traverals of the network:

```
groups = find_islands(mpc);
mpc1 = extract_islands(mpc, groups, 1);
mpc3 = extract_islands(mpc, groups, 3);
```

(continues on next page)

(continued from previous page)

Extract the 2nd island, including custom fields, where `mpc.bus_label{b}` contains a label for bus `b`, and `mpc.gen_name{g}`, `mpc.emissions.rate(g, :)`, and `mpc.genloc(:, g)` contain, respectively, the generator's name, emission rates and location coordinates:

```
custom.bus{1} = {'bus_label'};
custom.gen{1} = {'gen_name', {'emissions', 'rate'}};
custom.gen{2} = {'genloc'};
mpc = extract_islands(mpc, 1, custom);
```

Note: Fields `bus_name`, `gentype` and `genfuel` are handled automatically and do not need to be included in `custom`.

See also [`find_islands\(\)`](#) (page 337), [`case_info\(\)`](#) (page 333), [`connected_components\(\)`](#) (page 357).

feval_w_path

feval_w_path(*fpath*, *fname*, *varargin*)

[`feval_w_path\(\)`](#) (page 336) - Calls a function located by the specified path.

```
FEVAL_W_PATH(FPATH, F, x1, ..., xn)
[y1, ..., yn] = FEVAL_W_PATH(FPATH, F, x1, ..., xn)
```

Identical to the built-in `FEVAL`, except that the **function** `F` need not be in the MATLAB/Octave **path** if it is defined in a file in the **path** specified by `FPATH`. Assumes that the current working directory is always first in the MATLAB/Octave **path**.

Inputs:

`FPATH` - string containing the **path** to the **function** to be called, can be absolute or relative to current working directory
`F` - string containing the name of the **function** to be called
`x1, ..., xn` - variable number of **input** arguments to be passed to `F`

Output:

`y1, ..., yn` - variable number arguments returned by `F` (depending on the caller)

Note that **any** sub-**functions** located in the directory specified by `FPATH` will also be available to be called by the `F` **function**.

Examples:

```
% Assume '/opt/testfunctions' is NOT in the MATLAB/Octave path, but
% /opt/testfunctions/mytestfcn.m defines the function mytestfcn()
% which takes 2 input arguments and outputs 1 return argument.
y = feval_w_path('/opt/testfunctions', 'mytestfcn', x1, x2);
```

find_bridges

find_bridges(*mpc*)

find_bridges() (page 337) - Finds bridges in a network.

```
[ISLANDS, BRIDGES, NONBRIDGES] = FIND_BRIDGES(MPC)
```

Returns the islands, bridges **and** non-bridges in a network.

Bridges are filtered out using Tarjan's algorithm. A BRIDGE is a branch whose removal breaks the island to multiple parts. The **return** value BRIDGES is a **cell** array of vectors of the bus indices **for** each island.

find_islands

find_islands(*mpc*)

find_islands() (page 337) - Finds islands in a network.

```
GROUPS = FIND_ISLANDS(MPC)
[GROUPS, ISOLATED] = FIND_ISLANDS(MPC)
```

Returns the islands in a network. The **return** value GROUPS is a **cell** array of vectors of the bus indices **for** each island. The second **and** optional **return** value ISOLATED is a vector of indices of isolated buses that have no connecting branches.

See also *extract_islands()* (page 335), *connected_components()* (page 357).

genfuels

genfuels()

genfuels() - Return list of standard values for generator fuel types.

```
GF = GENFUELS()
```

Returns a **cell** array of strings containing the following standard generator fuel types **for** use in the optional MPC.GENFUEL field of the MATPOWER **case struct**. This is to be considered an unordered list, where the position of a particular fuel **type** in the list is **not** defined **and** is therefore subject to change.

biomass	- Biomass
coal	- Coal
dfo	- Distillate Fuel Oil (Diesel, F01, F02, F04)
geothermal	- Geothermal
hydro	- Hydro
hydrops	- Hydro Pumped Storage
jetfuel	- Jet Fuel
lng	- Liquefied Natural Gas

(continues on next page)

(continued from previous page)

ng	- Natural Gas
nuclear	- Nuclear
oil	- Unspecified Oil
refuse	- Refuse, Municipal Solid Waste
rfo	- Residual Fuel Oil (F05, F06)
solar	- Solar
syncgen	- Synchronous Condensor
wasteheat	- Waste Heat
wind	- Wind
wood	- Wood or Wood Waste
other	- Other
unknown	- Unknown
dl	- Dispatchable Load
ess	- Energy Storage System

Example:

```

if ~ismember(mpc.genfuel{k}, genfuels())
    error('unknown fuel type');
end

```

See also `gentypes()`, `savecase()`.

gentypes

gentypes()

`gentypes()` - Return list of standard values for generator unit types.

```
GT = GENTYPES()
```

Returns a `cell` array of strings containing the following standard two character generator unit types `for` use in the optional `MPC.GENTYPE` field of the MATPOWER `case struct`. This is to be considered an unordered list, where the position of a particular fuel `type` in the list is `not` defined `and` is therefore subject to change.

From Form EIA-860 Instructions, Table 2. Prime Mover Codes `and` Descriptions
https://www.eia.gov/survey/form/eia_860/instructions.pdf

BA	- Energy Storage, Battery
CE	- Energy Storage, Compressed Air
CP	- Energy Storage, Concentrated Solar Power
FW	- Energy Storage, Flywheel
PS	- Hydraulic Turbine, Reversible (pumped storage)
ES	- Energy Storage, Other
ST	- Steam Turbine, including nuclear, geothermal <code>and</code> solar steam (does <code>not</code> include combined cycle)
GT	- Combustion (Gas) Turbine (includes <code>jet</code> engine design)
IC	- Internal Combustion Engine (diesel, piston, reciprocating)
CA	- Combined Cycle Steam Part
CT	- Combined Cycle Combustion Turbine Part (<code>type</code> of coal <code>or</code> solid must be reported as energy <code>source</code>)

(continues on next page)

(continued from previous page)

```

    for integrated coal gasification)
CS - Combined Cycle Single Shaft
    (combustion turbine and steam turbine share a single generator)
CC - Combined Cycle Total Unit
    (use only for plants/generators that are in planning stage,
    for which specific generator details cannot be provided)
HA - Hydrokinetic, Axial Flow Turbine
HB - Hydrokinetic, Wave Buoy
HK - Hydrokinetic, Other
HY - Hydroelectric Turbine (includes turbines associated with
    delivery of water by pipeline)
BT - Turbines Used in a Binary Cycle
    (including those used for geothermal applications)
PV - Photovoltaic
WT - Wind Turbine, Onshore
WS - Wind Turbine, Offshore
FC - Fuel Cell
OT - Other
Additional codes (some from PowerWorld)
UN - Unknown
JE - Jet Engine
NB - ST - Boiling Water Nuclear Reactor
NG - ST - Graphite Nuclear Reactor
NH - ST - High Temperature Gas Nuclear Reactor
NP - ST - Pressurized Water Nuclear Reactor
IT - Internal Combustion Turbo Charged
SC - Synchronous Condenser
DC - represents DC ties
MP - Motor/Pump
W1 - Wind Turbine, Type 1
W2 - Wind Turbine, Type 2
W3 - Wind Turbine, Type 3
W4 - Wind Turbine, Type 4
SV - Static Var Compensator
DL - Dispatchable Load

```

Example:

```

if ~ismember(mpc.gentype{k}, gentypes())
    error('unknown generator unit type');
end

```

See also `genfuels()`, `savecase()`.

get_losses**get_losses**(baseMVA, bus, branch)*get_losses()* (page 340) - Returns series losses (and reactive injections) per branch.

```

LOSS = GET_LOSSES(RESULTS)
LOSS = GET_LOSSES(BASEMVA, BUS, BRANCH)

[LOSS, CHG] = GET_LOSSES(RESULTS)
[LOSS, FCHG, TCHG] = GET_LOSSES(RESULTS)
[LOSS, FCHG, TCHG, DLOSS_DV] = GET_LOSSES(RESULTS)
[LOSS, FCHG, TCHG, DLOSS_DV, DCHG_DVM] = GET_LOSSES(RESULTS)

```

Computes branch series losses, and optionally reactive injections from line charging, as functions of bus voltages and branch parameters, using the following formulae:

$$\begin{aligned} \text{loss} &= \text{abs}(V_f / \tau - V_t)^2 / (R_s - j X_s) \\ \text{fchg} &= \text{abs}(V_f / \tau)^2 * B_c / 2 \\ \text{tchg} &= \text{abs}(V_t)^2 * B_c / 2 \end{aligned}$$

Optionally, computes the partial derivatives of the line losses with respect to voltage angles and magnitudes.

Input:

RESULTS - a MATPOWER **case struct** with bus voltages corresponding to a valid power flow solution.
(Can optionally be specified as individual fields BASEMVA, BUS, and BRANCH.)

Output(s):

LOSS - **complex** NL x 1 vector of losses (in MW), where NL is the number of branches in the system, representing only the losses in the series impedance element of the PI model for each branch.

CHG - NL x 1 vector of total reactive injection for each line (in MVAR), representing the line charging injections of both of the shunt elements of PI model for each branch.

FCHG - Same as CHG, but for the element at the "from" end of the branch only.

TCHG - Same as CHG, but for the element at the "to" end of the branch.

DLOSS_DV - Struct with partial derivatives of LOSS with respect to bus voltages, with fields:

- .a - Partial with respect to bus voltage angles.
- .m - Partial with respect to bus voltage magnitudes.

DCHG_DVM - Struct with partial derivatives of FCHG and TCHG with respect to bus voltage magnitudes, with fields:

- .f - Partial of FCHG with respect to bus voltage magnitudes.
- .t - Partial of TCHG with respect to bus voltage magnitudes.

Example:

```

results = runpf(myCase);
[loss, chg] = get_losses(results);
total_system_real_losses = sum(real(loss));

```

(continues on next page)

(continued from previous page)

```
total_system_reac_losses = sum(imag(loss)) - sum(chg);

[loss, fchg, tchg, dloss_dV] = get_losses(results);
```

hasPQcap

hasPQcap(*gen, hilo*)

hasPQcap() (page 341) - Checks for P-Q capability curve constraints.

TORF = HASPQCAP(GEN, HILO) returns a column vector of 1's and 0's. The 1's correspond to rows of the GEN matrix which correspond to generators which have defined a capability curve (with sloped upper and/or lower bound on Q) and require that additional linear constraints be added to the OPF.

The GEN matrix in version 2 of the MATPOWER case format includes columns for specifying a P-Q capability curve for a generator defined as the intersection of two half-planes and the box constraints on P and Q. The two half planes are defined respectively as the area below the line connecting (Pc1, Qc1max) and (Pc2, Qc2max) and the area above the line connecting (Pc1, Qc1min) and (Pc2, Qc2min).

If the optional 2nd argument is 'U' this function returns true only for rows corresponding to generators that require the upper constraint on Q. If it is 'L', only for those requiring the lower constraint. If the 2nd argument is not specified or has any other value it returns true for rows corresponding to gens that require either or both of the constraints.

It is smart enough to return true only if the corresponding linear constraint is not redundant w.r.t the box constraints.

idx_brch

idx_brch()

idx_brch() (page 341) - Defines constants for named column indices to branch matrix.

Example:

```
[F_BUS, T_BUS, BR_R, BR_X, BR_B, RATE_A, RATE_B, RATE_C, ...
TAP, SHIFT, BR_STATUS, PF, QF, PT, QT, MU_SF, MU_ST, ...
ANGMIN, ANGMAX, MU_ANGMIN, MU_ANGMAX] = idx_brch;
```

Some examples of usage, after defining the constants using the line above, are:

```
branch(4, BR_STATUS) = 0; % take branch 4 out of service
Ploss = branch(:, PF) + branch(:, PT); % compute real power loss vector
```

(continues on next page)

(continued from previous page)

The `index`, name and meaning of each column of the branch matrix is given below:

columns 1-11 must be included in `input` matrix (in `case` file)

1	F_BUS	f, from bus number
2	T_BUS	t, to bus number
3	BR_R	r, resistance (p.u.)
4	BR_X	x, reactance (p.u.)
5	BR_B	b, total line charging susceptance (p.u.)
6	RATE_A	rateA, MVA rating A (long term rating)
7	RATE_B	rateB, MVA rating B (short term rating)
8	RATE_C	rateC, MVA rating C (emergency rating)
9	TAP	ratio, transformer off nominal turns ratio
10	SHIFT	angle, transformer phase shift angle (degrees)
11	BR_STATUS	initial branch status, 1 - in service, 0 - out of service
12	ANGMIN	minimum angle difference, $\text{angle}(V_f) - \text{angle}(V_t)$ (degrees)
13	ANGMAX	maximum angle difference, $\text{angle}(V_f) - \text{angle}(V_t)$ (degrees)

(The voltage angle difference is taken to be unbounded below if $\text{ANGMIN} < -360$ and unbounded above if $\text{ANGMAX} > 360$. If both parameters are zero, it is unconstrained.)

columns 14-17 are added to matrix after power flow or OPF solution
they are typically not present in the `input` matrix

14	PF	real power injected into "from" end of branch (MW)
15	QF	reactive power injected into "from" end of branch (MVar)
16	PT	real power injected into "to" end of branch (MW)
17	QT	reactive power injected into "to" end of branch (MVar)

columns 18-21 are added to matrix after OPF solution

they are typically not present in the `input` matrix

(assume OPF objective function has units, u)

18	MU_SF	Kuhn-Tucker multiplier on MVA limit at "from" bus (u/MVA)
19	MU_ST	Kuhn-Tucker multiplier on MVA limit at "to" bus (u/MVA)
20	MU_ANGMIN	Kuhn-Tucker multiplier lower angle difference limit (u/degree)
21	MU_ANGMAX	Kuhn-Tucker multiplier upper angle difference limit (u/degree)

See also `define_constants`.

idx_bus

idx_bus()

`idx_bus()` (page 342) - Defines constants for named column indices to bus matrix.

Example:

```
[PQ, PV, REF, NONE, BUS_I, BUS_TYPE, PD, QD, GS, BS, BUS_AREA, VM, ...
VA, BASE_KV, ZONE, VMAX, VMIN, LAM_P, LAM_Q, MU_VMAX, MU_VMIN] = idx_bus;
```

Some examples of `usage`, after defining the constants using the `line` above, are:

(continues on next page)

(continued from previous page)

```
Pd = bus(4, PD);      % get the real power demand at bus 4
bus(:, VMIN) = 0.95;  % set the min voltage magnitude to 0.95 at all buses
```

The **index**, **name** and **meaning** of each column of the bus matrix is given below:

columns 1-13 must be included in **input** matrix (in **case** file)

1	BUS_I	bus number (positive integer)
2	BUS_TYPE	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
3	PD	Pd, real power demand (MW)
4	QD	Qd, reactive power demand (MVar)
5	GS	Gs, shunt conductance (MW demanded at V = 1.0 p.u.)
6	BS	Bs, shunt susceptance (MVar injected at V = 1.0 p.u.)
7	BUS_AREA	area number, (positive integer)
8	VM	Vm, voltage magnitude (p.u.)
9	VA	Va, voltage angle (degrees)
10	BASE_KV	baseKV, base voltage (kV)
11	ZONE	zone, loss zone (positive integer)
12	VMAX	maxVm, maximum voltage magnitude (p.u.)
13	VMIN	minVm, minimum voltage magnitude (p.u.)

columns 14-17 are added to matrix after OPF solution

they are typically **not** present in the **input** matrix

		(assume OPF objective function has units, u)
14	LAM_P	Lagrange multiplier on real power mismatch (u/MW)
15	LAM_Q	Lagrange multiplier on reactive power mismatch (u/MVar)
16	MU_VMAX	Kuhn-Tucker multiplier on upper voltage limit (u/p.u.)
17	MU_VMIN	Kuhn-Tucker multiplier on lower voltage limit (u/p.u.)

additional constants, used to assign/compare values in the BUS_TYPE column

1	PQ	PQ bus
2	PV	PV bus
3	REF	reference bus
4	NONE	isolated bus

See also `define_constants`.

idx_cost

idx_cost()

`idx_cost()` (page 343) - Defines constants for named column indices to `gencost` matrix.

Example:

```
[PW_LINEAR, POLYNOMIAL, MODEL, STARTUP, SHUTDOWN, NCOST, COST] = idx_cost;
```

Some examples of **usage**, after defining the constants using the **line** above, are:

(continues on next page)

(continued from previous page)

```

start = gencost(4, STARTUP);           % get startup cost of generator 4
gencost(2, [MODEL, NCOST:COST+1]) = [ POLYNOMIAL 2 30 0 ];
% set the cost of generator 2 to a linear function COST = 30 * Pg

```

The **index**, name and meaning of each column of the gencost matrix is given below:

columns 1-5

1	MODEL	cost model, 1 = piecewise linear, 2 = polynomial
2	STARTUP	startup cost in US dollars
3	SHUTDOWN	shutdown cost in US dollars
4	NCOST	number $N = n+1$ of data points to follow defining an n -segment piecewise linear cost function , or of cost coefficients defining an n -th order polynomial cost function
5	COST	parameters defining total cost function $f(p)$ begin in this column (MODEL = 1) : $p_1, f_1, p_2, f_2, \dots, p_N, f_N$ where $p_1 < p_2 < \dots < p_N$ and the cost $f(p)$ is defined by the coordinates $(p_1, f_1), (p_2, f_2), \dots, (p_N, f_N)$ of the end/break -points of the piecewise linear cost fcn (MODEL = 2) : c_n, \dots, c_1, c_0 N coefficients of an n -th order polynomial cost function , starting with highest order, where cost is $f(p) = c_n p^n + \dots + c_1 p + c_0$

additional constants, used to assign/compare values in the MODEL column

1	PW_LINEAR	piecewise linear generator cost model
2	POLYNOMIAL	polynomial generator cost model

See also `define_constants`.

idx_ct

idx_ct()

`idx_ct()` (page 344) - Defines constants for named column indices to changes table

```

[CT_LABEL, CT_PROB, CT_TABLE, CT_TBUS, CT_TGEN, CT_TBRCH, CT_TAREABUS, ...
CT_TAREAGEN, CT_TAREABRCH, CT_ROW, CT_COL, CT_CHGTYPE, CT_REP, ...
CT_REL, CT_ADD, CT_NEWVAL, CT_TLOAD, CT_TAREALOAD, CT_LOAD_ALL_PQ, ...
CT_LOAD_FIX_PQ, CT_LOAD_DIS_PQ, CT_LOAD_ALL_P, CT_LOAD_FIX_P, ...
CT_LOAD_DIS_P, CT_TGENCOST, CT_TAREAGENCOST, CT_MODCOST_F, ...
CT_MODCOST_X] = idx_ct;

```

CT_LABEL: column of changes **table** where the change **set** label is stored

CT_PROB: column of changes **table** where the probability of the change **set** is stored

CT_TABLE: column of the changes **table** where the **type** of **system** data **table** to be modified is stored;

(continues on next page)

(continued from previous page)

```

type CT_TBUS indicates bus table
type CT_TGEN indicates gen table
type CT_TBRCH indicates branch table
type CT_TLOAD indicates a load modification (bus and/or gen tables)
type CT_TAREABUS indicates area-wide change in bus table
type CT_TAREAGEN indicates area-wide change in generator table
type CT_TAREABRCH indicates area-wide change in branch table
type CT_TAREALOAD indicates area-wide change in load
                        (bus and/or gen tables)

```

CT_ROW: column of changes table where the row number in the data table to be modified is stored. A value of "0" in this column has the special meaning "apply to all rows". For an area-wide type of change, the area number is stored here instead.

CT_COL: column of changes table where the number of the column in the data table to be modified is stored
 For CT_TLOAD and CT_TAREALOAD, the value entered in this column is one of the following codes (or its negative), rather than a column index:

```

type CT_LOAD_ALL_PQ modify all loads, real & reactive
type CT_LOAD_FIX_PQ  modify only fixed loads, real & reactive
type CT_LOAD_DIS_PQ  modify only dispatchable loads, real & reactive
type CT_LOAD_ALL_P   modify all loads, real only
type CT_LOAD_FIX_P   modify only fixed loads, real only
type CT_LOAD_DIS_P   modify only dispatchable loads, real only

```

If the negative of one of these codes is used, then any affected dispatchable loads will have their costs scaled as well.

For CT_TGENCOST and CT_TAREAGENCOST, in addition to an actual column index, this value can also take one of the following codes to indicate a scaling (CT_REL change type) or shifting (CT_ADD change type) of the specified cost functions:

```

type CT_MODCOST_F scales or shifts the cost function vertically
type CT_MODCOST_X scales or shifts the cost function horizontally

```

See MODCOST.

CT_CHGTYPE: column of changes table where the type of change to be made is stored:

```

type CT_REP replaces old value by value in CT_NEWVAL column
type CT_REL multiplies old value by factor in CT_NEWVAL column
type CT_ADD adds value in CT_NEWVAL column to old value

```

See also [apply_changes\(\)](#) (page 331), [modcost\(\)](#).

idx_dcline

idx_dcline()

idx_dcline() (page 346) - Defines constants for named column indices to dcline matrix.

Example:

```
c = idx_dcline;
```

Some examples of *usage*, after defining the constants using the *line* above, are:

```
mpc.dcline(4, c.BR_STATUS) = 0;           % take dcline 4 out of service
```

The *index*, name and meaning of each column of the dcline matrix is given below:

columns 1-17 must be included in *input* matrix (in *case* file)

1	F_BUS	f, "from" bus number
2	T_BUS	t, "to" bus number
3	BR_STATUS	initial dcline status, 1 - in service, 0 - out of service
4	PF	MW flow at "from" bus ("from" -> "to")
5	PT	MW flow at "to" bus ("from" -> "to")
6	QF	MVar injection at "from" bus ("from" -> "to")
7	QT	MVar injection at "to" bus ("from" -> "to")
8	VF	voltage setpoint at "from" bus (p.u.)
9	VT	voltage setpoint at "to" bus (p.u.)
10	PMIN	lower limit on PF (MW flow at "from" end)
11	PMAX	upper limit on PF (MW flow at "from" end)
12	QMINF	lower limit on MVar injection at "from" bus
13	QMAXF	upper limit on MVar injection at "from" bus
14	QMINT	lower limit on MVar injection at "to" bus
15	QMAXT	upper limit on MVar injection at "to" bus
16	LOSS0	constant term of linear loss <i>function</i> (MW)
17	LOSS1	linear term of linear loss <i>function</i> (MW/MW) (loss = LOSS0 + LOSS1 * PF)

columns 18-23 are added to matrix after OPF solution

they are typically *not* present in the *input* matrix

(assume OPF objective *function* has units, u)

18	MU_PMIN	Kuhn-Tucker multiplier on lower flow lim at "from" bus (u/MW)
19	MU_PMAX	Kuhn-Tucker multiplier on upper flow lim at "from" bus (u/MW)
20	MU_QMINF	Kuhn-Tucker multiplier on lower VAR lim at "from" bus (u/MVar)
21	MU_QMAXF	Kuhn-Tucker multiplier on upper VAR lim at "from" bus (u/MVar)
22	MU_QMINT	Kuhn-Tucker multiplier on lower VAR lim at "to" bus (u/MVar)
23	MU_QMAXT	Kuhn-Tucker multiplier on upper VAR lim at "to" bus (u/MVar)

See also *toggle_dcline()* (page 307).

idx_gen

idx_gen()

idx_gen() (page 347) - Defines constants for named column indices to gen matrix.

Example:

```
[GEN_BUS, PG, QG, QMAX, QMIN, VG, MBASE, GEN_STATUS, PMAX, PMIN, ...
MU_PMAX, MU_PMIN, MU_QMAX, MU_QMIN, PC1, PC2, QC1MIN, QC1MAX, ...
QC2MIN, QC2MAX, RAMP_AGC, RAMP_10, RAMP_30, RAMP_Q, APF] = idx_gen;
```

Some examples of **usage**, after defining the constants using the **line** above, are:

```
Pg = gen(4, PG); % get the real power output of generator 4
gen(:, PMIN) = 0; % set to zero the minimum real power limit of all gens
```

The **index**, name **and** meaning of each column of the gen matrix is given below:

columns 1-21 must be included in **input** matrix (in **case** file)

1	GEN_BUS	bus number
2	PG	Pg, real power output (MW)
3	QG	Qg, reactive power output (MVar)
4	QMAX	Qmax, maximum reactive power output (MVar)
5	QMIN	Qmin, minimum reactive power output (MVar)
6	VG	Vg, voltage magnitude setpoint (p.u.)
7	MBASE	mBase, total MVA base of machine, defaults to baseMVA
8	GEN_STATUS	status, > 0 - in service, <= 0 - out of service
9	PMAX	Pmax, maximum real power output (MW)
10	PMIN	Pmin, minimum real power output (MW)
11	PC1	Pc1, lower real power output of PQ capability curve (MW)
12	PC2	Pc2, upper real power output of PQ capability curve (MW)
13	QC1MIN	Qc1min, minimum reactive power output at Pc1 (MVar)
14	QC1MAX	Qc1max, maximum reactive power output at Pc1 (MVar)
15	QC2MIN	Qc2min, minimum reactive power output at Pc2 (MVar)
16	QC2MAX	Qc2max, maximum reactive power output at Pc2 (MVar)
17	RAMP_AGC	ramp rate for load following/AGC (MW/min)
18	RAMP_10	ramp rate for 10 minute reserves (MW)
19	RAMP_30	ramp rate for 30 minute reserves (MW)
20	RAMP_Q	ramp rate for reactive power (2 sec timescale) (MVar/min)
21	APF	area participation factor

columns 22-25 are added to matrix after OPF solution

they are typically **not** present in the **input** matrix

(assume OPF objective **function** has units, u)

22	MU_PMAX	Kuhn-Tucker multiplier on upper Pg limit (u/MW)
23	MU_PMIN	Kuhn-Tucker multiplier on lower Pg limit (u/MW)
24	MU_QMAX	Kuhn-Tucker multiplier on upper Qg limit (u/MVar)
25	MU_QMIN	Kuhn-Tucker multiplier on lower Qg limit (u/MVar)

See also `define_constants`.

isload

isload(*gen*)

isload() - Checks for dispatchable loads.

TORF = ISLOAD(GEN) returns a column vector of 1's and 0's. The 1's correspond to rows of the GEN matrix which represent dispatchable loads. The current test is $P_{min} < 0$ AND $P_{max} == 0$. This may need to be revised to allow sensible specification of both elastic demand and pumped storage units.

load2disp

load2disp(*mpc0, fname, idx, voll*)

load2disp() - Converts fixed loads to dispatchable.

```
MPC = LOAD2DISP(MPC0);  
MPC = LOAD2DISP(MPC0, FNAME);  
MPC = LOAD2DISP(MPC0, FNAME, IDX);  
MPC = LOAD2DISP(MPC0, FNAME, IDX, VOLL);
```

Takes a MATPOWER case file or struct and converts fixed loads to dispatchable loads and returns the resulting case struct. Inputs are as follows:

MPC0 - File name or struct with initial MATPOWER case.

FNAME (optional) - Name to use to save resulting MATPOWER case. If empty, the case will not be saved to a file.

IDX (optional) - Vector of bus indices of loads to be converted. If empty or not supplied, it will convert all loads with positive real power demand.

VOLL (optional) - Scalar or vector specifying the value of lost load to use as the value for the dispatchable loads. If it is a scalar it is used for all loads, if a vector, the dimension must match that of IDX. Default is \$5000 per MWh.

loadshed

loadshed(*gen*, *ild*)

loadshed() - Returns a vector of curtailments of dispatchable loads.

```
SHED = LOADSHED(GEN)
SHED = LOADSHED(GEN, ILD)
```

Returns a column vector of MW curtailments of dispatchable loads.

Inputs:

GEN - MATPOWER generator matrix
ILD - (optional) NLD x 1 vector of generator indices corresponding to the dispatchable loads of interest, default is **all** dispatchable loads as determined by the ISLOAD() **function**.

Output:

SHED - NLD x 1 vector of the MW curtailment **for** each dispatchable **load** of interest

Example:

```
total_load_shed = max(loadshed(mpc.gen));
```

modcost

modcost(*gencost*, *alpha*, *modtype*)

modcost() - Modifies generator costs by shifting or scaling (F or X).

```
NEWGENCOST = MODCOST(GENCOST, ALPHA)
NEWGENCOST = MODCOST(GENCOST, ALPHA, MODTYPE)
```

For each generator cost $F(X)$ (**for real or reactive power**) in GENCOST, this **function** modifies the cost by scaling **or** shifting the **function** by ALPHA, depending on the value of MODTYPE, **and** **and** returns the modified GENCOST. Rows of GENCOST can be a mix of polynomial **or** piecewise linear costs. ALPHA can be a scalar, applied to each row of GENCOST, **or** an NG x 1 vector, where each element is applied to the corresponding row of GENCOST.

MODTYPE takes one of the 4 possible values (let $F_{\alpha}(X)$ denote the modified **function**):

```
'SCALE_F' (default) :  $F_{\alpha}(X) == F(X) * ALPHA$ 
'SCALE_X'           :  $F_{\alpha}(X * ALPHA) == F(X)$ 
'SHIFT_F'           :  $F_{\alpha}(X) == F(X) + ALPHA$ 
'SHIFT_X'           :  $F_{\alpha}(X + ALPHA) == F(X)$ 
```


mpver

mpver(*varargin*)

mpver() - Prints or returns installed MATPOWER version info.

```
mpver
v = mpver
v = mpver('all')
```

When called with an output argument and no input argument, **mpver**() returns the current MATPOWER version numbers. With an input argument (e.g. 'all') it returns a struct with the fields Name, Version, Release, and Date (*all char arrays*). Calling **mpver**() without assigning the return value prints the version and release date of the current installation of MATPOWER, MATLAB (or MATLAB), the Optimization Toolbox, MP-Test, MIPS, MP-Opt-Model, MOST, and any optional MATPOWER packages.

poly2pwl

poly2pwl(*polycost*, *Pmin*, *Pmax*, *npts*)

poly2pwl() - Converts polynomial cost variable to piecewise linear.

```
PWLCOST = POLY2PWL(POLYCOST, PMIN, PMAX, NPTS) converts the polynomial
cost variable POLYCOST into a piece-wise linear cost by evaluating at
NPTS evenly spaced points between PMIN and PMAX. If the range does not
include 0, then it is evaluated at 0 and NPTS-1 evenly spaced points
between PMIN and PMAX.
```

polycost

polycost(*gencost*, *Pg*, *der*)

polycost() - Evaluates polynomial generator cost & derivatives.

```
F = POLYCOST(GENCOST, PG) returns the vector of costs evaluated at PG

DF = POLYCOST(GENCOST, PG, 1) returns the vector of first derivatives
of costs evaluated at PG

D2F = POLYCOST(GENCOST, PG, 2) returns the vector of second derivatives
of costs evaluated at PG

GENCOST must contain only polynomial costs
PG is in MW, not p.u. (works for QG too)

This is a more efficient implementation that what can be done with
MATLAB's built-in POLYVAL and POLYDER functions.
```

pqcost

pqcost(gencost, ng, on)

pqcost() - Splits the gencost variable into two pieces if costs are given for Qg.

[PCOST, QCOST] = PQCOST(GENCOST, NG, ON) checks whether GENCOST has cost information **for** reactive **power** generation (**rows** ng+1 to 2*ng). If so, it returns the first NG **rows** in PCOST **and** the last NG **rows** in QCOST. Otherwise, leaves QCOST empty. Also does some **error** checking. If ON is specified (list of indices of generators which are on **line**) it only returns the **rows** corresponding to these generators.

scale_load

scale_load(dmd, bus, gen, load_zone, opt, gencost)

scale_load() (page 351) - Scales fixed and/or dispatchable loads.

```
MPC = SCALE_LOAD(LOAD, MPC);
MPC = SCALE_LOAD(LOAD, MPC, LOAD_ZONE)
MPC = SCALE_LOAD(LOAD, MPC, LOAD_ZONE, OPT)
BUS = SCALE_LOAD(LOAD, BUS);
[BUS, GEN] = SCALE_LOAD(LOAD, BUS, GEN, LOAD_ZONE, OPT)
[BUS, GEN, GENCOST] = ...
    SCALE_LOAD(LOAD, BUS, GEN, LOAD_ZONE, OPT, GENCOST)
```

Scales active (**and** optionally reactive) loads in each zone by a zone-specific ratio, i.e. $R(k)$ **for** zone k . Inputs are ...

LOAD - Each element specifies the amount of scaling **for** the corresponding **load** zone, either as a direct scale **factor** **or** as a target quantity. If there are n_z **load** zones this vector has n_z elements.

MPC - standard MATPOWER **case struct** **or case** file name

BUS - standard BUS matrix with n_b **rows**, where the fixed active **and** reactive loads available **for** scaling are specified in **columns** PD **and** QD

GEN - (optional) standard GEN matrix with n_g **rows**, where the dispatchable loads available **for** scaling are specified by **columns** PG, QG, PMIN, QMIN **and** QMAX (in **rows** **for** which ISLOAD(GEN) returns **true**). If GEN is empty, it assumes there are no dispatchable loads.

LOAD_ZONE - (optional) n_b element vector where the value of each element is either zero **or** the **index** of the **load** zone to which the corresponding bus belongs. If $LOAD_ZONE(b) = k$ then the loads at bus b will be scaled according to the value of $LOAD(k)$. If $LOAD_ZONE(b) = 0$, the loads at bus b

(continues on next page)

(continued from previous page)

will **not** be modified. If `LOAD_ZONE` is empty, the default is determined by the dimensions of the `LOAD` vector. If `LOAD` is a scalar, a **single system**-wide zone including **all** buses is used, i.e. `LOAD_ZONE = ONES(nb, 1)`. If `LOAD` is a vector, the default `LOAD_ZONE` is defined as the areas specified in the `BUS` matrix, i.e. `LOAD_ZONE = BUS(:, BUS_AREA)`, and `LOAD` should have dimension = `MAX(BUS(:, BUS_AREA))`.

`OPT` - (optional) **struct** with three possible fields, `'scale'`, `'pq'` and `'which'` that determine the behavior as follows:

`OPT.scale` (default is `'FACTOR'`)

`'FACTOR'` : `LOAD` consists of direct scale factors, where
 $\text{LOAD}(k) = \text{scale factor } R(k) \text{ for zone } k$
`'QUANTITY'` : `LOAD` consists of target quantities, where
 $\text{LOAD}(k) = \text{desired total active load in MW for zone } k \text{ after scaling by an appropriate } R(k)$

`OPT.pq` (default is `'PQ'`)

`'PQ'` : scale both active and reactive loads
`'P'` : scale only active loads

`OPT.which` (default is `'BOTH'` if `GEN` is provided, else `'FIXED'`)

`'FIXED'` : scale only fixed loads
`'DISPATCHABLE'` : scale only dispatchable loads
`'BOTH'` : scale both fixed and dispatchable loads

`OPT.cost` : (default = -1) flag to include cost in scaling or not

-1 : include cost if `gencost` is available
0 : do not include cost
1 : include cost (error if `gencost` not available)

`GENCOST` - (optional) standard `GENCOST` matrix with `ng` (or `2*ng`) rows, where the dispatchable load rows are determined by the `GEN` matrix. If included, the quantity axis of the marginal "cost" or benefit function of any dispatchable loads will be scaled with the size of the load itself (using `MODCOST` twice, once with `MODTYPE` equal to `SCALE_F` and once with `SCALE_X`).

Examples:

Scale all real and reactive fixed loads up by 10%.

```
bus = scale_load(1.1, bus);
```

Scale all active loads (fixed and dispatchable) at the first 10 buses so their total equals 100 MW, and at next 10 buses so their total equals 50 MW.

```
load_zone = zeros(nb, 1);
load_zone(1:10) = 1;
load_zone(11:20) = 2;
opt = struct('pq', 'P', 'scale', 'QUANTITY');
```

(continues on next page)

(continued from previous page)

```
dmd = [100; 50];
[bus, gen] = scale_load(dmd, bus, gen, load_zone, opt);
```

See also [total_load\(\)](#) (page 353).

total_load

total_load(bus, gen, load_zone, opt, mpopt)

[total_load\(\)](#) (page 353) - Returns vector of total load in each load zone.

```
PD = TOTAL_LOAD(MPC)
PD = TOTAL_LOAD(MPC, LOAD_ZONE)
PD = TOTAL_LOAD(MPC, LOAD_ZONE, OPT)
PD = TOTAL_LOAD(MPC, LOAD_ZONE, OPT, MPOPT)
PD = TOTAL_LOAD(BUS)
PD = TOTAL_LOAD(BUS, GEN)
PD = TOTAL_LOAD(BUS, GEN, LOAD_ZONE)
PD = TOTAL_LOAD(BUS, GEN, LOAD_ZONE, OPT)
PD = TOTAL_LOAD(BUS, GEN, LOAD_ZONE, OPT, MPOPT)
[PD, QD] = TOTAL_LOAD(...) returns both active and reactive power
demand for each zone.
```

MPC - standard MATPOWER **case struct**

BUS - standard BUS matrix with nb **rows**, where the fixed active and reactive loads are specified in **columns** PD and QD

GEN - (optional) standard GEN matrix with ng **rows**, where the dispatchable loads are specified by **columns** PG, QG, PMIN, QMIN and QMAX (in **rows** for which ISLOAD(GEN) returns **true**). If GEN is empty, it assumes there are no dispatchable loads.

LOAD_ZONE - (optional) nb element vector where the value of each element is either zero or the **index** of the **load zone** to which the corresponding bus belongs. If LOAD_ZONE(b) = k then the loads at bus b will added to the values of PD(k) and QD(k). If LOAD_ZONE is empty, the default is defined as the areas specified in the BUS matrix, i.e. LOAD_ZONE = BUS(:, BUS_AREA) and load will have dimension = MAX(BUS(:, BUS_AREA)). LOAD_ZONE can also take the following string values:

- 'all' - use a **single zone** for the entire system (**return** scalar)
- 'area' - use LOAD_ZONE = BUS(:, BUS_AREA), same as default
- 'bus' - use a different zone **for** each bus (i.e. to compute final values of bus-wise loads, including voltage dependent fixed loads and or dispatchable loads)

OPT - (optional) option **struct**, with the following fields:

- 'type' - string specifying types of loads to include, default is 'BOTH' if GEN is provided, otherwise 'FIXED'
- 'FIXED' : sum only fixed loads

(continues on next page)

(continued from previous page)

```

'DISPATCHABLE' : sum only dispatchable loads
'BOTH'          : sum both fixed and dispatchable loads
'nominal' - 1 : use nominal load for dispatchable loads
            0 : (default) use actual realized load for
                dispatchable loads

```

For backward compatibility with MATPOWER 4.x, OPT can also take the form of a string, with the same options as OPT.type above. In this case, again for backward compatibility, it is the "nominal" load that is computed for dispatchable loads, not the actual realized load. Using a string for OPT is deprecated and will be removed in a future version.

MPOPT - (optional) MATPOWER options struct, which may specify a voltage dependent (ZIP) load model for fixed loads

Examples:

Return the total active load for each area as defined in BUS_AREA.

```
Pd = total_load(bus);
```

Return total active and reactive load, fixed and dispatchable, for entire system.

```
[Pd, Qd] = total_load(bus, gen, 'all');
```

Return the total of the nominal dispatchable loads at buses 10-20.

```
load_zone = zeros(nb, 1);
load_zone(10:20) = 1;
opt = struct('type', 'DISPATCHABLE', 'nominal', 1);
Pd = total_load(mpc, load_zone, opt)
```

See also `scale_load()` (page 351).

5.2.13 Private Feature Detection Functions

have_feature_e4st

have_feature_e4st()

`have_feature_e4st()` (page 354) - Detect availability/version info for E4ST.

Private feature detection function implementing 'e4st' tag for `have_feature()` to detect availability/version of E4ST, the Engineering, Economic, and Environmental Electricity Simulation Tool (<https://e4st.com>).

See also `have_feature()`.

have_feature_minopf

have_feature_minopf()

have_feature_minopf() (page 355) - Detect availability/version info for MINOPF.

Private feature detection function implementing 'minopf' tag for `have_feature()` to detect availability/version of MINOPF, a MINOS-based optimal power flow (OPF) solver.

See also `have_feature()`, `minopf`.

have_feature_most

have_feature_most()

have_feature_most() (page 355) - Detect availability/version info for MOST.

Private feature detection function implementing ':func:`most`' tag for `have_feature()` to detect availability/version of MOST (MATPOWER Optimal Scheduling Tool).

See also `have_feature()`, `most()`.

have_feature_mp_core

have_feature_mp_core()

have_feature_mp_core() (page 355) - Detect availability of MP-Core.

Private feature detection function implementing 'mp_core' tag for `have_feature()` to detect availability/version of MP-Core.

See also `have_feature()`.

have_feature_pdipmopf

have_feature_pdipmopf()

have_feature_pdipmopf() (page 355) - Detect availability/version info for PDIPMOPF.

Private feature detection function implementing 'pdipmopf' tag for `have_feature()` to detect availability/version of PDIPMOPF, a primal-dual interior point method optimal power flow (OPF) solver included in TSPOPF. (<https://www.pserc.cornell.edu/tspopf>)

See also `have_feature()`, `pdipmopf`.

have_feature_regexp_split

have_feature_regexp_split()

have_feature_regexp_split() (page 356) - Detect availability/version info for REGEXP 'split'.

Private feature detection function implementing 'regexp_split' tag for `have_feature()` to detect support for the 'split' argument to REGEXP.

See also `have_feature()`, `regexp`.

have_feature_scpdipmopf

have_feature_scpdipmopf()

have_feature_scpdipmopf() (page 356) - Detect availability/version info for SCPDIPMOPF.

Private feature detection function implementing 'scpdipmopf' tag for `have_feature()` to detect availability/version of SCPDIPMOPF, step-controlled primal-dual interior point method optimal power flow (OPF) solver included in TSOPF. (<https://www.pserc.cornell.edu/tspopf>)

See also `have_feature()`, `scpdipmopf`.

have_feature_sdp_pf

have_feature_sdp_pf()

have_feature_sdp_pf() (page 356) - Detect availability/version info for SDP_PF.

Private feature detection function implementing 'sdp_pf' tag for `have_feature()` to detect availability/version of SDP_PF, a MATPOWER extension for applications of semi-definite programming relaxations of power flow equations (https://github.com/MATPOWER/mx-sdp_pf/).

See also `have_feature()`.

have_feature_smartmarket

have_feature_smartmarket()

have_feature_smartmarket() (page 356) - Detect availability/version info for SMARTMARKET.

Private feature detection function implementing 'smartmarket' tag for `have_feature()` to detect availability/version of RUNMARKET and related files for running an energy auction, found under smartmarket in MATPOWER Extras. (<https://github.com/MATPOWER/matpower-extras/>).

See also `have_feature()`, `runmarket`.

have_feature_syngrid

have_feature_syngrid()

have_feature_syngrid() (page 357) - Detect availability/version info for SynGrid.

Private feature detection function implementing 'syngrid' tag for `have_feature()` to detect availability/version of SynGrid, Synthetic Grid Creation for MATPOWER (<https://github.com/MATPOWER/mx-syngrid>).

See also `have_feature()`, `syngrid`.

have_feature_table

have_feature_table()

have_feature_table() (page 357) - Detect availability/version info for table.

Private feature detection function implementing 'table' tag for `have_feature()` to detect availability/version of TABLE, included in MATLAB R2013b and as of this writing in Mar 2024, available for Octave as Tablicious: <https://github.com/apjanke/octave-tablicious>

See also `have_feature()`, `table`.

have_feature_tralmopf

have_feature_tralmopf()

have_feature_tralmopf() (page 357) - Detect availability/version info for TRALMOPF

Private feature detection function implementing 'tralmopf' tag for `have_feature()` to detect availability/version of TRALMOPF, trust region based augmented Lagrangian optimal power flow (OPF) solver included in TSPOPF. (<https://www.pserc.cornell.edu/tspopf>)

See also `have_feature()`, `tralmopf`.

5.2.14 Other Functions

connected_components

`connected_components(C, groups, unvisited)`

connected_components() (page 357) - Returns the connected components of a graph.

```
[GROUPS, ISOLATED] = CONNECTED_COMPONENTS(C)
```

Returns the connected components of a directed graph, specified by a node-branch incidence matrix `C`, where `C(I, J) = -1` if node `J` is connected to the beginning of branch `I`, `1` if it is connected to the end of branch `I`, and zero otherwise. The return value `GROUPS` is a cell array of vectors of the node indices for each component. The second return value `ISOLATED` is a vector of indices of isolated nodes that have no connecting branches.

mpoption_info_clp

mpoption_info_clp(*selector*)

mpoption_info_clp() (page 358) - Returns MATPOWER option info for CLP.

```
DEFAULT_OPTS = MPOPTION_INFO_CLP('D')
VALID_OPTS   = MPOPTION_INFO_CLP('V')
EXCEPTIONS   = MPOPTION_INFO_CLP('E')
```

Returns a structure **for** CLP options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mpooption().

mpoption_info_cplex

mpoption_info_cplex(*selector*)

mpoption_info_cplex() (page 358) - Returns MATPOWER option info for CPLEX.

```
DEFAULT_OPTS = MPOPTION_INFO_CPLEX('D')
VALID_OPTS   = MPOPTION_INFO_CPLEX('V')
EXCEPTIONS   = MPOPTION_INFO_CPLEX('E')
```

Returns a structure **for** CPLEX options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mpooption().

mpoption_info_fmincon

mpoption_info_fmincon(*selector*)

mpoption_info_fmincon() (page 359) - Returns MATPOWER option info for FMINCON.

```

DEFAULT_OPTS = MPOPTION_INFO_FMINCON('D')
VALID_OPTS   = MPOPTION_INFO_FMINCON('V')
EXCEPTIONS   = MPOPTION_INFO_FMINCON('E')

```

Returns a structure **for** FMINCON options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_glpk

mpoption_info_glpk(*selector*)

mpoption_info_glpk() (page 359) - Returns MATPOWER option info for GLPK.

```

DEFAULT_OPTS = MPOPTION_INFO_GLPK('D')
VALID_OPTS   = MPOPTION_INFO_GLPK('V')
EXCEPTIONS   = MPOPTION_INFO_GLPK('E')

```

Returns a structure **for** GLPK options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_gurobi

mpoption_info_gurobi(*selector*)

mpoption_info_gurobi() (page 360) - Returns MATPOWER option info for Gurobi.

```
DEFAULT_OPTS = MPOPTION_INFO_GUROBI('D')
VALID_OPTS   = MPOPTION_INFO_GUROBI('V')
EXCEPTIONS   = MPOPTION_INFO_GUROBI('E')
```

Returns a structure **for** Gurobi options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_intlinprog

mpoption_info_intlinprog(*selector*)

mpoption_info_intlinprog() (page 360) - Returns MATPOWER option info for INTLINPROG.

```
DEFAULT_OPTS = MPOPTION_INFO_INTLINPROG('D')
VALID_OPTS   = MPOPTION_INFO_INTLINPROG('V')
EXCEPTIONS   = MPOPTION_INFO_INTLINPROG('E')
```

Returns a structure **for** INTLINPROG options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_ipopt

mpoption_info_ipopt(*selector*)

mpoption_info_ipopt() (page 361) - Returns MATPOWER option info for IPOPT.

```

DEFAULT_OPTS = MPOPTION_INFO_IPOPT('D')
VALID_OPTS   = MPOPTION_INFO_IPOPT('V')
EXCEPTIONS   = MPOPTION_INFO_IPOPT('E')

```

Returns a structure **for** IPOPT options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_knitro

mpoption_info_knitro(*selector*)

mpoption_info_knitro() (page 361) - Returns MATPOWER option info for Artelys Knitro.

```

DEFAULT_OPTS = MPOPTION_INFO_KNITRO('D')
VALID_OPTS   = MPOPTION_INFO_KNITRO('V')
EXCEPTIONS   = MPOPTION_INFO_KNITRO('E')

```

Returns a structure **for** Knitro options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_linprog

mpoption_info_linprog(*selector*)

mpoption_info_linprog() (page 362) - Returns MATPOWER option info for LINPROG.

```
DEFAULT_OPTS = MPOPTION_INFO_LINPROG('D')
VALID_OPTS   = MPOPTION_INFO_LINPROG('V')
EXCEPTIONS   = MPOPTION_INFO_LINPROG('E')
```

Returns a structure **for** LINPROG options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_mips

mpoption_info_mips(*selector*)

mpoption_info_mips() (page 362) - Returns MATPOWER option info for MIPS (optional fields).

```
DEFAULT_OPTS = MPOPTION_INFO_MIPS('D')
VALID_OPTS   = MPOPTION_INFO_MIPS('V')
EXCEPTIONS   = MPOPTION_INFO_MIPS('E')
```

Returns a structure **for** MIPS options **for** MATPOWER containing ...

(1) default options,
(2) valid options, **or**
(3) NESTED_STRUCT_COPY exceptions **for** setting options
... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_mosek

mpoption_info_mosek(*selector*)

mpoption_info_mosek() (page 363) - Returns MATPOWER option info for MOSEK.

```

DEFAULT_OPTS = MPOPTION_INFO_MOSEK('D')
VALID_OPTS   = MPOPTION_INFO_MOSEK('V')
EXCEPTIONS   = MPOPTION_INFO_MOSEK('E')

```

Returns a structure **for** MOSEK options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_osqp

mpoption_info_osqp(*selector*)

mpoption_info_osqp() (page 363) - Returns MATPOWER option info for OSQP.

```

DEFAULT_OPTS = MPOPTION_INFO_OSQP('D')
VALID_OPTS   = MPOPTION_INFO_OSQP('V')
EXCEPTIONS   = MPOPTION_INFO_OSQP('E')

```

Returns a structure **for** OSQP options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_info_quadprog

mpoption_info_quadprog(*selector*)

mpoption_info_quadprog() (page 364) - Returns MATPOWER option info for QUADPROG.

```

DEFAULT_OPTS = MPOPTION_INFO_QUADPROG('D')
VALID_OPTS   = MPOPTION_INFO_QUADPROG('V')
EXCEPTIONS   = MPOPTION_INFO_QUADPROG('E')

```

Returns a structure **for** QUADPROG options **for** MATPOWER containing ...

(1) default options,
 (2) valid options, **or**
 (3) NESTED_STRUCT_COPY exceptions **for** setting options
 ... depending on the value of the **input** argument.

This **function** is used by MPOPTION to **set** default options, check validity of option names **or** modify option setting/copying behavior **for** this subset of optional MATPOWER options.

See also mppoption().

mpoption_old

mpoption_old(*varargin*)

mpoption_old() (page 364) - Used to set and retrieve old-style MATPOWER options vector.

```

OPT = MPOPTION_OLD
    returns the default options vector

OPT = MPOPTION_OLD(NAME1, VALUE1, NAME2, VALUE2, ...)
    returns the default options vector with new values for up to 7
    options, NAME# is the name of an option, and VALUE# is the new
    value.

OPT = MPOPTION_OLD(OPT, NAME1, VALUE1, NAME2, VALUE2, ...)
    same as above except it uses the options vector OPT as a base
    instead of the default options vector.

```

Examples:

```

opt = mppoption_old('PF_ALG', 2, 'PF_TOL', 1e-4);
opt = mppoption_old(opt, 'OPF_ALG', 565, 'VERBOSE', 2);

```

The currently defined options are as follows:

idx	NAME, default	description [options]
<hr/>		
power flow options		
1	PF_ALG, 1	AC power flow algorithm
	[1 - Newton's method]
	[2 - Fast-Decoupled (XB version)]
	[3 - Fast-Decoupled (BX version)]

(continues on next page)

(continued from previous page)

```

[ 4 - Gauss-Seidel ]
2 - PF_TOL, 1e-8      termination tolerance on per unit
                      P & Q mismatch
3 - PF_MAX_IT, 10      maximum number of iterations for
                      Newton's method
4 - PF_MAX_IT_FD, 30    maximum number of iterations for
                      fast decoupled method
5 - PF_MAX_IT_GS, 1000  maximum number of iterations for
                      Gauss-Seidel method
6 - ENFORCE_Q_LIMS, 0   enforce gen reactive power limits
                      at expense of |V|
[ 0 - do NOT enforce limits ]
[ 1 - enforce limits, simultaneous bus type conversion ]
[ 2 - enforce limits, one-at-a-time bus type conversion ]
10 - PF_DC, 0           DC modeling for power flow & OPF
[ 0 - use AC formulation & corresponding algorithm options ]
[ 1 - use DC formulation, ignore AC algorithm options ]
OPF options
11 - OPF_ALG, 0          solver to use for AC OPF
[ 0 - choose default solver based on availability in the ]
[ following order, 540, 560 ]
[ 500 - MINOPF, MINOS-based solver, requires optional ]
[ MEX-based MINOPF package, available from: ]
[ http://www.pserc.cornell.edu/minopf/ ]
[ 520 - fmincon, MATLAB Optimization Toolbox >= 2.x ]
[ 540 - PDIPM, primal/dual interior point method, requires ]
[ optional MEX-based TSPOPF package, available from: ]
[ http://www.pserc.cornell.edu/tspopf/ ]
[ 545 - SC-PDIPM, step-controlled variant of PDIPM, requires ]
[ TSPOPF (see 540) ]
[ 550 - TRALM, trust region based augmented Lagrangian ]
[ method, requires TSPOPF (see 540) ]
[ 560 - MIPS, MATPOWER Interior Point Solver ]
[ primal/dual interior point method (pure MATLAB) ]
[ 565 - MIPS-sc, step-controlled variant of MIPS ]
[ primal/dual interior point method (pure MATLAB) ]
[ 580 - IPOPT, requires MEX interface to IPOPT solver ]
[ available from: https://projects.coin-or.org/Ipopt/ ]
[ 600 - Artelys Knitro, requires Knitro solver, available from: ]
[ https://www.artelys.com/solvers/knitro/ ]
16 - OPF_VIOLATION, 5e-6 constraint violation tolerance
17 - CONSTR_TOL_X, 1e-4  termination tol on x for fmincon/Knitro
18 - CONSTR_TOL_F, 1e-4  termination tol on f for fmincon/Knitro
19 - CONSTR_MAX_IT, 0     max number of iterations for fmincon
[ 0 => default ]
24 - OPF_FLOW_LIM, 0      qty to limit for branch flow constraints
[ 0 - apparent power flow (limit in MVA) ]
[ 1 - active power flow (limit in MW) ]
[ 2 - current magnitude (limit in MVA at 1 p.u. voltage) ]
25 - OPF_IGNORE_ANG_LIM, 0 ignore angle difference limits for branches
                      even if specified [ 0 or 1 ]
26 - OPF_ALG_DC, 0        solver to use for DC OPF

```

(continues on next page)

(continued from previous page)

```

[ 0 - choose default solver based on availability in the ]
[ following order: 500, 600, 700, 100, 300, 200 ]
[ 100 - BPMPD, requires optional MEX-based BPMPD_MEX package ]
[ available from: http://www.pserc.cornell.edu/bpmpd/ ]
[ 200 - MIPS, MATLAB Interior Point Solver ]
[ primal/dual interior point method (pure MATLAB) ]
[ 250 - MIPS-sc, step-controlled variant of MIPS ]
[ 300 - MATLAB Optimization Toolbox, QUADPROG, LINPROG ]
[ 400 - IPOPT, requires MEX interface to IPOPT solver ]
[ available from: https://projects.coin-or.org/Ipopt/ ]
[ 500 - CPLEX, requires MATLAB interface to CPLEX solver ]
[ 600 - MOSEK, requires MATLAB interface to MOSEK solver ]
[ available from: https://www.mosek.com/ ]
[ 700 - GUROBI, requires Gurobi optimizer (v. 5+) ]
[ available from: https://www.gurobi.com ]
output options
31 - VERBOSE, 1 amount of progress info printed
[ 0 - print no progress info ]
[ 1 - print a little progress info ]
[ 2 - print a lot of progress info ]
[ 3 - print all progress info ]
32 - OUT_ALL, -1 controls pretty-printing of results
[ -1 - individual flags control what prints ]
[ 0 - do not print anything ]
[ (overrides individual flags) ]
[ 1 - print everything ]
[ (overrides individual flags) ]
33 - OUT_SYS_SUM, 1 print system summary [ 0 or 1 ]
34 - OUT_AREA_SUM, 0 print area summaries [ 0 or 1 ]
35 - OUT_BUS, 1 print bus detail [ 0 or 1 ]
36 - OUT_BRANCH, 1 print branch detail [ 0 or 1 ]
37 - OUT_GEN, 0 print generator detail [ 0 or 1 ]
(OUT_BUS also includes gen info)
38 - OUT_ALL_LIM, -1 controls what constraint info is printed
[ -1 - individual flags control what constraint info prints ]
[ 0 - no constraint info (overrides individual flags) ]
[ 1 - binding constraint info (overrides individual flags) ]
[ 2 - all constraint info (overrides individual flags) ]
39 - OUT_V_LIM, 1 control output of voltage limit info
[ 0 - do not print ]
[ 1 - print binding constraints only ]
[ 2 - print all constraints ]
[ (same options for OUT_LINE_LIM, OUT_PG_LIM, OUT_QG_LIM) ]
40 - OUT_LINE_LIM, 1 control output of line flow limit info
41 - OUT_PG_LIM, 1 control output of gen P limit info
42 - OUT_QG_LIM, 1 control output of gen Q limit info
44 - OUT_FORCE, 0 print results even if success = 0
[ 0 or 1 ]
52 - RETURN_RAW_DER, 0 return constraint and derivative info
in results.raw (in fields g, dg, df, d2f)
FMINCON options
55 - FMC_ALG, 4 algorithm used by fmincon for OPF

```

(continues on next page)

(continued from previous page)

```

                                for Optimization Toolbox 4 and later
[ 1 - active-set                                     ]
[ 2 - interior-point, w/default 'bfgs' Hessian approx ]
[ 3 - interior-point, w/ 'lbfgs' Hessian approx       ]
[ 4 - interior-point, w/exact user-supplied Hessian   ]
[ 5 - interior-point, w/Hessian via finite differences ]

Artelys Knitro options
  58 - KNITRO_OPT, 0          a non-zero integer N indicates that all
                              Knitro options should be handled by a
                              Knitro options file named
                              'knitro_user_options_N.txt'

IPOPT options
  60 - IPOPT_OPT, 0          See IPOPT_OPTIONS for details.

MINOPF options
  61 - MNS_FEASTOL, 0 (1e-3) primal feasibility tolerance,
                              set to value of OPF_VIOLATION by default
  62 - MNS_ROWTOL, 0 (1e-3) row tolerance
                              set to value of OPF_VIOLATION by default
  63 - MNS_XTOL, 0 (1e-3) x tolerance
                              set to value of CONSTR_TOL_X by default
  64 - MNS_MAJDAMP, 0 (0.5) major damping parameter
  65 - MNS_MINDAMP, 0 (2.0) minor damping parameter
  66 - MNS_PENALTY_PARM, 0 (1.0) penalty parameter
  67 - MNS_MAJOR_IT, 0 (200) major iterations
  68 - MNS_MINOR_IT, 0 (2500) minor iterations
  69 - MNS_MAX_IT, 0 (2500) iterations limit
  70 - MNS_VERBOSITY, -1
      [ -1 - controlled by VERBOSE option ]
      [ 0 - print nothing ]
      [ 1 - print only termination status message ]
      [ 2 - print termination status and screen progress ]
      [ 3 - print screen progress, report file (usually fort.9) ]
  71 - MNS_CORE, 0 (1200 * nb + 2 * (nb + ng)^2) memory allocation
  72 - MNS_SUPBASIC_LIM, 0 (2*nb + 2*ng) superbasics limit
  73 - MNS_MULT_PRICE, 0 (30) multiple price

MIPS (including MIPS-sc), PDIPM, SC-PDIPM, and TRALM options
  81 - PDIPM_FEASTOL, 0      feasibility (equality) tolerance
                              for MIPS, PDIPM and SC-PDIPM, set
                              to value of OPF_VIOLATION by default
  82 - PDIPM_GRADTOL, 1e-6   gradient tolerance for MIPS, PDIPM
                              and SC-PDIPM
  83 - PDIPM_COMPTOL, 1e-6   complementary condition (inequality)
                              tolerance for MIPS, PDIPM and SC-PDIPM
  84 - PDIPM_COSTTOL, 1e-6   optimality tolerance for MIPS, PDIPM
                              and SC-PDIPM
  85 - PDIPM_MAX_IT, 150     maximum number of iterations for MIPS,
                              PDIPM and SC-PDIPM
  86 - SCPDIPM_RED_IT, 20    maximum number of MIPS-sc or SC-PDIPM

```

(continues on next page)

(continued from previous page)

```

87 - TRALM_FEASTOL, 0      reductions per iteration
                           feasibility tolerance for TRALM
                           set to value of OPF_VIOLATION by default
88 - TRALM_PRIMETOL, 5e-4  primal variable tolerance for TRALM
89 - TRALM_DUALTOL, 5e-4   dual variable tolerance for TRALM
90 - TRALM_COSTTOL, 1e-5   optimality tolerance for TRALM
91 - TRALM_MAJOR_IT, 40    maximum number of major iterations
92 - TRALM_MINOR_IT, 100   maximum number of minor iterations
93 - SMOOTHING_RATIO, 0.04 piecewise linear curve smoothing ratio
                           used in SC-PDIPM and TRALM

CPLEX options
95 - CPLEX_LPMETHOD, 0    solution algorithm for continuous LPs
    [ 0 - automatic: let CPLEX choose ]
    [ 1 - primal simplex ]
    [ 2 - dual simplex ]
    [ 3 - network simplex ]
    [ 4 - barrier ]
    [ 5 - sifting ]
    [ 6 - concurrent (dual, barrier, and primal) ]
96 - CPLEX_QPMETHOD, 0    solution algorithm for continuous QPs
    [ 0 - automatic: let CPLEX choose ]
    [ 1 - primal simplex optimizer ]
    [ 2 - dual simplex optimizer ]
    [ 3 - network optimizer ]
    [ 4 - barrier optimizer ]
97 - CPLEX_OPT, 0          See CPLEX_OPTIONS for details

MOSEK options
111 - MOSEK_LP_ALG, 0      solution algorithm for continuous LPs
                           (MSK_IPAR_OPTIMIZER)
    [ 0 - automatic: let MOSEK choose ]
    [ 1 - interior point ]
    [ 4 - primal simplex ]
    [ 5 - dual simplex ]
    [ 6 - primal dual simplex ]
    [ 7 - automatic simplex (MOSEK chooses which simplex method) ]
    [ 10 - concurrent ]
112 - MOSEK_MAX_IT, 0 (400) interior point max iterations
                           (MSK_IPAR_INTPNT_MAX_ITERATIONS)
113 - MOSEK_GAP_TOL, 0 (1e-8) interior point relative gap tolerance
                           (MSK_DPAR_INTPNT_TOL_REL_GAP)
114 - MOSEK_MAX_TIME, 0 (-1) maximum time allowed for solver
                           (MSK_DPAR_OPTIMIZER_MAX_TIME)
115 - MOSEK_NUM_THREADS, 0 (1) maximum number of threads to use
                           (MSK_IPAR_INTPNT_NUM_THREADS)
116 - MOSEK_OPT, 0         See MOSEK_OPTIONS for details

Gurobi options
121 - GRB_METHOD, -1       solution algorithm (Method)
    [ -1 - automatic, let Gurobi decide ]
    [ 0 - primal simplex ]

```

(continues on next page)

(continued from previous page)

```

[ 1 - dual simplex ]
[ 2 - barrier ]
[ 3 - concurrent (LP only) ]
[ 4 - deterministic concurrent (LP only) ]
122 - GRB_TIMELIMIT, Inf maximum time allowed for solver (TimeLimit)
123 - GRB_THREADS, 0 (auto) maximum number of threads to use (Threads)
124 - GRB_OPT, 0 See GUROBI_OPTIONS for details

```

psse_convert

psse_convert(warns, data, verbose)

psse_convert() (page 369) - Converts data read from PSS/E RAW file to MATPOWER case.

```

[MPC, WARNINGS] = PSSE_CONVERT(WARNINGS, DATA)
[MPC, WARNINGS] = PSSE_CONVERT(WARNINGS, DATA, VERBOSE)

```

Converts data read from a **version** RAW data file into a MATPOWER **case struct**.

Input:

WARNINGS : **cell** array of strings containing accumulated warning messages

DATA : **struct** read by PSSE_READ (see PSSE_READ **for** details).

VERBOSE : 1 to display progress info, 0 (default) otherwise

Output:

MPC : a MATPOWER **case struct** created from the PSS/E data

WARNINGS : **cell** array of strings containing updated accumulated warning messages

See also *psse_read()* (page 374).

psse_convert_hvdc

psse_convert_hvdc(dc, bus)

psse_convert_hvdc() (page 369) - Convert HVDC data from PSS/E RAW to MATPOWER.

```

DCLINE = PSSE_CONVERT_HVDC(DC, BUS)

```

Convert **all** two terminal HVDC **line** data read from a PSS/E RAW data file into MATPOWER format. Returns a dcline matrix **for** inclusion in a MATPOWER **case struct**.

Inputs:

DC : matrix of raw two terminal HVDC **line** data returned by PSSE_READ in data.twodc.num

BUS : MATPOWER bus matrix

(continues on next page)

(continued from previous page)

Output:

DCLINE : a MATPOWER dcline matrix suitable **for** inclusion in
a MATPOWER **case struct**.

See also [psse_convert\(\)](#) (page 369).

psse_convert_xfmr

psse_convert_xfmr(warns, trans2, trans3, verbose, baseMVA, bus, bus_name)

[psse_convert_xfmr\(\)](#) (page 370) - Convert transformer data from PSS/E RAW to MATPOWER.

```
[XFMR, BUS, WARNINGS] = PSSE_CONVERT_XFMR(WARNINGS, TRANS2, TRANS3, ...
                                         VERBOSE, BASEMVA, BUS)
[XFMR, BUS, WARNINGS, BUS_NAME] = PSSE_CONVERT_XFMR(WARNINGS, TRANS2, ...
                                                    TRANS3, VERBOSE, BASEMVA, BUS, BUS_NAME)
```

Convert **all** transformer data read from a PSS/E RAW data file into MATPOWER format. Returns a branch matrix corresponding to the transformers **and** an updated bus matrix, with additional buses added **for** the star points of three winding transformers.

Inputs:

WARNINGS : **cell** array of strings containing accumulated
warning messages
TRANS2 : matrix of raw two winding transformer data returned
by PSSE_READ in data.trans2.num
TRANS3 : matrix of raw three winding transformer data returned
by PSSE_READ in data.trans3.num
VERBOSE : **1** to **display** progress **info**, **0** (default) **otherwise**
BASEMVA : **system** MVA base
BUS : MATPOWER bus matrix
BUS_NAME: (optional) **cell** array of bus names

Outputs:

XFMR : MATPOWER branch matrix of transformer data
BUS : updated MATPOWER bus matrix, with additional buses
added **for** star points of three winding transformers
WARNINGS : **cell** array of strings containing updated accumulated
warning messages
BUS_NAME: (optional) updated **cell** array of bus names

See also [psse_convert\(\)](#) (page 369).

psse_parse**psse_parse**(records, sections, verbose, rev)*psse_parse()* (page 371) - Parses the data from a PSS/E RAW data file.

```

DATA = PSSE_PARSE(RECORDS, SECTIONS)
DATA = PSSE_PARSE(RECORDS, SECTIONS, VERBOSE)
DATA = PSSE_PARSE(RECORDS, SECTIONS, VERBOSE, REV)
[DATA, WARNINGS] = PSSE_PARSE(RECORDS, SECTIONS, ...)

```

Parses the data from a PSS/E RAW data file (as read by PSSE_READ) into a **struct**.

Inputs:

- RECORDS : **cell** array of strings, corresponding to the lines in the RAW file
- SECTIONS : **struct** array with indices marking the beginning and **end** of each section, and the name of the section, fields are:
 - first : **index** into RECORDS of first **line** of section
 - last : **index** into RECORDS of last **line** of section
 - name : name of the section, as extracted from the END OF ... DATA comments
- VERBOSE : **1** (default) to **display** progress **info**, **0** **otherwise**
- REV : (optional) assume the **input** file is of this PSS/E revision number, attempts to determine REV from the file by default

Output(s):

- DATA : a **struct** with the following fields, each with two sub-fields, '**num**' and '**txt**' containing the numeric and **text** data read from the file **for** the corresponding section
 - id
 - bus
 - load**
 - gen
 - shunt
 - branch
 - trans2
 - trans3
 - area**
 - twodc
 - swshunt
- WARNINGS : **cell** array of strings containing accumulated **warning** messages

See also `psse2mpc()`, `psse_read()` (page 374), `psse_parse_section()` (page 372), `psse_parse_line()` (page 372).

psse_parse_line

psse_parse_line(*str*, *t*)

psse_parse_line() (page 372) - Reads and parses a single line from a PSS/E RAW data file.

```
[DATA, COMMENT] = PSSE_PARSE_LINE(FID)
[DATA, COMMENT] = PSSE_PARSE_LINE(FID, TEMPLATE)
[DATA, COMMENT] = PSSE_PARSE_LINE(STR)
[DATA, COMMENT] = PSSE_PARSE_LINE(STR, TEMPLATE)
```

Parses a **single line** from a PSS/E RAW data file, either directly read from the file, **or** passed as a string.

Inputs:

FID : (optional) file id of file from which to read the **line**
 STR : string containing the **line** to be parsed
 TEMPLATE : (optional) string of characters indicating how to interpret the **type** of the corresponding column, options are as follows:

d, f **or** g : integer floating point number to be converted via SSCANF with **%d, %f or %g, respectively**.
 D, F **or** G : integer floating point number, possibly enclosed in **single or double** quotes, to be converted via SSCANF with **%d, %f or %g, respectively**.
 c **or** s : character **or** string, possibly enclosed in **single or double** quotes, which are stripped from the string

Note: Data **columns** in STR that have no valid corresponding entry in TEMPLATE (beyond **end** of TEMPLATE, **or** a character other than those listed, e.g. **'.'**) are returned as a string with no conversion. TEMPLATE entries **for** which there is no corresponding column are returned as **NaN or** empty string, depending on the **type**.

Outputs:

DATA : a **cell** array whose elements contain the contents of the corresponding column in the data, converted according to the TEMPLATE.
 COMMENT : (optional) possible comment at the **end** of the **line**

psse_parse_section

psse_parse_section(*warns*, *records*, *sections*, *s*, *verbose*, *label*, *template*)

psse_parse_section() (page 372) - Parses the data from a section of a PSS/E RAW data file.

```
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, SECTIONS, SIDX, ...
                                       VERBOSE, LABEL, TEMPLATE)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, SECTIONS, SIDX, ...
                                       VERBOSE, LABEL)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, SECTIONS, SIDX, ...
                                       VERBOSE)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, SECTIONS, SIDX)
```

(continues on next page)

(continued from previous page)

```
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, VERBOSE, LABEL, ...
                                     TEMPLATE)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, VERBOSE, LABEL)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS, VERBOSE)
[DATA, WARNINGS] = PSSE_PARSE_SECTION(WARNINGS, RECORDS)
```

Inputs:

WARNINGS : **cell** array of strings containing accumulated warning messages

RECORDS : a **cell** array of strings returned by PSSE_READ

SECTIONS : a **struct** array returned by PSSE_READ

SIDX : (optional) **index** if the section to be read if included, the RECORD indices are taken from SECTIONS(SIDX), **otherwise** use all RECORDS

VERBOSE : 1 to **display** progress info, 0 (default) **otherwise**

LABEL : (optional) name **for** the section, to be compared with the section name typically found in the END OF <LABEL> DATA comment at the **end** of each section

TEMPLATE : (optional) string of characters indicating how to interpret the **type** of the corresponding column, options are as follows:

- d, f **or** g : integer floating point number to be converted via SSCANF with %d, %f **or** %g, *respectively*.
- D, F **or** G : integer floating point number, possibly enclosed in **single or double** quotes, to be converted via SSCANF with %d, %f **or** %g, *respectively*.
- c **or** s : character **or** string, possibly enclosed in **single or double** quotes, which are stripped from the string

Note: Data **columns** in RECORDS that have no valid corresponding entry in TEMPLATE (beyond **end** of TEMPLATE, **or** a character other than those listed, e.g. '.') are returned in DATA.txt with no conversion. TEMPLATE entries **for** which there is no corresponding column in RECORDS are returned as NaN and empty, respectively, in DATA.num **and** DATA.txt.

Output:

DATA : a **struct** with two fields:

- num : matrix containing the numeric data **for** the section, **for** **columns** with no numeric data, num contain NaNs.
- txt : a **cell** array containing the non-numeric (**char/string**) data **for** the section, **for** **columns** with numeric data, txt entries are empty

WARNINGS : **cell** array of strings containing updated accumulated warning messages

See also psse2mpc(), [psse_parse\(\)](#) (page 371).

psse_read

psse_read(rawfile_name, verbose)

psse_read() (page 374) - Reads the data from a PSS/E RAW data file.

```
[RECORDS, SECTIONS] = PSSE_READ(RAWFILE_NAME)
```

```
[RECORDS, SECTIONS] = PSSE_READ(RAWFILE_NAME, VERBOSE)
```

Reads the data from a PSS/E RAW data file into a **cell** array of strings, corresponding to the lines/records in the file. It detects the beginning **and** ending indices of each section as well as **any** Q **record** used to indicate the **end** of the data.

Input:

RAWFILE_NAME : name of the PSS/E RAW file to be read
(opened directly with FILEREAD)

VERBOSE : 1 to **display** progress **info**, 0 (default) **otherwise**

Output:

RECORDS : a **cell** array of strings, one **for** each **line** in the file (new **line** characters **not** included)

SECTIONS : a **struct** array with the following fields

first : **index** into RECORDS of first **line** of the section

last : **index** into RECORDS of last **line** of the section

name : name of the section (**if** available) extracted

from the 'END OF <NAME> DATA, BEGIN ... DATA'

comment typically found in the terminator **line**

See also psse2mpc().

5.3 Legacy Tests

5.3.1 Legacy MATPOWER Tests

t_apply_changes

t_apply_changes(quiet)

t_apply_changes() (page 374) - Tests for *apply_changes()* (page 331).

t_auction_minopf**t_auction_minopf**(*quiet*)*t_auction_minopf*() (page 375) - Tests for code in auction.m, using MINOPF solver.**t_auction_mips****t_auction_mips**(*quiet*)*t_auction_mips*() (page 375) - Tests for code in auction.m, using MIPS solver.**t_auction_tspopf_pdipm****t_auction_tspopf_pdipm**(*quiet*)*t_auction_tspopf_pdipm*() (page 375) - Tests for code in auction.m, using PDIPMOPF solver.**t_chgtab****t_chgtab**()*t_chgtab*() (page 375) - Returns a change table for testing *apply_changes*() (page 331).**t_cpf****t_cpf**(*quiet*)*t_cpf*() (page 375) - Tests for legacy continuation power flow.**t_dcline****t_dcline**(*quiet*)*t_dcline*() (page 375) - Tests for DC line extension in *toggle_dcline*() (page 307).**t_ext2int2ext****t_ext2int2ext**(*quiet*)*t_ext2int2ext*() (page 375) - Tests *ext2int*(), *int2ext*(), and related functions.Includes tests for *get_reorder*() (page 262), *set_reorder*() (page 262), *e2i_data*() (page 257), *i2e_data*() (page 260), *e2i_field*() (page 258), *i2e_field*() (page 261), *ext2int*(), and *int2ext*() .

t_feval_w_path

t_feval_w_path(*quiet*)

t_feval_w_path() (page 376) - Tests for *feval_w_path()* (page 336).

t_get_losses

t_get_losses(*quiet*)

t_get_losses() (page 376) - Tests for *get_losses()* (page 340).

t_hasPQcap

t_hasPQcap(*quiet*)

t_hasPQcap() (page 376) - Tests for *hasPQcap()* (page 341).

t_hessian

t_hessian(*quiet*)

t_hessian() (page 376) - Numerical tests of 2nd derivative code.

t_islands

t_islands(*quiet*)

t_islands() (page 376) - Tests for *find_islands()* (page 337), *extract_islands()* (page 335), *connected_components()* (page 357) and *case_info()* (page 333).

t_jacobian

t_jacobian(*quiet*)

t_jacobian() (page 376) - Numerical tests of partial derivative code.

t_load2disp**t_load2disp**(*quiet*)*t_load2disp*() (page 377) - Tests for load2disp().**t_loadcase****t_loadcase**(*quiet*)*t_loadcase*() (page 377) - Test that loadcase() works with a struct as well as case file.**t_makeLODF****t_makeLODF**(*quiet*)*t_makeLODF*() (page 377) - Tests for *makeLODF*() (page 329).**t_makePTDF****t_makePTDF**(*quiet*)*t_makePTDF*() (page 377) - Tests for *makePTDF*() (page 329).**t_margcost****t_margcost**(*quiet*)*t_margcost*() (page 377) - Tests for margcost().**t_miqps_matpower****t_miqps_matpower**(*quiet*)*t_miqps_matpower*() (page 377) - Tests of MIQP solvers via (deprecated) *miqps_matpower*() (page 327).**t_modcost****t_modcost**(*quiet*)*t_modcost*() (page 377) - Tests for code in modcost().

t_moption

t_moption(*quiet*)

[*t_moption\(\)*](#) (page 378) - Tests for moption().

t_moption_ov

t_moption_ov()

[*t_moption_ov\(\)*](#) (page 378) - Example of option overrides from file.

t_off2case

t_off2case(*quiet*)

[*t_off2case\(\)*](#) (page 378) - Tests for off2case.

t_opf_dc_bpmpd

t_opf_dc_bpmpd(*quiet*)

[*t_opf_dc_bpmpd\(\)*](#) (page 378) - Tests for legacy DC optimal power flow using BPMPD_MEX solver.

t_opf_dc_clp

t_opf_dc_clp(*quiet*)

[*t_opf_dc_clp\(\)*](#) (page 378) - Tests for legacy DC optimal power flow using CLP solver.

t_opf_dc_cplex

t_opf_dc_cplex(*quiet*)

[*t_opf_dc_cplex\(\)*](#) (page 378) - Tests for legacy DC optimal power flow using CPLEX solver.

t_opf_dc_default

t_opf_dc_default(*quiet*)

[*t_opf_dc_default\(\)*](#) (page 378) - Tests for legacy DC optimal power flow using DEFAULT solver.

t_opf_dc_glpk**t_opf_dc_glpk**(*quiet*)*t_opf_dc_glpk()* (page 379) - Tests for legacy DC optimal power flow using GLPK solver.**t_opf_dc_gurobi****t_opf_dc_gurobi**(*quiet*)*t_opf_dc_gurobi()* (page 379) - Tests for legacy DC optimal power flow using Gurobi solver.**t_opf_dc_ipopt****t_opf_dc_ipopt**(*quiet*)*t_opf_dc_ipopt()* (page 379) - Tests for legacy DC optimal power flow using MIPS solver.**t_opf_dc_mips****t_opf_dc_mips**(*quiet*)*t_opf_dc_mips()* (page 379) - Tests for legacy DC optimal power flow using MIPS solver.**t_opf_dc_mips_sc****t_opf_dc_mips_sc**(*quiet*)*t_opf_dc_mips_sc()* (page 379) - Tests for legacy DC optimal power flow using MIPS-sc solver.**t_opf_dc_mosek****t_opf_dc_mosek**(*quiet*)*t_opf_dc_mosek()* (page 379) - Tests for legacy DC optimal power flow using MOSEK solver.**t_opf_dc_osqp****t_opf_dc_osqp**(*quiet*)*t_opf_dc_osqp()* (page 379) - Tests for legacy DC optimal power flow using OSQP solver.

t_opf_dc_ot

t_opf_dc_ot(*quiet*)

t_opf_dc_ot() (page 380) - Tests for legacy DC optimal power flow using Opt Tbx solvers.

t_opf_default

t_opf_default(*quiet*)

t_opf_default() (page 380) - Tests for legacy AC optimal power flow using default solver.

t_opf_fmincon

t_opf_fmincon(*quiet*)

t_opf_fmincon() (page 380) - Tests for legacy FMINCON-based optimal power flow.

t_opf_ipopt

t_opf_ipopt(*quiet*)

t_opf_ipopt() (page 380) - Tests for legacy IPOPT-based AC optimal power flow.

t_opf_knitro

t_opf_knitro(*quiet*)

t_opf_knitro() (page 380) - Tests for legacy Artelys Knitro-based optimal power flow.

t_opf_minopf

t_opf_minopf(*quiet*)

t_opf_minopf() (page 380) - Tests for legacy MINOS-based optimal power flow.

t_opf_mips

t_opf_mips(*quiet*)

t_opf_mips() (page 380) - Tests for legacy MIPS-based AC optimal power flow.

t_opf_model

t_opf_model(*quiet*)

[*t_opf_model\(\)*](#) (page 381) - Tests for [*opf_model*](#) (page 223).

t_opf_softlims

t_opf_softlims(*quiet*)

[*t_opf_softlims\(\)*](#) (page 381) - Tests for userfcn callbacks (softlims) w/OPF.

Includes high-level tests of soft limits implementations.

t_opf_tspopf_pdipm

t_opf_tspopf_pdipm(*quiet*)

[*t_opf_tspopf_pdipm\(\)*](#) (page 381) - Tests for legacy PDIPM-based optimal power flow.

t_opf_tspopf_scpdipm

t_opf_tspopf_scpdipm(*quiet*)

[*t_opf_tspopf_scpdipm\(\)*](#) (page 381) - Tests for legacy SCPDIPM-based optimal power flow.

t_opf_tspopf_tralm

t_opf_tspopf_tralm(*quiet*)

[*t_opf_tspopf_tralm\(\)*](#) (page 381) - Tests for legacy TRALM-based optimal power flow.

t_opf_userfcns

t_opf_userfcns(*quiet*)

[*t_opf_userfcns\(\)*](#) (page 381) - Tests for userfcn callbacks (reserves/iflms) w/OPF.

Includes high-level tests of reserves and iflms implementations.

t_pf_ac

t_pf_ac(*quiet*)

[t_pf_ac\(\)](#) (page 382) - Tests for legacy AC power flow solvers.

t_pf_dc

t_pf_dc(*quiet*)

[t_pf_dc\(\)](#) (page 382) - Tests for legacy DC power flow solver.

t_pf_radial

t_pf_radial(*quiet*)

[t_pf_radial\(\)](#) (page 382) - Tests for legacy distribution power flow solvers.

t_printpf

t_printpf(*quiet*)

[t_printpf\(\)](#) (page 382) - Tests for `printpf()`.

t_psse

t_psse(*quiet*)

[t_psse\(\)](#) (page 382) - Tests for `psse2mpc()` and related functions.

t_qps_matpower

t_qps_matpower(*quiet*)

[t_qps_matpower\(\)](#) (page 382) - Tests of QP solvers via (deprecated) [qps_matpower\(\)](#) (page 327).

t_runmarket

t_runmarket(*quiet*)

[t_runmarket\(\)](#) (page 382) - Tests for `runmkt`, `smartmkt` and `auction`.

t_runopf_w_res**t_runopf_w_res**(*quiet*)*t_runopf_w_res()* (page 383) - Tests *runopf_w_res()* (page 238) and the associated callbacks.**t_scale_load****t_scale_load**(*quiet*)*t_scale_load()* (page 383) - Tests for *scale_load()* (page 351).**t_total_load****t_total_load**(*quiet*)*t_total_load()* (page 383) - Tests for *total_load()* (page 353).**t_totcost****t_totcost**(*quiet*)*t_totcost()* (page 383) - Tests for *totcost()*.**t_vdep_load****t_vdep_load**(*quiet*)*t_vdep_load()* (page 383) - Test voltage dependent ZIP load model for legacy PF, CPF, OPF.

5.3.2 Legacy MATPOWER Test Data

opf_nle_fcn1**opf_nle_fcn1**(*x*)*opf_nle_fcn1()* (page 383) - Example user-defined nonlinear OPF constraint function.

opf_nle_hess1**opf_nle_hess1**(*x*, *lambda*)*opf_nle_hess1*() (page 384) - Example user-defined nonlinear OPF constraint Hessian.**t_auction_case****t_auction_case**()*t_auction_case*() (page 384) - Power flow data for testing auction code.

Please see caseformat for details on the case file format.

t_case30_userfcns**t_case30_userfcns**()*t_case30_userfcns*() (page 384) - Power flow data for 30 bus, 6 gen case w/reserves & iflms.

Please see caseformat for details on the case file format.

Same as case30.m, but with fixed reserve and interface flow limit data. The reserve data is defined in the fields of mpc.reserves and the interface flow limit data in mpc.if at the bottom of the file.

t_case9_dcline**t_case9_dcline**()*t_case9_dcline*() (page 384) - Same as *t_case9_opfv2*() (page 385) with addition of DC line data.

Please see caseformat for details on the case file format.

See also *toggle_dcline*() (page 307), *idx_dcline*() (page 346).**t_case9_opf****t_case9_opf**()*t_case9_opf*() (page 384) - Power flow data for 9 bus, 3 generator case, with OPF data.

Please see caseformat for details on the case file format.

t_case9_opfv2**t_case9_opfv2()**

t_case9_opfv2() (page 385) - Power flow data for 9 bus, 3 generator case, with OPF data.

Please see caseformat for details on the case file format.

t_case9_pf**t_case9_pf()**

t_case9_pf() (page 385) - Power flow data for 9 bus, 3 generator case, no OPF data.

Please see caseformat for details on the case file format.

t_case9_pfv2**t_case9_pfv2()**

t_case9_pfv2() (page 385) - Power flow data for 9 bus, 3 generator case, no OPF data.

Please see caseformat for details on the case file format.

t_case9_save2psse**t_case9_save2psse()**

t_case9_save2psse() (page 385) - Power flow data to test save2psse().

Please see caseformat for details on the case file format.

t_case_ext**t_case_ext()**

t_case_ext() (page 385) - Case data in external format used to test ext2int() and int2ext().

t_case_int**t_case_int()**

t_case_int() (page 385) - Case data in internal format used to test ext2int() and int2ext().

t_cpf_cb1

t_cpf_cb1(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

t_cpf_cb1() (page 386) - User callback function 1 for continuation power flow testing.

t_cpf_cb2

t_cpf_cb2(*k, nx, cx, px, done, rollback, evnts, cb_data, cb_args, results*)

t_cpf_cb2() (page 386) - User callback function 2 for continuation power flow testing.

Previous Versions

Reference documentation for previous versions of MATPOWER can be found in the corresponding Function Reference.

- [MATPOWER 4.0 Function Reference](#)
- [MATPOWER 4.1 Function Reference](#)
- [MATPOWER 5.0 Function Reference](#)
- [MATPOWER 5.1 Function Reference](#)
- [MATPOWER 6.0 Function Reference](#)
- [MATPOWER 7.0 Function Reference](#)
- [MATPOWER 7.1 Function Reference](#)

A

- `add_aux_data()` (*mp.math_model* method), 126
- `add_constraints()` (*mp.math_model* method), 127
- `add_constraints()` (*mp.mm_element* method), 147
- `add_constraints()` (*mp.mme_branch_opf_ac* method), 150
- `add_constraints()` (*mp.mme_branch_opf_acc* method), 151
- `add_constraints()` (*mp.mme_branch_opf_acp* method), 151
- `add_constraints()` (*mp.mme_branch_opf_dc* method), 152
- `add_constraints()` (*mp.mme_bus_opf_acc* method), 153
- `add_constraints()` (*mp.mme_buslink_opf_acc* method), 205
- `add_constraints()` (*mp.mme_buslink_opf_acp* method), 205
- `add_constraints()` (*mp.mme_buslink_pf_ac* method), 202
- `add_constraints()` (*mp.mme_buslink_pf_acc* method), 203
- `add_constraints()` (*mp.mme_buslink_pf_acp* method), 203
- `add_constraints()` (*mp.mme_gen_opf_ac* method), 156
- `add_constraints()` (*mp.mme_gen_opf_ac_oval* method), 215
- `add_constraints()` (*mp.mme_gen_opf_dc* method), 156
- `add_constraints()` (*mp.mme_legacy_dcline_opf* method), 214
- `add_constraints()` (*mp.mme_reserve_gen* method), 181
- `add_constraints()` (*mp.mme_reserve_zone* method), 181
- `add_costs()` (*mp.math_model* method), 127
- `add_costs()` (*mp.math_model_pf* method), 129
- `add_costs()` (*mp.mm_element* method), 148
- `add_costs()` (*mp.mme_gen_opf* method), 155
- `add_costs()` (*mp.mme_gen_opf_ac* method), 156
- `add_costs()` (*mp.mme_legacy_dcline_opf* method), 214
- `add_costs()` (*mp.mme_reserve_gen* method), 181
- `add_elements()` (*mp.mapped_array* method), 171
- `add_legacy_cost()` (*mp.mm_shared_opf_legacy* method), 145
- `add_legacy_cost()` (*opf_model* method), 227
- `add_legacy_user_constraints()` (*mp.mm_shared_opf_legacy* method), 145
- `add_legacy_user_constraints_ac()` (*mp.mm_shared_opf_legacy* method), 145
- `add_legacy_user_costs()` (*mp.mm_shared_opf_legacy* method), 145
- `add_legacy_user_vars()` (*mp.mm_shared_opf_legacy* method), 145
- `add_named_set()` (*mp.math_model_opf_acci_legacy* method), 136
- `add_named_set()` (*mp.math_model_opf_accs_legacy* method), 137
- `add_named_set()` (*mp.math_model_opf_acpi_legacy* method), 138
- `add_named_set()` (*mp.math_model_opf_acps_legacy* method), 139
- `add_named_set()` (*mp.math_model_opf_dc_legacy* method), 141
- `add_named_set()` (*opf_model* method), 227
- `add_names()` (*mp.mapped_array* method), 171
- `add_node()` (*mp.net_model* method), 98
- `add_node_balance_constraints()` (*mp.math_model* method), 127
- `add_node_balance_constraints()` (*mp.math_model_opf_acci* method), 133
- `add_node_balance_constraints()` (*mp.math_model_opf_accs* method), 133
- `add_node_balance_constraints()` (*mp.math_model_opf_acpi* method), 134
- `add_node_balance_constraints()` (*mp.math_model_opf_acps* method), 134
- `add_node_balance_constraints()` (*mp.math_model_opf_acci* method), 136
- `add_node_balance_constraints()` (*mp.math_model_opf_accs* method), 137
- `add_node_balance_constraints()` (*mp.math_model_opf_acpi* method), 138
- `add_node_balance_constraints()` (*mp.math_model_opf_acps* method), 139
- `add_node_balance_constraints()` (*mp.math_model_opf_dc* method), 140
- `add_node_balance_constraints()` (*mp.math_model_pf_acci* method), 130
- `add_node_balance_constraints()` (*mp.math_model_pf_accs* method), 130
- `add_node_balance_constraints()` (*mp.math_model_pf_acpi* method), 131
- `add_node_balance_constraints()` (*mp.math_model_pf_acps* method), 131
- `add_node_balance_constraints()` (*mp.math_model_pf_dc* method), 132
- `add_nodes()` (*mp.net_model* method), 95
- `add_nodes()` (*mp.mm_element* method), 113
- `add_port()` (*mp.net_model* method), 98
- `add_state()` (*mp.net_model* method), 98
- `add_states()` (*mp.net_model* method), 96
- `add_states()` (*mp.mm_element* method), 113
- `add_system_constraints()` (*mp.math_model* method), 127
- `add_system_constraints()` (*mp.math_model_opf_acci_legacy* method), 136
- `add_system_constraints()` (*mp.math_model_opf_accs_legacy* method), 137
- `add_system_constraints()` (*mp.math_model_opf_acpi_legacy* method), 139
- `add_system_constraints()` (*mp.math_model_opf_acps_legacy* method), 140
- `add_system_constraints()` (*mp.math_model_opf_dc_legacy* method), 141
- `add_system_costs()` (*mp.math_model* method), 128
- `add_system_costs()` (*mp.math_model_opf_acci_legacy* method),

136
 add_system_costs() (mp.math_model_opf_accs_legacy method), 137
 add_system_costs() (mp.math_model_opf_acpi_legacy method), 139
 add_system_costs() (mp.math_model_opf_acps_legacy method), 140
 add_system_costs() (mp.math_model_opf_dc_legacy method), 141
 add_system_vars() (mp.math_model method), 126
 add_system_vars() (mp.math_model_opf method), 135
 add_system_vars() (mp.math_model_pf method), 129
 add_system_vars_pf() (mp.mm_shared_pfcpf_acci method), 143
 add_system_vars_pf() (mp.mm_shared_pfcpf_accs method), 143
 add_system_vars_pf() (mp.mm_shared_pfcpf_acpi method), 144
 add_system_vars_pf() (mp.mm_shared_pfcpf_acps method), 144
 add_system_vars_pf() (mp.mm_shared_pfcpf_dc method), 144
 add_system_varset_pf() (mp.mm_shared_pfcpf_ac method), 141
 add_userfcn() (built-in function), 305
 add_var() (mp.net_model method), 99
 add_vars() (mp.math_model method), 126
 add_vars() (mp.math_model_opf_acci_legacy method), 136
 add_vars() (mp.math_model_opf_accs_legacy method), 137
 add_vars() (mp.math_model_opf_acpi_legacy method), 139
 add_vars() (mp.math_model_opf_acps_legacy method), 140
 add_vars() (mp.math_model_opf_dc_legacy method), 141
 add_vars() (mp.mm_element method), 147
 add_vars() (mp.mme_buslink_pf_ac method), 202
 add_vars() (mp.mme_gen_opf method), 155
 add_vars() (mp.mme_legacy_dcline_opf method), 214
 add_vars() (mp.mme_reserve_gen method), 181
 add_vvars() (mp.net_model method), 97
 add_vvars() (mp.nm_element method), 113
 add_vvars() (mp.nme_bus3p_acc method), 197
 add_vvars() (mp.nme_bus3p_acp method), 197
 add_vvars() (mp.nme_bus_acc method), 119
 add_vvars() (mp.nme_bus_acp method), 119
 add_vvars() (mp.nme_bus_dc method), 120
 add_zvars() (mp.net_model method), 97
 add_zvars() (mp.nm_element method), 114
 add_zvars() (mp.nme_buslink method), 200
 add_zvars() (mp.nme_gen3p method), 197
 add_zvars() (mp.nme_gen_ac method), 120
 add_zvars() (mp.nme_gen_dc method), 121
 add_zvars() (mp.nme_legacy_dcline_ac method), 211
 add_zvars() (mp.nme_legacy_dcline_dc method), 212
 ang_diff_fcn() (mp.nme_branch_acc method), 118
 ang_diff_hess() (mp.nme_branch_acc method), 118
 ang_diff_params() (mp.nme_branch_opf method), 150
 ang_diff_prices() (mp.nme_branch_opf method), 150
 ang_diff_prices() (mp.nme_branch_opf_acc method), 151
 apply_changes() (built-in function), 331
 apply_vm_setpoint() (mp.dme_gen method), 55
 apply_vm_setpoint() (mp.dme_gen3p method), 188
 apply_vm_setpoints() (mp.dme_legacy_dcline method), 209
 aux_data (mp.math_model attribute), 125
 aux_data_va_vm() (mp.form_ac method), 85
 aux_data_va_vm() (mp.form_acc method), 87
 aux_data_va_vm() (mp.form_acp method), 91

B

B (mp.form_dc attribute), 92
 b_fr (mp.dme_branch attribute), 50
 b_to (mp.dme_branch attribute), 50
 base_kva (mp.data_model attribute), 31
 base_mva (mp.data_model attribute), 31
 bs (mp.dme_shunt attribute), 59
 build() (mp.data_model method), 31

build() (mp.dm_converter method), 62
 build() (mp.math_model method), 125
 build() (mp.math_model_opf_acci_legacy method), 136
 build() (mp.math_model_opf_accs_legacy method), 137
 build() (mp.math_model_opf_acpi_legacy method), 139
 build() (mp.math_model_opf_acps_legacy method), 140
 build() (mp.math_model_opf_dc_legacy method), 141
 build() (mp.net_model method), 95
 build_aux_data() (mp.math_model_opf method), 135
 build_aux_data() (mp.mm_shared_pfcpf method), 141
 build_aux_data() (mp.mm_shared_pfcpf_acci method), 143
 build_aux_data() (mp.mm_shared_pfcpf_acpi method), 144
 build_aux_data() (mp.mm_shared_pfcpf_acps method), 144
 build_aux_data() (mp.mm_shared_pfcpf_dc method), 144
 build_aux_data_i() (mp.mm_shared_pfcpf_ac_i method), 142
 build_base_aux_data() (mp.math_model method), 126
 build_cost_params() (mp.dme_gen_opf method), 56
 build_cost_params() (mp.dme_legacy_dcline_opf method), 210
 build_cost_params() (mp.mme_gen_opf_ac method), 156
 build_cost_params() (mp.mme_gen_opf_dc method), 156
 build_cost_params() (mp.mme_legacy_dcline_opf method), 214
 build_legacy() (mp.mm_shared_opf_legacy method), 145
 build_params() (mp.data_model method), 32
 build_params() (mp.dm_element method), 44
 build_params() (mp.dme_branch method), 50
 build_params() (mp.dme_bus method), 53
 build_params() (mp.dme_bus3p method), 187
 build_params() (mp.dme_buslink method), 195
 build_params() (mp.dme_gen method), 55
 build_params() (mp.dme_gen3p method), 188
 build_params() (mp.dme_legacy_dcline method), 209
 build_params() (mp.dme_line3p method), 192
 build_params() (mp.dme_load method), 58
 build_params() (mp.dme_load3p method), 190
 build_params() (mp.dme_reserve_gen method), 179
 build_params() (mp.dme_reserve_zone method), 181
 build_params() (mp.dme_shunt method), 60
 build_params() (mp.dme_xfmr3p method), 193
 build_params() (mp.net_model method), 96
 build_params() (mp.net_model_ac method), 104
 build_params() (mp.net_model_dc method), 110
 build_params() (mp.nm_element method), 114
 build_params() (mp.nme_branch_ac method), 117
 build_params() (mp.nme_branch_dc method), 118
 build_params() (mp.nme_buslink method), 200
 build_params() (mp.nme_gen3p method), 197
 build_params() (mp.nme_gen_ac method), 120
 build_params() (mp.nme_gen_dc method), 121
 build_params() (mp.nme_legacy_dcline_ac method), 211
 build_params() (mp.nme_legacy_dcline_dc method), 212
 build_params() (mp.nme_line3p method), 199
 build_params() (mp.nme_load3p method), 198
 build_params() (mp.nme_load_ac method), 122
 build_params() (mp.nme_load_dc method), 122
 build_params() (mp.nme_shunt_ac method), 123
 build_params() (mp.nme_shunt_dc method), 124
 build_params() (mp.nme_xfmr3p method), 199
 bus (mp.dmce_load3p_mpc2 attribute), 184
 bus (mp.dmce_load_mpc2 attribute), 73
 bus (mp.dmce_shunt_mpc2 attribute), 74
 bus (mp.dme_buslink attribute), 194
 bus (mp.dme_gen attribute), 54
 bus (mp.dme_gen3p attribute), 188
 bus (mp.dme_load attribute), 57
 bus (mp.dme_load3p attribute), 189
 bus (mp.dme_shunt attribute), 59
 bus3p (mp.dme_buslink attribute), 194
 bus_name_export() (mp.dmce_bus_mpc2 method), 72

bus_name_import() (*mp.dmce_bus_mpc2* method), 72
 bus_on (*mp.dme_gen* attribute), 54
 bus_on (*mp.dme_gen3p* attribute), 188
 bus_status_import() (*mp.dmce_bus3p_mpc2* method), 184
 bus_status_import() (*mp.dmce_bus_mpc2* method), 72
 bustypes() (*built-in function*), 332

C

C (*mp.nm_element* attribute), 111
 cache (*mp.sm_legacy_cost* attribute), 173
 calc_branch_angle() (*built-in function*), 333
 calc_v_i_sum() (*built-in function*), 262
 calc_v_pq_sum() (*built-in function*), 263
 calc_v_y_sum() (*built-in function*), 264
 callback_vlim() (*mp.math_model_cpf_acp* method), 133
 case_info() (*built-in function*), 333
 caseformat() (*built-in function*), 239
 cdf2mpc() (*built-in function*), 242
 compare_case() (*built-in function*), 333
 connected_components() (*built-in function*), 357
 convert_x_m2n() (*mp.math_model_opf_acc* method), 135
 convert_x_m2n() (*mp.math_model_opf_acp* method), 138
 convert_x_m2n() (*mp.math_model_opf_dc* method), 140
 convert_x_m2n() (*mp.mm_shared_pfcpf_acc* method), 142
 convert_x_m2n() (*mp.mm_shared_pfcpf_acp* method), 143
 convert_x_m2n() (*mp.mm_shared_pfcpf_dc* method), 145
 copy() (*mp.data_model* method), 31
 copy() (*mp.dm_converter* method), 62
 copy() (*mp.dm_element* method), 43
 copy() (*mp.mapped_array* method), 170
 cost (*mp.mme_gen_opf* attribute), 155
 cost (*mp.mme_legacy_dcline_opf* attribute), 213
 cost (*opf_model* attribute), 224
 cost_table (*class in mp*), 164
 cost_table() (*mp.cost_table* method), 165
 cost_table2gencost() (*mp.dmce_gen_mpc2* static method), 73
 cost_table_utils (*class in mp*), 167
 count() (*mp.data_model* method), 32
 count() (*mp.dm_element* method), 43
 count() (*mp.nm_element* method), 113
 cpf_corrector() (*built-in function*), 272
 cpf_current_mpc() (*built-in function*), 273
 cpf_default_callback() (*built-in function*), 273
 cpf_detect_events() (*built-in function*), 275
 cpf_flim_event() (*built-in function*), 276
 cpf_flim_event_cb() (*built-in function*), 276
 cpf_nose_event() (*built-in function*), 277
 cpf_nose_event_cb() (*built-in function*), 277
 cpf_p() (*built-in function*), 277
 cpf_p_jac() (*built-in function*), 278
 cpf_plim_event() (*built-in function*), 279
 cpf_plim_event_cb() (*built-in function*), 279
 cpf_predictor() (*built-in function*), 279
 cpf_qlim_event() (*built-in function*), 280
 cpf_qlim_event_cb() (*built-in function*), 280
 cpf_register_callback() (*built-in function*), 281
 cpf_register_event() (*built-in function*), 282
 cpf_tangent() (*built-in function*), 282
 cpf_target_lam_event() (*built-in function*), 283
 cpf_target_lam_event_cb() (*built-in function*), 283
 cpf_vlim_event() (*built-in function*), 284
 cpf_vlim_event_cb() (*built-in function*), 284
 create_line_construction_table() (*mp.dmce_line3p_mpc2* method), 185
 create_line_construction_table() (*mp.dme_line3p* method), 191
 ctol (*mp.dme_shared_opf* attribute), 61

cxn_idx_prop() (*mp.dm_element* method), 41
 cxn_idx_prop() (*mp.dme_branch* method), 50
 cxn_idx_prop() (*mp.dme_buslink* method), 195
 cxn_idx_prop() (*mp.dme_gen* method), 55
 cxn_idx_prop() (*mp.dme_gen3p* method), 188
 cxn_idx_prop() (*mp.dme_legacy_dcline* method), 209
 cxn_idx_prop() (*mp.dme_line3p* method), 191
 cxn_idx_prop() (*mp.dme_load* method), 58
 cxn_idx_prop() (*mp.dme_load3p* method), 190
 cxn_idx_prop() (*mp.dme_shunt* method), 60
 cxn_idx_prop() (*mp.dme_xfmr3p* method), 193
 cxn_type() (*mp.dm_element* method), 41
 cxn_type() (*mp.dme_branch* method), 50
 cxn_type() (*mp.dme_buslink* method), 195
 cxn_type() (*mp.dme_gen* method), 55
 cxn_type() (*mp.dme_gen3p* method), 188
 cxn_type() (*mp.dme_legacy_dcline* method), 209
 cxn_type() (*mp.dme_line3p* method), 191
 cxn_type() (*mp.dme_load* method), 58
 cxn_type() (*mp.dme_load3p* method), 190
 cxn_type() (*mp.dme_shunt* method), 60
 cxn_type() (*mp.dme_xfmr3p* method), 193
 cxn_type_prop() (*mp.dm_element* method), 41

D

D (*mp.nm_element* attribute), 112
 d2Abr_dV2() (*built-in function*), 322
 d2Ibr_dV2() (*built-in function*), 320
 d2Imis_dV2() (*built-in function*), 323
 d2Imis_dVdSg() (*built-in function*), 324
 d2Sbr_dV2() (*built-in function*), 321
 d2Sbus_dV2() (*built-in function*), 326
 dAbr_dV() (*built-in function*), 315
 data_exists() (*mp.dmc_element* method), 67
 data_field() (*mp.dmc_element* method), 66
 data_field() (*mp.dmce_branch_mpc2* method), 72
 data_field() (*mp.dmce_bus3p_mpc2* method), 184
 data_field() (*mp.dmce_bus_mpc2* method), 72
 data_field() (*mp.dmce_buslink_mpc2* method), 186
 data_field() (*mp.dmce_gen3p_mpc2* method), 184
 data_field() (*mp.dmce_gen_mpc2* method), 73
 data_field() (*mp.dmce_legacy_dcline_mpc2* method), 207
 data_field() (*mp.dmce_line3p_mpc2* method), 185
 data_field() (*mp.dmce_load3p_mpc2* method), 184
 data_field() (*mp.dmce_load_mpc2* method), 73
 data_field() (*mp.dmce_reserve_gen_mpc2* method), 178
 data_field() (*mp.dmce_reserve_zone_mpc2* method), 178
 data_field() (*mp.dmce_shunt_mpc2* method), 74
 data_field() (*mp.dmce_xfmr3p_mpc2* method), 185
 data_model (*class in mp*), 30
 data_model() (*mp.data_model* method), 31
 data_model_build() (*mp.task* method), 16
 data_model_build() (*mp.task_cpf* method), 23
 data_model_build_post() (*mp.task* method), 17
 data_model_build_post() (*mp.task_opf* method), 24
 data_model_build_post() (*mp.task_opf_legacy* method), 28
 data_model_build_pre() (*mp.task* method), 16
 data_model_class() (*mp.extension* method), 175
 data_model_class() (*mp.task* method), 15
 data_model_class_default() (*mp.task* method), 15
 data_model_class_default() (*mp.task_cpf* method), 23
 data_model_class_default() (*mp.task_opf* method), 24
 data_model_cpf (*class in mp*), 37
 data_model_cpf() (*mp.data_model_cpf* method), 37
 data_model_create() (*mp.task* method), 16
 data_model_element() (*mp.dmc_element* method), 66
 data_model_element() (*mp.mm_element* method), 147

- data_model_element() (*mp.nm_element* method), 112
 data_model_opf (*class in mp*), 37
 data_model_opf() (*mp.data_model_opf* method), 37
 data_model_update() (*mp.mm_element* method), 148
 data_model_update_off() (*mp.mm_element* method), 148
 data_model_update_on() (*mp.mm_element* method), 148
 data_model_update_on() (*mp.mme_branch_opf_ac* method), 150
 data_model_update_on() (*mp.mme_branch_opf_dc* method), 152
 data_model_update_on() (*mp.mme_branch_pf_ac* method), 149
 data_model_update_on() (*mp.mme_branch_pf_dc* method), 150
 data_model_update_on() (*mp.mme_bus3p* method), 200
 data_model_update_on() (*mp.mme_bus_opf_acc* method), 153
 data_model_update_on() (*mp.mme_bus_opf_acp* method), 154
 data_model_update_on() (*mp.mme_bus_opf_dc* method), 154
 data_model_update_on() (*mp.mme_bus_pf_ac* method), 152
 data_model_update_on() (*mp.mme_bus_pf_dc* method), 152
 data_model_update_on() (*mp.mme_gen3p* method), 201
 data_model_update_on() (*mp.mme_gen_opf_ac* method), 156
 data_model_update_on() (*mp.mme_gen_opf_dc* method), 156
 data_model_update_on() (*mp.mme_gen_pf_ac* method), 154
 data_model_update_on() (*mp.mme_gen_pf_dc* method), 155
 data_model_update_on() (*mp.mme_legacy_dcline_opf_ac* method), 214
 data_model_update_on() (*mp.mme_legacy_dcline_opf_dc* method), 214
 data_model_update_on() (*mp.mme_legacy_dcline_pf_ac* method), 213
 data_model_update_on() (*mp.mme_legacy_dcline_pf_dc* method), 213
 data_model_update_on() (*mp.mme_line3p* method), 201
 data_model_update_on() (*mp.mme_load_cpf* method), 158
 data_model_update_on() (*mp.mme_load_pf_ac* method), 157
 data_model_update_on() (*mp.mme_load_pf_dc* method), 157
 data_model_update_on() (*mp.mme_reserve_gen* method), 181
 data_model_update_on() (*mp.mme_reserve_zone* method), 181
 data_model_update_on() (*mp.mme_shunt_cpf* method), 159
 data_model_update_on() (*mp.mme_shunt_pf_ac* method), 158
 data_model_update_on() (*mp.mme_shunt_pf_dc* method), 159
 data_model_update_on() (*mp.mme_xfmr3p* method), 202
 data_subs() (*mp.dmc_element* method), 67
 data_subs() (*mp.dmce_reserve_gen_mpc2* method), 178
 data_subs() (*mp.dmce_reserve_zone_mpc2* method), 178
 dc (*mp.task_opf* attribute), 24
 dc (*mp.task_pf* attribute), 21
 dcline_cost_export() (*mp.dmce_legacy_dcline_mpc2* method), 207
 dcline_cost_import() (*mp.dmce_legacy_dcline_mpc2* method), 207
 dcof() (*built-in function*), 288
 dcof_solver() (*built-in function*), 289
 dcpf() (*built-in function*), 264
 def_set_types() (*mp.math_model_opf_acci_legacy* method), 136
 def_set_types() (*mp.math_model_opf_accs_legacy* method), 137
 def_set_types() (*mp.math_model_opf_acpi_legacy* method), 139
 def_set_types() (*mp.math_model_opf_acps_legacy* method), 139
 def_set_types() (*mp.math_model_opf_dc_legacy* method), 141
 def_set_types() (*mp.net_model* method), 97
 def_set_types() (*mp.net_model_ac* method), 104
 def_set_types() (*mp.net_model_acc* method), 107
 def_set_types() (*mp.net_model_acp* method), 108
 def_set_types() (*mp.net_model_dc* method), 110
 def_set_types() (*opf_model* method), 225
 default_export_data_nrows() (*mp.dmc_element* method), 71
 default_export_data_table() (*mp.dmc_element* method), 71
 default_export_data_table() (*mp.dmce_branch_mpc2* method), 72
 default_export_data_table() (*mp.dmce_bus_mpc2* method), 72
 default_export_data_table() (*mp.dmce_gen_mpc2* method), 73
 default_export_data_table() (*mp.dmce_legacy_dcline_mpc2* method), 207
 define_constants() (*built-in function*), 334
 delete_elements() (*mp.mapped_array* method), 171
 dIbr_dV() (*built-in function*), 312
 diff_poly_fcn() (*mp.cost_table* static method), 167
 dImis_dV() (*built-in function*), 316
 disp_load_constant_pf_constraint() (*mp.mme_gen_opf_ac* method), 156
 display() (*mp.data_model* method), 33
 display() (*mp.dm_converter* method), 63
 display() (*mp.dm_element* method), 45
 display() (*mp.mapped_array* method), 172
 display() (*mp.math_model* method), 126
 display() (*mp.net_model* method), 97
 display() (*mp.nm_element* method), 117
 display() (*mp_table* method), 162
 display() (*opf_model* method), 225
 dm (*mp.task* attribute), 11
 dm_converter (*class in mp*), 62
 dm_converter_build() (*mp.task* method), 15
 dm_converter_class() (*mp.extension* method), 175
 dm_converter_class() (*mp.task* method), 14
 dm_converter_class() (*mp.task_cpf* method), 23
 dm_converter_class_mpc2_default() (*mp.task* method), 14
 dm_converter_class_mpc2_default() (*mp.task_opf_legacy* method), 28
 dm_converter_create() (*mp.task* method), 15
 dm_converter_element() (*mp.dm_element* method), 43
 dm_converter_mpc2 (*class in mp*), 64
 dm_converter_mpc2() (*mp.dm_converter_mpc2* method), 64
 dm_converter_mpc2_legacy (*class in mp*), 65
 dm_element (*class in mp*), 38
 dm_element_classes() (*mp.extension* method), 176
 dm_element_classes() (*mp.xt_3p* method), 183
 dm_element_classes() (*mp.xt_legacy_dcline* method), 206
 dm_element_classes() (*mp.xt_reserves* method), 177
 dmc (*mp.task* attribute), 11
 dmc_element (*class in mp*), 65
 dmc_element_classes() (*mp.extension* method), 175
 dmc_element_classes() (*mp.xt_3p* method), 182
 dmc_element_classes() (*mp.xt_legacy_dcline* method), 206
 dmc_element_classes() (*mp.xt_reserves* method), 177
 dmce_branch_mpc2 (*class in mp*), 72
 dmce_bus3p_mpc2 (*class in mp*), 184
 dmce_bus_mpc2 (*class in mp*), 72
 dmce_buslink_mpc2 (*class in mp*), 186
 dmce_gen3p_mpc2 (*class in mp*), 184
 dmce_gen_mpc2 (*class in mp*), 72
 dmce_legacy_dcline_mpc2 (*class in mp*), 207
 dmce_line3p_mpc2 (*class in mp*), 185
 dmce_load3p_mpc2 (*class in mp*), 184
 dmce_load_mpc2 (*class in mp*), 73
 dmce_reserve_gen_mpc2 (*class in mp*), 178
 dmce_reserve_zone_mpc2 (*class in mp*), 178
 dmce_shunt_mpc2 (*class in mp*), 74
 dmce_xfmr3p_mpc2 (*class in mp*), 185
 dme_branch (*class in mp*), 49
 dme_branch_opf (*class in mp*), 51
 dme_bus (*class in mp*), 52
 dme_bus3p (*class in mp*), 186
 dme_bus3p_opf (*class in mp*), 195
 dme_bus_opf (*class in mp*), 53
 dme_buslink (*class in mp*), 194
 dme_buslink_opf (*class in mp*), 196
 dme_gen (*class in mp*), 54
 dme_gen3p (*class in mp*), 187

dme_gen3p_opf (class in mp), 195
 dme_gen_opf (class in mp), 56
 dme_legacy_dcline (class in mp), 208
 dme_legacy_dcline_opf (class in mp), 210
 dme_line3p (class in mp), 190
 dme_line3p_opf (class in mp), 196
 dme_load (class in mp), 57
 dme_load3p (class in mp), 189
 dme_load3p_opf (class in mp), 195
 dme_load_cpf (class in mp), 58
 dme_load_opf (class in mp), 58
 dme_reserve_gen (class in mp), 179
 dme_reserve_zone (class in mp), 180
 dme_shared_opf (class in mp), 61
 dme_shunt (class in mp), 59
 dme_shunt_cpf (class in mp), 59
 dme_shunt_opf (class in mp), 60
 dme_xfmr3p (class in mp), 192
 dme_xfmr3p_opf (class in mp), 196
 dSbr_dV() (built-in function), 313
 dSbus_dV() (built-in function), 318

E

e2i_data() (built-in function), 257
 e2i_field() (built-in function), 258
 element_classes (mp.element_container attribute), 168
 element_container (class in mp), 168
 elements (mp.element_container attribute), 168
 end() (mp_table method), 161
 enforce_q_lims() (mp.task_pf method), 22
 ensure_ref_node() (mp.net_model method), 102
 et (mp.task attribute), 11
 eval_legacy_cost() (mp.mm_shared_opf_legacy method), 145
 eval_legacy_cost() (opf_model method), 225
 eval_poly_fcn() (mp.cost_table static method), 166
 event_vlim() (mp.math_model_cpf_acp method), 133
 expand_z_warmstart() (mp.math_model_cpf_acps method), 134
 export() (mp.dm_converter method), 63
 export() (mp.dmc_element method), 70
 export_col() (mp.dmc_element method), 71
 export_table_values() (mp.dmc_element method), 70
 export_vars() (mp.dm_element method), 42
 export_vars() (mp.dme_branch method), 50
 export_vars() (mp.dme_branch_opf method), 51
 export_vars() (mp.dme_bus method), 52
 export_vars() (mp.dme_bus_opf method), 53
 export_vars() (mp.dme_gen method), 55
 export_vars() (mp.dme_gen_opf method), 56
 export_vars() (mp.dme_legacy_dcline method), 209
 export_vars() (mp.dme_legacy_dcline_opf method), 210
 export_vars() (mp.dme_load_cpf method), 58
 export_vars() (mp.dme_reserve_gen method), 179
 export_vars() (mp.dme_reserve_zone method), 180
 export_vars() (mp.dme_shunt_cpf method), 59
 export_vars_offline_val() (mp.dm_element method), 42
 export_vars_offline_val() (mp.dme_branch method), 50
 export_vars_offline_val() (mp.dme_branch_opf method), 51
 export_vars_offline_val() (mp.dme_bus method), 52
 export_vars_offline_val() (mp.dme_bus_opf method), 53
 export_vars_offline_val() (mp.dme_gen method), 55
 export_vars_offline_val() (mp.dme_gen_opf method), 56
 export_vars_offline_val() (mp.dme_legacy_dcline method), 209
 export_vars_offline_val() (mp.dme_legacy_dcline_opf method), 210
 export_vars_offline_val() (mp.dme_reserve_gen method), 179
 export_vars_offline_val() (mp.dme_reserve_zone method), 180
 ext2int() (built-in function), 256

extension (class in mp), 174
 extract_islands() (built-in function), 335
 extract_named_args() (mp_table static method), 162

F

fbus (mp.dme_branch attribute), 49
 fbus (mp.dme_legacy_dcline attribute), 208
 fbus (mp.dme_line3p attribute), 191
 fbus (mp.dme_xfmr3p attribute), 193
 fbus_on (mp.dme_legacy_dcline attribute), 208
 fd_jac_approx() (mp.math_model_pf_acps method), 131
 fdpf() (built-in function), 265
 fdpf_B_matrix_models() (mp.math_model_pf_acps method), 131
 feval_w_path() (built-in function), 336
 find_bridges() (built-in function), 337
 find_form_class() (mp.form method), 76
 find_islands() (built-in function), 337
 fixed_q_idx (mp.task_pf attribute), 22
 fixed_q_qty (mp.task_pf attribute), 22
 fmincopf() (built-in function), 288
 form (class in mp), 74
 form_ac (class in mp), 76
 form_acc (class in mp), 85
 form_acp (class in mp), 89
 form_dc (class in mp), 91
 form_name() (mp.form method), 75
 form_name() (mp.form_acc method), 86
 form_name() (mp.form_acp method), 89
 form_name() (mp.form_dc method), 92
 form_name() (mp.math_model method), 125
 form_name() (mp.math_model_cpf_acci method), 133
 form_name() (mp.math_model_cpf_accs method), 133
 form_name() (mp.math_model_cpf_acpi method), 134
 form_name() (mp.math_model_cpf_acps method), 134
 form_name() (mp.math_model_opf_acci method), 136
 form_name() (mp.math_model_opf_accs method), 137
 form_name() (mp.math_model_opf_acpi method), 138
 form_name() (mp.math_model_opf_acps method), 139
 form_name() (mp.math_model_opf_dc method), 140
 form_name() (mp.math_model_pf_acci method), 130
 form_name() (mp.math_model_pf_accs method), 130
 form_name() (mp.math_model_pf_acpi method), 131
 form_name() (mp.math_model_pf_acps method), 131
 form_name() (mp.math_model_pf_dc method), 132
 form_tag() (mp.form method), 75
 form_tag() (mp.form_acc method), 86
 form_tag() (mp.form_acp method), 90
 form_tag() (mp.form_dc method), 92
 form_tag() (mp.math_model method), 125
 form_tag() (mp.math_model_cpf_acci method), 132
 form_tag() (mp.math_model_cpf_accs method), 133
 form_tag() (mp.math_model_cpf_acpi method), 134
 form_tag() (mp.math_model_cpf_acps method), 134
 form_tag() (mp.math_model_opf_acci method), 136
 form_tag() (mp.math_model_opf_accs method), 137
 form_tag() (mp.math_model_opf_acpi method), 138
 form_tag() (mp.math_model_opf_acps method), 139
 form_tag() (mp.math_model_opf_dc method), 140
 form_tag() (mp.math_model_pf_acci method), 130
 form_tag() (mp.math_model_pf_accs method), 130
 form_tag() (mp.math_model_pf_acpi method), 131
 form_tag() (mp.math_model_pf_acps method), 131
 form_tag() (mp.math_model_pf_dc method), 132
 format_tag() (mp.dm_converter method), 62
 format_tag() (mp.dm_converter_mpc2 method), 64
 freq (mp.dme_line3p attribute), 191

G

g_fr (*mp.dme_branch* attribute), 49
 g_to (*mp.dme_branch* attribute), 49
 gausspf() (built-in function), 265
 gen (*mp.dme_reserve_gen* attribute), 179
 gen_cost_export() (*mp.dmce_gen_mpc2* method), 73
 gen_cost_import() (*mp.dmce_gen_mpc2* method), 73
 gencost2cost_table() (*mp.dmce_gen_mpc2* static method), 73
 genfuels() (built-in function), 337
 gentypes() (built-in function), 338
 get_export_size() (*mp.dmc_element* method), 68
 get_export_size() (*mp.dmce_load_mpc2* method), 73
 get_export_size() (*mp.dmce_reserve_gen_mpc2* method), 178
 get_export_size() (*mp.dmce_shunt_mpc2* method), 74
 get_export_spec() (*mp.dmc_element* method), 67
 get_import_size() (*mp.dmc_element* method), 68
 get_import_size() (*mp.dmce_load_mpc2* method), 73
 get_import_size() (*mp.dmce_reserve_gen_mpc2* method), 178
 get_import_size() (*mp.dmce_shunt_mpc2* method), 74
 get_import_spec() (*mp.dmc_element* method), 67
 get_input_table_values() (*mp.dmc_element* method), 69
 get_losses() (built-in function), 340
 get_mpc() (*mp.mm_shared_opf_legacy* method), 145
 get_mpc() (*opf_model* method), 225
 get_node_idx() (*mp.net_model* method), 101
 get_nv() (*mp.nm_element* method), 114
 get_params() (*mp.form* method), 76
 get_port_idx() (*mp.net_model* method), 101
 get_reorder() (built-in function), 262
 get_state_idx() (*mp.net_model* method), 101
 get_table() (*mp_table_subclass* method), 164
 get_va() (*mp.net_model_ac* method), 106
 gs (*mp.dme_shunt* attribute), 59
 gs_x_update() (*mp.math_model_pf_acps* method), 131

H

has_name() (*mp.mapped_array* method), 171
 has_pq_cap() (*mp.mme_gen_opf_ac* method), 156
 hasPQcap() (built-in function), 341
 have_cost() (*mp.dme_gen* method), 55
 have_cost() (*mp.dme_gen_opf* method), 56
 have_cost() (*mp.dme_legacy_dcline* method), 209
 have_cost() (*mp.dme_legacy_dcline_opf* method), 210
 have_feature_e4st() (built-in function), 354
 have_feature_minopf() (built-in function), 355
 have_feature_most() (built-in function), 355
 have_feature_mpc_core() (built-in function), 355
 have_feature_pdipmopf() (built-in function), 355
 have_feature_regexp_split() (built-in function), 356
 have_feature_scpdipmopf() (built-in function), 356
 have_feature_sdp_pf() (built-in function), 356
 have_feature_smartmarket() (built-in function), 356
 have_feature_syngrid() (built-in function), 357
 have_feature_table() (built-in function), 357
 have_feature_tralmopf() (built-in function), 357
 horzcat() (*mp_table* method), 162

I

i (*mp.form_ac* attribute), 78
 i2e_data() (built-in function), 260
 i2e_field() (built-in function), 261
 i2on (*mp.dm_element* attribute), 40
 i_dm (*mp.task* attribute), 11
 i_mm (*mp.task* attribute), 11
 i_nm (*mp.task* attribute), 11
 ID() (*mp.dm_element* method), 44

ID2i (*mp.dm_element* attribute), 40
 idx_brch() (built-in function), 341
 idx_bus() (built-in function), 342
 idx_cost() (built-in function), 343
 idx_ct() (built-in function), 344
 idx_dcline() (built-in function), 346
 idx_gen() (built-in function), 347
 import() (*mp.dm_converter* method), 62
 import() (*mp.dm_converter_mpc2* method), 64
 import() (*mp.dmc_element* method), 68
 import() (*mp.dmce_line3p_mpc2* method), 185
 import() (*mp.dmce_reserve_gen_mpc2* method), 178
 import_col() (*mp.dmc_element* method), 69
 import_cost() (*mp.dmce_reserve_gen_mpc2* method), 178
 import_qty() (*mp.dmce_reserve_gen_mpc2* method), 178
 import_ramp() (*mp.dmce_reserve_gen_mpc2* method), 178
 import_req() (*mp.dmce_reserve_zone_mpc2* method), 178
 import_table_values() (*mp.dmc_element* method), 69
 import_zones() (*mp.dmce_reserve_zone_mpc2* method), 178
 incidence_matrix() (*mp.nm_element* method), 115
 init_export() (*mp.dm_converter* method), 63
 init_export() (*mp.dm_converter_mpc2* method), 64
 init_export_data() (*mp.dmc_element* method), 70
 init_export_data() (*mp.dmce_bus_mpc2* method), 72
 init_set_types() (*mp.math_model_opf_acci_legacy* method), 136
 init_set_types() (*mp.math_model_opf_accs_legacy* method), 137
 init_set_types() (*mp.math_model_opf_acpi_legacy* method), 139
 init_set_types() (*mp.math_model_opf_acps_legacy* method), 140
 init_set_types() (*mp.math_model_opf_dc_legacy* method), 141
 init_set_types() (*mp.net_model* method), 97
 init_set_types() (*opf_model* method), 225
 init_set_types_legacy() (*mp.mm_shared_opf_legacy* method), 145
 init_status() (*mp.dm_element* method), 44
 init_status() (*mp.dme_bus* method), 52
 init_status() (*mp.dme_bus3p* method), 187
 initial_voltage_angle() (*mp.net_model_acc* method), 108
 initial_voltage_angle() (*mp.net_model_acp* method), 109
 initialize() (*mp.data_model* method), 32
 initialize() (*mp.dm_element* method), 43
 initialize() (*mp.dme_branch* method), 50
 initialize() (*mp.dme_buslink* method), 195
 initialize() (*mp.dme_gen* method), 55
 initialize() (*mp.dme_gen3p* method), 188
 initialize() (*mp.dme_legacy_dcline* method), 209
 initialize() (*mp.dme_line3p* method), 192
 initialize() (*mp.dme_load* method), 58
 initialize() (*mp.dme_load3p* method), 190
 initialize() (*mp.dme_shunt* method), 60
 initialize() (*mp.dme_xfmr3p* method), 193
 lnln (*mp.form_ac* attribute), 78
 lnln_hess (*mp.form_ac* attribute), 78
 install_matpower() (built-in function), 2
 int2ext() (built-in function), 259
 interior_va() (*mp.math_model_opf* method), 135
 interior_va() (*mp.math_model_opf_acc* method), 135
 interior_vm() (*mp.mme_bus_opf_ac* method), 153
 interior_x0() (*mp.math_model_opf* method), 135
 interior_x0() (*mp.mme_bus3p_opf_acc* method), 203
 interior_x0() (*mp.mme_bus3p_opf_acp* method), 203
 interior_x0() (*mp.mme_bus_opf_acc* method), 153
 interior_x0() (*mp.mme_bus_opf_acp* method), 154
 interior_x0() (*mp.mme_bus_opf_dc* method), 154
 interior_x0() (*mp.mme_buslink_opf* method), 205
 interior_x0() (*mp.mme_gen3p_opf* method), 204
 interior_x0() (*mp.mme_gen_opf* method), 155
 interior_x0() (*mp.mme_legacy_dcline_opf* method), 214
 interior_x0() (*mp.mme_line3p_opf* method), 204

interior_x0() (*mp.mme_xfmr3p_opf* method), 204
 is_valid() (*mp.NODE_TYPE* static method), 172
 isempty() (*mp_table* method), 160
 isload() (*built-in* function), 348
 isload() (*mp.dme_gen* method), 55
 istable() (*mp_table* method), 160
 iterations (*mp.task_pf* attribute), 21

K

K (*mp.form_dc* attribute), 92

L

L (*mp.form_ac* attribute), 78
 label() (*mp.dm_element* method), 41
 label() (*mp.dme_branch* method), 50
 label() (*mp.dme_bus* method), 52
 label() (*mp.dme_bus3p* method), 187
 label() (*mp.dme_buslink* method), 194
 label() (*mp.dme_gen* method), 55
 label() (*mp.dme_gen3p* method), 188
 label() (*mp.dme_legacy_dcline* method), 209
 label() (*mp.dme_line3p* method), 191
 label() (*mp.dme_load* method), 57
 label() (*mp.dme_load3p* method), 189
 label() (*mp.dme_reserve_gen* method), 179
 label() (*mp.dme_reserve_zone* method), 180
 label() (*mp.dme_shunt* method), 60
 label() (*mp.dme_xfmr3p* method), 193
 labels() (*mp.dm_element* method), 41
 labels() (*mp.dme_branch* method), 50
 labels() (*mp.dme_bus* method), 52
 labels() (*mp.dme_bus3p* method), 187
 labels() (*mp.dme_buslink* method), 195
 labels() (*mp.dme_gen* method), 55
 labels() (*mp.dme_gen3p* method), 188
 labels() (*mp.dme_legacy_dcline* method), 209
 labels() (*mp.dme_line3p* method), 191
 labels() (*mp.dme_load* method), 58
 labels() (*mp.dme_load3p* method), 189
 labels() (*mp.dme_reserve_gen* method), 179
 labels() (*mp.dme_reserve_zone* method), 180
 labels() (*mp.dme_shunt* method), 60
 labels() (*mp.dme_xfmr3p* method), 193
 lc (*mp.dme_line3p* attribute), 191
 lc_tab (*mp.dme_line3p* attribute), 191
 lc_table_var_names() (*mp.dme_line3p* method), 191
 lc_y_idx (*mp.dme_line3p* attribute), 191
 legacy_post_run() (*mp.task_cpf_legacy* method), 27
 legacy_post_run() (*mp.task_opf_legacy* method), 28
 legacy_post_run() (*mp.task_pf_legacy* method), 26
 legacy_user_mod_inputs() (*mp.dm_converter_mpc2_legacy* method), 65
 legacy_user_nln_constraints() (*mp.dm_converter_mpc2_legacy* method), 65
 legacy_user_var_names() (*mp.math_model_opf_acci_legacy* method), 136
 legacy_user_var_names() (*mp.math_model_opf_accs_legacy* method), 137
 legacy_user_var_names() (*mp.math_model_opf_acpi_legacy* method), 139
 legacy_user_var_names() (*mp.math_model_opf_acps_legacy* method), 140
 legacy_user_var_names() (*mp.math_model_opf_dc_legacy* method), 141
 len (*mp.dme_line3p* attribute), 191
 length() (*mp.mapped_array* method), 170
 load2disp() (*built-in* function), 348

load_dm() (*in module mp*), 7
 load_dm() (*mp.task* method), 11
 loadcase() (*built-in* function), 243
 loadshed() (*built-in* function), 349
 loss0 (*mp.dme_legacy_dcline* attribute), 208
 loss1 (*mp.dme_legacy_dcline* attribute), 208
 loss_tol (*mp.dme_branch* attribute), 50

M

M (*mp.form_ac* attribute), 78
 main_table_var_names() (*mp.dm_element* method), 42
 main_table_var_names() (*mp.dme_branch* method), 50
 main_table_var_names() (*mp.dme_branch_opf* method), 51
 main_table_var_names() (*mp.dme_bus* method), 52
 main_table_var_names() (*mp.dme_bus3p* method), 187
 main_table_var_names() (*mp.dme_bus_opf* method), 53
 main_table_var_names() (*mp.dme_buslink* method), 195
 main_table_var_names() (*mp.dme_gen* method), 55
 main_table_var_names() (*mp.dme_gen3p* method), 188
 main_table_var_names() (*mp.dme_gen_opf* method), 56
 main_table_var_names() (*mp.dme_legacy_dcline* method), 209
 main_table_var_names() (*mp.dme_legacy_dcline_opf* method), 210
 main_table_var_names() (*mp.dme_line3p* method), 191
 main_table_var_names() (*mp.dme_load* method), 58
 main_table_var_names() (*mp.dme_load3p* method), 190
 main_table_var_names() (*mp.dme_reserve_gen* method), 179
 main_table_var_names() (*mp.dme_reserve_zone* method), 180
 main_table_var_names() (*mp.dme_shunt* method), 60
 main_table_var_names() (*mp.dme_xfmr3p* method), 193
 make_vcorr() (*built-in* function), 266
 make_zpv() (*built-in* function), 266
 makeAang() (*built-in* function), 291
 makeAppq() (*built-in* function), 291
 makeAvl() (*built-in* function), 292
 makeAy() (*built-in* function), 292
 makeB() (*built-in* function), 327
 makeBdc() (*built-in* function), 328
 makeJac() (*built-in* function), 328
 makeLODF() (*built-in* function), 329
 makePTDF() (*built-in* function), 329
 makeSbus() (*built-in* function), 330
 makeSdzip() (*built-in* function), 331
 makeYbus() (*built-in* function), 331
 mapped_array (*class in mp*), 169
 mapped_array() (*mp.mapped_array* method), 170
 margcost() (*built-in* function), 293
 math_model (*class in mp*), 124
 math_model_build() (*mp.task* method), 20
 math_model_class() (*mp.extension* method), 175
 math_model_class() (*mp.task* method), 19
 math_model_class_default() (*mp.task* method), 19
 math_model_class_default() (*mp.task_cpf* method), 23
 math_model_class_default() (*mp.task_opf* method), 25
 math_model_class_default() (*mp.task_opf_legacy* method), 28
 math_model_class_default() (*mp.task_pf* method), 22
 math_model_cpf_acc (*class in mp*), 132
 math_model_cpf_acc() (*mp.math_model_cpf_acc* method), 132
 math_model_cpf_acci (*class in mp*), 132
 math_model_cpf_accs (*class in mp*), 133
 math_model_cpf_acp (*class in mp*), 133
 math_model_cpf_acp() (*mp.math_model_cpf_acp* method), 133
 math_model_cpf_acpi (*class in mp*), 134
 math_model_cpf_acps (*class in mp*), 134
 math_model_create() (*mp.task* method), 19
 math_model_element() (*mp.nm_element* method), 113
 math_model_opf (*class in mp*), 134
 math_model_opf_ac (*class in mp*), 135

`math_model_opf_acc` (class in `mp`), 135
`math_model_opf_acc()` (`mp.math_model_opf_acc` method), 135
`math_model_opf_acci` (class in `mp`), 136
`math_model_opf_acci_legacy` (class in `mp`), 136
`math_model_opf_acci_legacy()` (`mp.math_model_opf_acci_legacy` method), 136
`math_model_opf_accs` (class in `mp`), 137
`math_model_opf_accs_legacy` (class in `mp`), 137
`math_model_opf_accs_legacy()` (`mp.math_model_opf_accs_legacy` method), 137
`math_model_opf_acp` (class in `mp`), 138
`math_model_opf_acp()` (`mp.math_model_opf_acp` method), 138
`math_model_opf_acpi` (class in `mp`), 138
`math_model_opf_acpi_legacy` (class in `mp`), 138
`math_model_opf_acpi_legacy()` (`mp.math_model_opf_acpi_legacy` method), 138
`math_model_opf_acps` (class in `mp`), 139
`math_model_opf_acps_legacy` (class in `mp`), 139
`math_model_opf_acps_legacy()` (`mp.math_model_opf_acps_legacy` method), 139
`math_model_opf_dc` (class in `mp`), 140
`math_model_opf_dc()` (`mp.math_model_opf_dc` method), 140
`math_model_opf_dc_legacy` (class in `mp`), 140
`math_model_opf_dc_legacy()` (`mp.math_model_opf_dc_legacy` method), 140
`math_model_opt()` (`mp.task` method), 20
`math_model_opt()` (`mp.task_cpf` method), 24
`math_model_pf` (class in `mp`), 129
`math_model_pf_ac` (class in `mp`), 130
`math_model_pf_ac()` (`mp.math_model_pf_ac` method), 130
`math_model_pf_acci` (class in `mp`), 130
`math_model_pf_accs` (class in `mp`), 130
`math_model_pf_acpi` (class in `mp`), 131
`math_model_pf_acps` (class in `mp`), 131
`math_model_pf_dc` (class in `mp`), 132
`math_model_pf_dc()` (`mp.math_model_pf_dc` method), 132
`max_pwl_cost()` (`mp.cost_table` method), 166
`max_pwl_cost()` (`mp.cost_table_utils` static method), 168
`max_pwl_gencost()` (`mp.dme_gen_opf` method), 56
`message` (`mp.task` attribute), 11
`miqps_matpower()` (built-in function), 327
`mm` (`mp.task` attribute), 11
`mm_element` (class in `mp`), 146
`mm_element_classes()` (`mp.extension` method), 176
`mm_element_classes()` (`mp.xt_3p` method), 183
`mm_element_classes()` (`mp.xt_legacy_dcline` method), 207
`mm_element_classes()` (`mp.xt_oval_cap_curve` method), 215
`mm_element_classes()` (`mp.xt_reserves` method), 177
`mm_opt` (`mp.task` attribute), 11
`mm_shared_opf_legacy` (class in `mp`), 145
`mm_shared_pfcpf` (class in `mp`), 141
`mm_shared_pfcpf_ac` (class in `mp`), 141
`mm_shared_pfcpf_ac_i` (class in `mp`), 142
`mm_shared_pfcpf_acc` (class in `mp`), 142
`mm_shared_pfcpf_acci` (class in `mp`), 143
`mm_shared_pfcpf_accs` (class in `mp`), 143
`mm_shared_pfcpf_acp` (class in `mp`), 143
`mm_shared_pfcpf_acpi` (class in `mp`), 144
`mm_shared_pfcpf_acps` (class in `mp`), 144
`mm_shared_pfcpf_dc` (class in `mp`), 144
`mme_branch` (class in `mp`), 149
`mme_branch_opf` (class in `mp`), 150
`mme_branch_opf_ac` (class in `mp`), 150
`mme_branch_opf_acc` (class in `mp`), 151
`mme_branch_opf_acp` (class in `mp`), 151
`mme_branch_opf_dc` (class in `mp`), 151
`mme_branch_pf_ac` (class in `mp`), 149
`mme_branch_pf_dc` (class in `mp`), 150
`mme_bus` (class in `mp`), 152
`mme_bus3p` (class in `mp`), 200
`mme_bus3p_opf_acc` (class in `mp`), 203
`mme_bus3p_opf_acp` (class in `mp`), 203
`mme_bus_opf_ac` (class in `mp`), 153
`mme_bus_opf_acc` (class in `mp`), 153
`mme_bus_opf_acp` (class in `mp`), 153
`mme_bus_opf_dc` (class in `mp`), 154
`mme_bus_pf_ac` (class in `mp`), 152
`mme_bus_pf_dc` (class in `mp`), 152
`mme_buslink` (class in `mp`), 202
`mme_buslink_opf` (class in `mp`), 205
`mme_buslink_opf_acc` (class in `mp`), 205
`mme_buslink_opf_acp` (class in `mp`), 205
`mme_buslink_pf_ac` (class in `mp`), 202
`mme_buslink_pf_acc` (class in `mp`), 202
`mme_buslink_pf_acp` (class in `mp`), 203
`mme_gen` (class in `mp`), 154
`mme_gen3p` (class in `mp`), 201
`mme_gen3p_opf` (class in `mp`), 204
`mme_gen_opf` (class in `mp`), 155
`mme_gen_opf_ac` (class in `mp`), 156
`mme_gen_opf_ac_oval` (class in `mp`), 215
`mme_gen_opf_dc` (class in `mp`), 156
`mme_gen_pf_ac` (class in `mp`), 154
`mme_gen_pf_dc` (class in `mp`), 155
`mme_legacy_dcline` (class in `mp`), 212
`mme_legacy_dcline_opf` (class in `mp`), 213
`mme_legacy_dcline_opf_ac` (class in `mp`), 214
`mme_legacy_dcline_opf_dc` (class in `mp`), 214
`mme_legacy_dcline_pf_ac` (class in `mp`), 213
`mme_legacy_dcline_pf_dc` (class in `mp`), 213
`mme_line3p` (class in `mp`), 201
`mme_line3p_opf` (class in `mp`), 204
`mme_load` (class in `mp`), 157
`mme_load_cpf` (class in `mp`), 158
`mme_load_pf_ac` (class in `mp`), 157
`mme_load_pf_dc` (class in `mp`), 157
`mme_reserve_gen` (class in `mp`), 181
`mme_reserve_zone` (class in `mp`), 181
`mme_shunt` (class in `mp`), 158
`mme_shunt_cpf` (class in `mp`), 159
`mme_shunt_pf_ac` (class in `mp`), 158
`mme_shunt_pf_dc` (class in `mp`), 159
`mme_xfmr3p` (class in `mp`), 201
`mme_xfmr3p_opf` (class in `mp`), 204
`mod_set_types_legacy()` (`mp.mm_shared_opf_legacy` method), 145
`modcost()` (built-in function), 349
`model_params()` (`mp.form` method), 75
`model_params()` (`mp.form_ac` method), 79
`model_params()` (`mp.form_dc` method), 92
`model_vvars()` (`mp.form` method), 75
`model_vvars()` (`mp.form_acc` method), 86
`model_vvars()` (`mp.form_acp` method), 90
`model_vvars()` (`mp.form_dc` method), 92
`model_zvars()` (`mp.form` method), 75
`model_zvars()` (`mp.form_ac` method), 79
`model_zvars()` (`mp.form_dc` method), 93
`modify_element_classes()` (`mp.element_container` method), 168
`mp_foo_table` (built-in class), 220
`mp_table` (built-in class), 159
`mp_table()` (`mp_table` method), 160
`mp_table_class()` (built-in function), 8
`mp_table_subclass` (built-in class), 163
`mpc` (`opf_model` attribute), 224
`mpoption()` (built-in function), 243
`mpoption_info_clp()` (built-in function), 358
`mpoption_info_cplex()` (built-in function), 358

mpoption_info_fmincon() (built-in function), 359
 mpoption_info_glpk() (built-in function), 359
 mpoption_info_gurobi() (built-in function), 360
 mpoption_info_intlinprog() (built-in function), 360
 mpoption_info_ipopt() (built-in function), 361
 mpoption_info_knitro() (built-in function), 361
 mpoption_info_linprog() (built-in function), 362
 mpoption_info_mips() (built-in function), 362
 mpoption_info_mosek() (built-in function), 363
 mpoption_info_osqp() (built-in function), 363
 mpoption_info_quadprog() (built-in function), 364
 mpoption_old() (built-in function), 364
 mpver() (built-in function), 350

N

n (mp.dm_element attribute), 40
 N (mp.form_ac attribute), 78
 name (mp.task attribute), 11
 name (mp.task_pf attribute), 21
 name() (mp.dm_element method), 40
 name() (mp.dmc_element method), 66
 name() (mp.dmce_branch_mpc2 method), 72
 name() (mp.dmce_bus3p_mpc2 method), 184
 name() (mp.dmce_bus_mpc2 method), 72
 name() (mp.dmce_buslink_mpc2 method), 186
 name() (mp.dmce_gen3p_mpc2 method), 184
 name() (mp.dmce_gen_mpc2 method), 73
 name() (mp.dmce_legacy_dcline_mpc2 method), 207
 name() (mp.dmce_line3p_mpc2 method), 185
 name() (mp.dmce_load3p_mpc2 method), 184
 name() (mp.dmce_load_mpc2 method), 73
 name() (mp.dmce_reserve_gen_mpc2 method), 178
 name() (mp.dmce_reserve_zone_mpc2 method), 178
 name() (mp.dmce_shunt_mpc2 method), 74
 name() (mp.dmce_xfmr3p_mpc2 method), 185
 name() (mp.dme_branch method), 50
 name() (mp.dme_bus method), 52
 name() (mp.dme_bus3p method), 187
 name() (mp.dme_buslink method), 194
 name() (mp.dme_gen method), 55
 name() (mp.dme_gen3p method), 188
 name() (mp.dme_legacy_dcline method), 209
 name() (mp.dme_line3p method), 191
 name() (mp.dme_load method), 57
 name() (mp.dme_load3p method), 189
 name() (mp.dme_reserve_gen method), 179
 name() (mp.dme_reserve_zone method), 180
 name() (mp.dme_shunt method), 60
 name() (mp.dme_xfmr3p method), 193
 name() (mp.mm_element method), 147
 name() (mp.mme_branch method), 149
 name() (mp.mme_bus method), 152
 name() (mp.mme_bus3p method), 200
 name() (mp.mme_buslink method), 202
 name() (mp.mme_gen method), 154
 name() (mp.mme_gen3p method), 201
 name() (mp.mme_legacy_dcline method), 212
 name() (mp.mme_line3p method), 201
 name() (mp.mme_load method), 157
 name() (mp.mme_reserve_gen method), 181
 name() (mp.mme_reserve_zone method), 181
 name() (mp.mme_shunt method), 158
 name() (mp.mme_xfmr3p method), 201
 name() (mp.net_model method), 95
 name() (mp.nm_element method), 112
 name() (mp.nme_branch method), 117
 name() (mp.nme_bus method), 119
 name() (mp.nme_bus3p method), 196
 name() (mp.nme_buslink method), 200
 name() (mp.nme_gen method), 120
 name() (mp.nme_gen3p method), 197
 name() (mp.nme_legacy_dcline method), 211
 name() (mp.nme_line3p method), 199
 name() (mp.nme_load method), 121
 name() (mp.nme_load3p method), 198
 name() (mp.nme_shunt method), 123
 name() (mp.nme_xfmr3p method), 199
 name2idx() (mp.mapped_array method), 171
 net_model (class in mp), 93
 net_model_ac (class in mp), 103
 net_model_acc (class in mp), 107
 net_model_acc() (mp.net_model_acc method), 107
 net_model_acp (class in mp), 108
 net_model_acp() (mp.net_model_acp method), 108
 net_model_dc (class in mp), 109
 net_model_dc() (mp.net_model_dc method), 109
 network_model_build() (mp.task method), 18
 network_model_build() (mp.task_cpf method), 23
 network_model_build_post() (mp.task method), 18
 network_model_build_post() (mp.task_pf method), 22
 network_model_build_pre() (mp.task method), 18
 network_model_class() (mp.extension method), 175
 network_model_class() (mp.task method), 17
 network_model_class_default() (mp.task method), 17
 network_model_class_default() (mp.task_opf method), 25
 network_model_class_default() (mp.task_pf method), 22
 network_model_create() (mp.task method), 17
 network_model_element() (mp.mm_element method), 147
 network_model_update() (mp.task method), 19
 network_model_update() (mp.task_cpf method), 23
 network_model_x_soln() (mp.math_model method), 129
 network_model_x_soln() (mp.task method), 18
 network_model_x_soln() (mp.task_cpf method), 23
 network_model_x_soln() (mp.task_pf method), 22
 newtonpf() (built-in function), 266
 newtonpf_I_cart() (built-in function), 267
 newtonpf_I_hybrid() (built-in function), 268
 newtonpf_I_polar() (built-in function), 268
 newtonpf_S_cart() (built-in function), 269
 newtonpf_S_hybrid() (built-in function), 270
 next_dm() (mp.task method), 13
 next_dm() (mp.task_pf method), 22
 next_mm() (mp.task method), 12
 next_mm() (mp.task_cpf method), 23
 next_nm() (mp.task method), 12
 nk (mp.nm_element attribute), 111
 nlpopf_solver() (built-in function), 290
 nm (mp.task attribute), 11
 nm_element (class in mp), 110
 nm_element_classes() (mp.extension method), 176
 nm_element_classes() (mp.xt_3p method), 183
 nm_element_classes() (mp.xt_legacy_dcline method), 206
 nme_branch (class in mp), 117
 nme_branch_ac (class in mp), 117
 nme_branch_acc (class in mp), 118
 nme_branch_acp (class in mp), 118
 nme_branch_dc (class in mp), 118
 nme_bus (class in mp), 119
 nme_bus3p (class in mp), 196
 nme_bus3p_acc (class in mp), 197
 nme_bus3p_acp (class in mp), 197
 nme_bus_acc (class in mp), 119
 nme_bus_acp (class in mp), 119
 nme_bus_dc (class in mp), 120
 nme_buslink (class in mp), 199

nme_buslink_acc (class in mp), 200
 nme_buslink_acp (class in mp), 200
 nme_gen (class in mp), 120
 nme_gen3p (class in mp), 197
 nme_gen3p_acc (class in mp), 198
 nme_gen3p_acp (class in mp), 198
 nme_gen_ac (class in mp), 120
 nme_gen_acc (class in mp), 121
 nme_gen_acp (class in mp), 121
 nme_gen_dc (class in mp), 121
 nme_legacy_dcline (class in mp), 211
 nme_legacy_dcline_ac (class in mp), 211
 nme_legacy_dcline_acc (class in mp), 212
 nme_legacy_dcline_acp (class in mp), 212
 nme_legacy_dcline_dc (class in mp), 212
 nme_line3p (class in mp), 199
 nme_load (class in mp), 121
 nme_load3p (class in mp), 198
 nme_load_ac (class in mp), 122
 nme_load_acc (class in mp), 122
 nme_load_acp (class in mp), 122
 nme_load_dc (class in mp), 122
 nme_shunt (class in mp), 123
 nme_shunt_ac (class in mp), 123
 nme_shunt_acc (class in mp), 123
 nme_shunt_acp (class in mp), 123
 nme_shunt_dc (class in mp), 124
 nme_xfmr3p (class in mp), 199
 nn() (mp.nm_element method), 112
 nn() (mp.nme_bus method), 119
 nn() (mp.nme_bus3p method), 196
 nodal_complex_current_balance() (mp.net_model_ac method), 105
 nodal_complex_current_balance_hess() (mp.net_model_ac method), 106
 nodal_complex_power_balance() (mp.net_model_ac method), 105
 nodal_complex_power_balance_hess() (mp.net_model_ac method), 106
 nodal_current_balance_fcn() (mp.math_model_opf_ac method), 135
 nodal_current_balance_hess() (mp.math_model_opf_ac method), 135
 nodal_power_balance_fcn() (mp.math_model_opf_ac method), 135
 nodal_power_balance_hess() (mp.math_model_opf_ac method), 135
 node (mp.net_model attribute), 95
 node_balance_equations() (mp.mm_shared_pfcpcpf_acci method), 143
 node_balance_equations() (mp.mm_shared_pfcpcpf_accs method), 143
 node_balance_equations() (mp.mm_shared_pfcpcpf_acpi method), 144
 node_balance_equations() (mp.mm_shared_pfcpcpf_acps method), 144
 node_indices() (mp.nm_element method), 115
 node_power_balance_prices() (mp.math_model_opf_acci method), 136
 node_power_balance_prices() (mp.math_model_opf_accs method), 137
 node_power_balance_prices() (mp.math_model_opf_acpi method), 138
 node_power_balance_prices() (mp.math_model_opf_acps method), 139
 NODE_TYPE (class in mp), 172
 node_types() (mp.net_model method), 101
 node_types() (mp.nm_element method), 116
 node_types() (mp.nme_bus method), 119

node_types() (mp.nme_bus3p method), 196
 NONE (mp.NODE_TYPE attribute), 172
 np() (mp.net_model method), 95
 np() (mp.nm_element method), 112
 np() (mp.nme_branch method), 117
 np() (mp.nme_buslink method), 200
 np() (mp.nme_gen method), 120
 np() (mp.nme_gen3p method), 197
 np() (mp.nme_legacy_dcline method), 211
 np() (mp.nme_line3p method), 199
 np() (mp.nme_load method), 121
 np() (mp.nme_load3p method), 198
 np() (mp.nme_shunt method), 123
 np() (mp.nme_xfmr3p method), 199
 nr (mp.dm_element attribute), 40
 nv (mp.net_model attribute), 95
 nz() (mp.net_model method), 95
 nz() (mp.nm_element method), 112
 nz() (mp.nme_buslink method), 200
 nz() (mp.nme_gen method), 120
 nz() (mp.nme_gen3p method), 197
 nz() (mp.nme_legacy_dcline method), 211

O

off (mp.dm_element attribute), 40
 on (mp.dm_element attribute), 40
 online() (mp.data_model method), 33
 opf() (built-in function), 285
 opf_args() (built-in function), 293
 opf_branch_ang_fcn() (built-in function), 295
 opf_branch_ang_hess() (built-in function), 296
 opf_branch_flow_fcn() (built-in function), 296
 opf_branch_flow_hess() (built-in function), 297
 opf_current_balance_fcn() (built-in function), 298
 opf_current_balance_hess() (built-in function), 298
 opf_execute() (built-in function), 295
 opf_gen_cost_fcn() (built-in function), 299
 opf_legacy_user_cost_fcn() (built-in function), 299
 opf_model (built-in class), 223
 opf_model() (opf_model method), 224
 opf_nle_fcn1() (built-in function), 383
 opf_nle_hess1() (built-in function), 384
 opf_power_balance_fcn() (built-in function), 300
 opf_power_balance_hess() (built-in function), 300
 opf_setup() (built-in function), 294
 opf_veq_fcn() (built-in function), 301
 opf_veq_hess() (built-in function), 301
 opf_vlim_fcn() (built-in function), 302
 opf_vlim_hess() (built-in function), 303
 opf_vref_fcn() (built-in function), 303
 opf_vref_hess() (built-in function), 304
 order_radial() (built-in function), 270
 oval_pq_capability_fcn() (mp.mme_gen_opf_ac_oval method), 215
 oval_pq_capability_hess() (mp.mme_gen_opf_ac_oval method), 216

P

p (mp.form_dc attribute), 92
 p_fr_lb (mp.dme_legacy_dcline attribute), 209
 p_fr_start (mp.dme_legacy_dcline attribute), 208
 p_fr_ub (mp.dme_legacy_dcline attribute), 209
 p_to_start (mp.dme_legacy_dcline attribute), 208
 param_ncols (mp.form_ac attribute), 78
 param_ncols (mp.form_dc attribute), 92
 parameterized() (mp.dme_load_cpf method), 58
 parameterized() (mp.dme_shunt_cpf method), 59

params_legacy_cost() (mp.mm_shared_opf_legacy method), 146
 params_legacy_cost() (opf_model method), 225
 params_var() (mp.net_model method), 100
 pd (mp.dme_load attribute), 57
 pd1 (mp.dme_load3p attribute), 189
 pd2 (mp.dme_load3p attribute), 189
 pd3 (mp.dme_load3p attribute), 189
 pd_i (mp.dme_load attribute), 57
 pd_z (mp.dme_load attribute), 57
 pf1 (mp.dme_load3p attribute), 189
 pf2 (mp.dme_load3p attribute), 189
 pf3 (mp.dme_load3p attribute), 189
 pf_va_fcn() (mp.mme_buslink_pf_acc method), 203
 pf_vm_fcn() (mp.mme_buslink_pf_acc method), 203
 pfsoln() (built-in function), 271
 pg1_start (mp.dme_buslink attribute), 194
 pg1_start (mp.dme_gen3p attribute), 188
 pg2_start (mp.dme_buslink attribute), 194
 pg2_start (mp.dme_gen3p attribute), 188
 pg3_start (mp.dme_buslink attribute), 194
 pg3_start (mp.dme_gen3p attribute), 188
 pg_lb (mp.dme_gen attribute), 55
 pg_start (mp.dme_gen attribute), 54
 pg_ub (mp.dme_gen attribute), 55
 poly2pwl() (built-in function), 350
 poly_cost_fcn() (mp.cost_table static method), 166
 poly_params() (mp.cost_table method), 165
 poly_params() (mp.cost_table_utils static method), 167
 polycost() (built-in function), 350
 port (mp.net_model attribute), 95
 port_active_power2_lim_fcn() (mp.form_ac method), 82
 port_active_power2_lim_hess() (mp.form_ac method), 84
 port_active_power_lim_fcn() (mp.form_ac method), 82
 port_active_power_lim_hess() (mp.form_ac method), 84
 port_apparent_power_lim_fcn() (mp.form_ac method), 82
 port_apparent_power_lim_hess() (mp.form_ac method), 83
 port_current_lim_fcn() (mp.form_ac method), 83
 port_current_lim_hess() (mp.form_ac method), 84
 port_inj_current() (mp.form_ac method), 79
 port_inj_current_hess() (mp.form_ac method), 80
 port_inj_current_hess_v() (mp.form_ac method), 81
 port_inj_current_hess_v() (mp.form_acc method), 86
 port_inj_current_hess_v() (mp.form_acp method), 90
 port_inj_current_hess_vz() (mp.form_ac method), 81
 port_inj_current_hess_vz() (mp.form_acc method), 86
 port_inj_current_hess_vz() (mp.form_acp method), 90
 port_inj_current_jac() (mp.form_ac method), 81
 port_inj_current_jac() (mp.form_acc method), 86
 port_inj_current_jac() (mp.form_acp method), 90
 port_inj_current_nln() (mp.nme_load_ac method), 122
 port_inj_nln() (mp.net_model_ac method), 104
 port_inj_nln_hess() (mp.net_model_ac method), 105
 port_inj_power() (mp.form_ac method), 79
 port_inj_power() (mp.form_dc method), 93
 port_inj_power_hess() (mp.form_ac method), 81
 port_inj_power_hess_v() (mp.form_ac method), 81
 port_inj_power_hess_v() (mp.form_acc method), 87
 port_inj_power_hess_v() (mp.form_acp method), 90
 port_inj_power_hess_vz() (mp.form_ac method), 82
 port_inj_power_hess_vz() (mp.form_acc method), 87
 port_inj_power_hess_vz() (mp.form_acp method), 91
 port_inj_power_jac() (mp.form_ac method), 81
 port_inj_power_jac() (mp.form_acc method), 87
 port_inj_power_jac() (mp.form_acp method), 90
 port_inj_power_nln() (mp.nme_load_ac method), 122
 port_inj_soln() (mp.net_model_ac method), 106
 port_inj_soln() (mp.net_model_dc method), 110
 pp_binding_rows_lim() (mp.dme_branch_opf method), 51
 pp_binding_rows_lim() (mp.dme_bus_opf method), 54
 pp_binding_rows_lim() (mp.dme_gen_opf method), 56
 pp_binding_rows_lim() (mp.dme_legacy_dcline_opf method), 210
 pp_binding_rows_lim() (mp.dme_reserve_gen method), 180
 pp_binding_rows_lim() (mp.dme_shared_opf method), 61
 pp_data() (mp.data_model method), 36
 pp_data() (mp.dm_element method), 46
 pp_data_cnt() (mp.dm_element method), 47
 pp_data_cnt() (mp.dme_branch method), 50
 pp_data_cnt() (mp.dme_bus method), 53
 pp_data_det() (mp.dm_element method), 48
 pp_data_ext() (mp.dm_element method), 47
 pp_data_ext() (mp.dme_bus method), 53
 pp_data_ext() (mp.dme_bus_opf method), 53
 pp_data_lim() (mp.dme_shared_opf method), 61
 pp_data_other() (mp.dme_shared_opf method), 61
 pp_data_row_det() (mp.dm_element method), 48
 pp_data_row_det() (mp.dme_branch method), 50
 pp_data_row_det() (mp.dme_bus method), 53
 pp_data_row_det() (mp.dme_bus3p method), 187
 pp_data_row_det() (mp.dme_bus_opf method), 54
 pp_data_row_det() (mp.dme_buslink method), 195
 pp_data_row_det() (mp.dme_gen method), 55
 pp_data_row_det() (mp.dme_gen3p method), 189
 pp_data_row_det() (mp.dme_legacy_dcline method), 210
 pp_data_row_det() (mp.dme_line3p method), 192
 pp_data_row_det() (mp.dme_load method), 58
 pp_data_row_det() (mp.dme_load3p method), 190
 pp_data_row_det() (mp.dme_reserve_gen method), 180
 pp_data_row_det() (mp.dme_reserve_zone method), 181
 pp_data_row_det() (mp.dme_shunt method), 60
 pp_data_row_det() (mp.dme_xfmr3p method), 194
 pp_data_row_lim() (mp.dme_branch_opf method), 51
 pp_data_row_lim() (mp.dme_bus_opf method), 54
 pp_data_row_lim() (mp.dme_gen_opf method), 56
 pp_data_row_lim() (mp.dme_legacy_dcline_opf method), 211
 pp_data_row_lim() (mp.dme_reserve_gen method), 180
 pp_data_row_lim() (mp.dme_shared_opf method), 61
 pp_data_sum() (mp.dm_element method), 47
 pp_data_sum() (mp.dme_branch method), 50
 pp_data_sum() (mp.dme_gen method), 55
 pp_data_sum() (mp.dme_gen3p method), 188
 pp_data_sum() (mp.dme_legacy_dcline method), 209
 pp_data_sum() (mp.dme_line3p method), 192
 pp_data_sum() (mp.dme_load method), 58
 pp_data_sum() (mp.dme_load3p method), 190
 pp_data_sum() (mp.dme_reserve_gen method), 179
 pp_data_sum() (mp.dme_shunt method), 60
 pp_data_sum() (mp.dme_xfmr3p method), 194
 pp_flags() (mp.data_model method), 33
 pp_flags() (mp.data_model_opf method), 37
 pp_get_footers() (mp.dm_element method), 46
 pp_get_footers_det() (mp.dm_element method), 48
 pp_get_footers_det() (mp.dme_branch method), 50
 pp_get_footers_det() (mp.dme_gen method), 55
 pp_get_footers_det() (mp.dme_load method), 58
 pp_get_footers_det() (mp.dme_reserve_gen method), 180
 pp_get_footers_det() (mp.dme_shunt method), 60
 pp_get_footers_lim() (mp.dme_shared_opf method), 61
 pp_get_footers_other() (mp.dme_shared_opf method), 61
 pp_get_headers() (mp.data_model method), 35
 pp_get_headers() (mp.dm_element method), 46
 pp_get_headers_cnt() (mp.data_model method), 35
 pp_get_headers_det() (mp.dm_element method), 48
 pp_get_headers_det() (mp.dme_branch method), 50
 pp_get_headers_det() (mp.dme_bus method), 53
 pp_get_headers_det() (mp.dme_bus3p method), 187
 pp_get_headers_det() (mp.dme_bus_opf method), 54

pp_get_headers_det() (*mp.dme_buslink* method), 195
 pp_get_headers_det() (*mp.dme_gen* method), 55
 pp_get_headers_det() (*mp.dme_gen3p* method), 189
 pp_get_headers_det() (*mp.dme_legacy_dcline* method), 210
 pp_get_headers_det() (*mp.dme_line3p* method), 192
 pp_get_headers_det() (*mp.dme_load* method), 58
 pp_get_headers_det() (*mp.dme_load3p* method), 190
 pp_get_headers_det() (*mp.dme_reserve_gen* method), 180
 pp_get_headers_det() (*mp.dme_reserve_zone* method), 181
 pp_get_headers_det() (*mp.dme_shunt* method), 60
 pp_get_headers_det() (*mp.dme_xfmr3p* method), 194
 pp_get_headers_ext() (*mp.data_model* method), 35
 pp_get_headers_lim() (*mp.dme_branch_opf* method), 51
 pp_get_headers_lim() (*mp.dme_bus_opf* method), 54
 pp_get_headers_lim() (*mp.dme_gen_opf* method), 56
 pp_get_headers_lim() (*mp.dme_legacy_dcline_opf* method), 210
 pp_get_headers_lim() (*mp.dme_reserve_gen* method), 180
 pp_get_headers_lim() (*mp.dme_shared_opf* method), 61
 pp_get_headers_other() (*mp.data_model* method), 36
 pp_get_headers_other() (*mp.data_model_opf* method), 38
 pp_get_headers_other() (*mp.dme_shared_opf* method), 61
 pp_get_title_det() (*mp.dm_element* method), 47
 pp_get_title_lim() (*mp.dme_branch_opf* method), 51
 pp_get_title_lim() (*mp.dme_shared_opf* method), 61
 pp_have_section() (*mp.data_model* method), 34
 pp_have_section() (*mp.dm_element* method), 45
 pp_have_section_cnt() (*mp.dm_element* method), 46
 pp_have_section_det() (*mp.dm_element* method), 47
 pp_have_section_det() (*mp.dme_branch* method), 50
 pp_have_section_det() (*mp.dme_bus* method), 53
 pp_have_section_det() (*mp.dme_bus3p* method), 187
 pp_have_section_det() (*mp.dme_buslink* method), 195
 pp_have_section_det() (*mp.dme_gen* method), 55
 pp_have_section_det() (*mp.dme_gen3p* method), 188
 pp_have_section_det() (*mp.dme_legacy_dcline* method), 210
 pp_have_section_det() (*mp.dme_line3p* method), 192
 pp_have_section_det() (*mp.dme_load* method), 58
 pp_have_section_det() (*mp.dme_load3p* method), 190
 pp_have_section_det() (*mp.dme_reserve_gen* method), 179
 pp_have_section_det() (*mp.dme_reserve_zone* method), 181
 pp_have_section_det() (*mp.dme_shunt* method), 60
 pp_have_section_det() (*mp.dme_xfmr3p* method), 194
 pp_have_section_ext() (*mp.dm_element* method), 47
 pp_have_section_ext() (*mp.dme_bus* method), 53
 pp_have_section_lim() (*mp.dme_branch_opf* method), 51
 pp_have_section_lim() (*mp.dme_bus_opf* method), 54
 pp_have_section_lim() (*mp.dme_gen_opf* method), 56
 pp_have_section_lim() (*mp.dme_legacy_dcline_opf* method), 210
 pp_have_section_lim() (*mp.dme_reserve_gen* method), 180
 pp_have_section_lim() (*mp.dme_shared_opf* method), 61
 pp_have_section_other() (*mp.dme_shared_opf* method), 61
 pp_have_section_sum() (*mp.dm_element* method), 47
 pp_have_section_sum() (*mp.dme_branch* method), 50
 pp_have_section_sum() (*mp.dme_gen* method), 55
 pp_have_section_sum() (*mp.dme_gen3p* method), 188
 pp_have_section_sum() (*mp.dme_legacy_dcline* method), 209
 pp_have_section_sum() (*mp.dme_line3p* method), 192
 pp_have_section_sum() (*mp.dme_load* method), 58
 pp_have_section_sum() (*mp.dme_load3p* method), 190
 pp_have_section_sum() (*mp.dme_reserve_gen* method), 179
 pp_have_section_sum() (*mp.dme_shunt* method), 60
 pp_have_section_sum() (*mp.dme_xfmr3p* method), 193
 pp_rows() (*mp.dm_element* method), 45
 pp_rows_lim() (*mp.dme_shared_opf* method), 61
 pp_rows_other() (*mp.dme_shared_opf* method), 61
 pp_section() (*mp.data_model* method), 35
 pp_section_label() (*mp.data_model* method), 34
 pp_section_list() (*mp.data_model* method), 34

pp_section_list() (*mp.data_model_opf* method), 38
 pp_set_tols_lim() (*mp.dme_shared_opf* method), 61
 PQ (*mp.NODE_TYPE* attribute), 172
 pq_capability_constraint() (*mp.mme_gen_opf_ac* method), 156
 pqcost() (*built-in function*), 351
 pretty_print() (*mp.data_model* method), 33
 pretty_print() (*mp.dm_element* method), 45
 pretty_print() (*mp.dme_branch_opf* method), 51
 pretty_print() (*mp.dme_gen_opf* method), 56
 pretty_print() (*mp.dme_legacy_dcline_opf* method), 210
 pretty_print() (*mp.dme_line3p* method), 192
 pretty_print() (*mp.dme_xfmr3p* method), 193
 print_soln() (*mp.task* method), 14
 print_soln_header() (*mp.task* method), 14
 print_soln_header() (*mp.task_opf* method), 24
 printpf() (*built-in function*), 253
 psse2mpc() (*built-in function*), 253
 psse_convert() (*built-in function*), 369
 psse_convert_hvdc() (*built-in function*), 369
 psse_convert_xfmr() (*built-in function*), 370
 psse_parse() (*built-in function*), 371
 psse_parse_line() (*built-in function*), 372
 psse_parse_section() (*built-in function*), 372
 psse_read() (*built-in function*), 374
 ptol (*mp.dme_shared_opf* attribute), 61
 PV (*mp.NODE_TYPE* attribute), 172
 pw11 (*mp.dmce_gen_mpc2* attribute), 72
 pw1_params() (*mp.cost_table* method), 166
 pw1_params() (*mp.cost_table_utils* static method), 168

Q

q_fr_lb (*mp.dme_legacy_dcline* attribute), 209
 q_fr_start (*mp.dme_legacy_dcline* attribute), 208
 q_fr_ub (*mp.dme_legacy_dcline* attribute), 209
 q_to_lb (*mp.dme_legacy_dcline* attribute), 209
 q_to_start (*mp.dme_legacy_dcline* attribute), 209
 q_to_ub (*mp.dme_legacy_dcline* attribute), 209
 qd (*mp.dme_load* attribute), 57
 qd_i (*mp.dme_load* attribute), 57
 qd_z (*mp.dme_load* attribute), 57
 qq1_start (*mp.dme_buslink* attribute), 194
 qq1_start (*mp.dme_gen3p* attribute), 188
 qq2_start (*mp.dme_buslink* attribute), 194
 qq2_start (*mp.dme_gen3p* attribute), 188
 qq3_start (*mp.dme_buslink* attribute), 194
 qq3_start (*mp.dme_gen3p* attribute), 188
 qq_lb (*mp.dme_gen* attribute), 55
 qq_start (*mp.dme_gen* attribute), 54
 qq_ub (*mp.dme_gen* attribute), 55
 qps_matpower() (*built-in function*), 327

R

r (*mp.dme_branch* attribute), 49
 r (*mp.dme_xfmr3p* attribute), 193
 r_ub (*mp.dme_reserve_gen* attribute), 179
 radial_pf() (*built-in function*), 271
 rate_a (*mp.dme_branch* attribute), 50
 rebuild() (*mp.data_model* method), 32
 rebuild() (*mp.dm_element* method), 45
 REF (*mp.NODE_TYPE* attribute), 172
 ref (*mp.task_pf* attribute), 21
 ref0 (*mp.task_pf* attribute), 21
 remove_userfcn() (*built-in function*), 307
 req (*mp.dme_reserve_zone* attribute), 180
 run() (*mp.task* method), 12
 run_cpf() (*built-in function*), 5
 run_mp() (*built-in function*), 4

run_opf() (built-in function), 6
 run_pf() (built-in function), 5
 run_post() (mp.task method), 13
 run_post() (mp.task_cpf_legacy method), 27
 run_post() (mp.task_opf_legacy method), 28
 run_post() (mp.task_pf_legacy method), 25
 run_pre() (mp.task method), 13
 run_pre() (mp.task_cpf method), 23
 run_pre() (mp.task_cpf_legacy method), 26
 run_pre() (mp.task_opf method), 24
 run_pre() (mp.task_opf_legacy method), 28
 run_pre() (mp.task_pf method), 22
 run_pre() (mp.task_pf_legacy method), 25
 run_pre_legacy() (mp.task_shared_legacy method), 29
 run_userfcn() (built-in function), 307
 runcpf() (built-in function), 230
 rundcpf() (built-in function), 236
 rundcpf() (built-in function), 235
 runduopf() (built-in function), 237
 runopf() (built-in function), 233
 runopf_w_res() (built-in function), 238
 runpf() (built-in function), 229
 runuopf() (built-in function), 234

S

s (mp.form_ac attribute), 78
 save() (mp.dm_converter method), 63
 save() (mp.dm_converter_mpc2 method), 64
 save2psse() (built-in function), 254
 save_soln() (mp.task method), 14
 savecase() (built-in function), 255
 savechgtab() (built-in function), 255
 scale_factor_fcn() (mp.dmce_load_mpc2 method), 73
 scale_load() (built-in function), 351
 set_bus_type_pq() (mp.dme_bus method), 53
 set_bus_type_pv() (mp.dme_bus method), 53
 set_bus_type_ref() (mp.dme_bus method), 53
 set_bus_v_lims_via_vg() (mp.data_model method), 36
 set_mpc() (opf_model method), 225
 set_node_type_pq() (mp.net_model method), 103
 set_node_type_pq() (mp.nm_element method), 117
 set_node_type_pq() (mp.nme_bus method), 119
 set_node_type_pv() (mp.net_model method), 103
 set_node_type_pv() (mp.nm_element method), 116
 set_node_type_pv() (mp.nme_bus method), 119
 set_node_type_ref() (mp.net_model method), 102
 set_node_type_ref() (mp.nm_element method), 116
 set_node_type_ref() (mp.nme_bus method), 119
 set_reorder() (built-in function), 262
 set_table() (mp_table_subclass method), 164
 set_type_idx_map() (mp.net_model method), 98
 set_type_label() (mp.net_model method), 99
 size() (mp.mapped_array method), 170
 size() (mp_table method), 160
 sm_legacy_cost (class in mp), 173
 sm_legacy_cost() (mp.sm_legacy_cost method), 173
 snln (mp.form_ac attribute), 78
 snln_hess (mp.form_ac attribute), 78
 soln (mp.nm_element attribute), 112
 solve_opts() (mp.math_model_opf_ac method), 135
 solve_opts() (mp.math_model_opf_dc method), 140
 solve_opts() (mp.math_model_pf method), 129
 solve_opts() (mp.math_model_pf_dc method), 132
 solve_opts_warmstart() (mp.math_model_cpf_acps method), 134
 source (mp.data_model attribute), 31
 stack_matrix_params() (mp.net_model method), 96

stack_vector_params() (mp.net_model method), 96
 start_cost_export() (mp.dmce_gen_mpc2 method), 73
 start_cost_import() (mp.dmce_gen_mpc2 method), 73
 state (mp.net_model attribute), 95
 subsasgn() (mp.mapped_array method), 172
 subsasgn() (mp_table method), 161
 subsref() (mp.mapped_array method), 171
 subsref() (mp_table method), 161
 success (mp.task attribute), 11
 symmat2vec() (mp.dme_line3p method), 192
 sys_wide_zip_loads() (mp.dmce_load_mpc2 method), 73

T

t_apply_changes() (built-in function), 374
 t_auction_case() (built-in function), 384
 t_auction_minopf() (built-in function), 375
 t_auction_mips() (built-in function), 375
 t_auction_tspopf_pdipm() (built-in function), 375
 t_case30_userfcns() (built-in function), 384
 t_case3p_a() (built-in function), 221
 t_case3p_b() (built-in function), 221
 t_case3p_c() (built-in function), 221
 t_case3p_d() (built-in function), 221
 t_case3p_e() (built-in function), 221
 t_case3p_f() (built-in function), 222
 t_case3p_g() (built-in function), 222
 t_case3p_h() (built-in function), 222
 t_case9_dcline() (built-in function), 384
 t_case9_gizmo() (built-in function), 222
 t_case9_opf() (built-in function), 384
 t_case9_opfv2() (built-in function), 385
 t_case9_pf() (built-in function), 385
 t_case9_pfv2() (built-in function), 385
 t_case9_save2psse() (built-in function), 385
 t_case_ext() (built-in function), 385
 t_case_int() (built-in function), 385
 t_chgtab() (built-in function), 375
 t_cpf() (built-in function), 375
 t_cpf_cb1() (built-in function), 386
 t_cpf_cb2() (built-in function), 386
 t_dcline() (built-in function), 375
 t_dmc_element() (built-in function), 218
 t_ext2int2ext() (built-in function), 375
 t_feval_w_path() (built-in function), 376
 t_get_losses() (built-in function), 376
 t_hasPQcap() (built-in function), 376
 t_hessian() (built-in function), 376
 t_islands() (built-in function), 376
 t_jacobian() (built-in function), 376
 t_load2disp() (built-in function), 377
 t_loadcase() (built-in function), 377
 t_makeLODF() (built-in function), 377
 t_makePTDF() (built-in function), 377
 t_margcost() (built-in function), 377
 t_miqps_matpower() (built-in function), 377
 t_modcost() (built-in function), 377
 t_mp_data_model() (built-in function), 218
 t_mp_dm_converter_mpc2() (built-in function), 218
 t_mp_mapped_array() (built-in function), 217
 t_mp_table() (built-in function), 218
 t_mpopoption() (built-in function), 378
 t_mpopoption_ov() (built-in function), 378
 t_mpxt_legacy_dcline() (built-in function), 220
 t_mpxt_reserves() (built-in function), 220
 t_nm_element() (built-in function), 218
 t_node_test() (built-in function), 219
 t_off2case() (built-in function), 378

t_opf_dc_bmpmd() (built-in function), 378
 t_opf_dc_clp() (built-in function), 378
 t_opf_dc_cplex() (built-in function), 378
 t_opf_dc_default() (built-in function), 378
 t_opf_dc_glpk() (built-in function), 379
 t_opf_dc_gurobi() (built-in function), 379
 t_opf_dc_ipopt() (built-in function), 379
 t_opf_dc_mips() (built-in function), 379
 t_opf_dc_mips_sc() (built-in function), 379
 t_opf_dc_mosek() (built-in function), 379
 t_opf_dc_osqp() (built-in function), 379
 t_opf_dc_ot() (built-in function), 380
 t_opf_default() (built-in function), 380
 t_opf_fmincon() (built-in function), 380
 t_opf_ipopt() (built-in function), 380
 t_opf_knitro() (built-in function), 380
 t_opf_minopf() (built-in function), 380
 t_opf_mips() (built-in function), 380
 t_opf_model() (built-in function), 381
 t_opf_softlims() (built-in function), 381
 t_opf_tspopf_pdipm() (built-in function), 381
 t_opf_tspopf_scpdipm() (built-in function), 381
 t_opf_tspopf_tralm() (built-in function), 381
 t_opf_userfcns() (built-in function), 381
 t_pf_ac() (built-in function), 382
 t_pf_dc() (built-in function), 382
 t_pf_radial() (built-in function), 382
 t_port_inj_current_acc() (built-in function), 218
 t_port_inj_current_acp() (built-in function), 219
 t_port_inj_power_acc() (built-in function), 219
 t_port_inj_power_acp() (built-in function), 219
 t_pretty_print() (built-in function), 220
 t_printf() (built-in function), 382
 t_psse() (built-in function), 382
 t_qps_matpower() (built-in function), 382
 t_run_mp() (built-in function), 219
 t_run_mp_3p() (built-in function), 219
 t_run_opf_default() (built-in function), 220
 t_runmarket() (built-in function), 382
 t_runopf_w_res() (built-in function), 383
 t_scale_load() (built-in function), 383
 t_total_load() (built-in function), 383
 t_totcost() (built-in function), 383
 t_vdep_load() (built-in function), 383
 ta (mp.dme_branch attribute), 50
 tab (mp.dm_element attribute), 40
 table_exists() (mp.dm_element method), 42
 table_var_map() (mp.dmc_element method), 68
 table_var_map() (mp.dmce_branch_mpc2 method), 72
 table_var_map() (mp.dmce_bus3p_mpc2 method), 184
 table_var_map() (mp.dmce_bus_mpc2 method), 72
 table_var_map() (mp.dmce_buslink_mpc2 method), 186
 table_var_map() (mp.dmce_gen3p_mpc2 method), 184
 table_var_map() (mp.dmce_gen_mpc2 method), 73
 table_var_map() (mp.dmce_legacy_dcline_mpc2 method), 207
 table_var_map() (mp.dmce_line3p_mpc2 method), 185
 table_var_map() (mp.dmce_load3p_mpc2 method), 185
 table_var_map() (mp.dmce_load_mpc2 method), 73
 table_var_map() (mp.dmce_reserve_gen_mpc2 method), 178
 table_var_map() (mp.dmce_reserve_zone_mpc2 method), 178
 table_var_map() (mp.dmce_shunt_mpc2 method), 74
 table_var_map() (mp.dmce_xfmr3p_mpc2 method), 185
 tag (mp.task attribute), 11
 tag (mp.task_pf attribute), 21
 task (class in mp), 9
 task_class() (mp.extension method), 174
 task_cpf (class in mp), 22
 task_cpf() (mp.task_cpf method), 23

task_cpf_legacy (class in mp), 26
 task_name() (mp.math_model method), 125
 task_name() (mp.math_model_opf method), 134
 task_name() (mp.math_model_pf method), 129
 task_opf (class in mp), 24
 task_opf_legacy (class in mp), 27
 task_pf (class in mp), 21
 task_pf_legacy (class in mp), 25
 task_shared_legacy (class in mp), 29
 task_tag() (mp.math_model method), 125
 task_tag() (mp.math_model_opf method), 134
 task_tag() (mp.math_model_pf method), 129
 tbus (mp.dme_branch attribute), 49
 tbus (mp.dme_legacy_dcline attribute), 208
 tbus (mp.dme_line3p attribute), 191
 tbus (mp.dme_xfmr3p attribute), 193
 tbus_on (mp.dme_legacy_dcline attribute), 208
 test_matpower() (built-in function), 217
 the_np (mp.net_model attribute), 95
 the_nz (mp.net_model attribute), 95
 tm (mp.dme_branch attribute), 50
 toggle_dcline() (built-in function), 307
 toggle_iflims() (built-in function), 308
 toggle_reserves() (built-in function), 309
 toggle_softlims() (built-in function), 310
 total_load() (built-in function), 353
 totcost() (built-in function), 304
 type (mp.dme_bus attribute), 52
 type (mp.dme_bus3p attribute), 186

U

uopf() (built-in function), 288
 update_mupq() (built-in function), 304
 update_rm_vars() (mp.math_model method), 128
 update_status() (mp.data_model method), 32
 update_status() (mp.dm_element method), 44
 update_status() (mp.dme_branch method), 50
 update_status() (mp.dme_bus method), 52
 update_status() (mp.dme_bus3p method), 187
 update_status() (mp.dme_buslink method), 195
 update_status() (mp.dme_gen method), 55
 update_status() (mp.dme_gen3p method), 188
 update_status() (mp.dme_legacy_dcline method), 209
 update_status() (mp.dme_line3p method), 192
 update_status() (mp.dme_load method), 58
 update_status() (mp.dme_load3p method), 190
 update_status() (mp.dme_reserve_gen method), 179
 update_status() (mp.dme_reserve_zone method), 180
 update_status() (mp.dme_shunt method), 60
 update_status() (mp.dme_xfmr3p method), 193
 update_z() (mp.mm_shared_pfcpf_ac method), 141
 update_z() (mp.mm_shared_pfcpf_dc method), 145
 userdata (mp.data_model attribute), 31

V

va (mp.net_model_dc attribute), 110
 va1_start (mp.dme_bus3p attribute), 186
 va2_start (mp.dme_bus3p attribute), 187
 va3_start (mp.dme_bus3p attribute), 187
 va_fcn() (mp.form_acc method), 87
 va_fcn() (mp.mme_buslink_opf_acc method), 205
 va_hess() (mp.form_acc method), 88
 va_hess() (mp.mme_buslink_opf_acc method), 205
 va_ref0 (mp.task_pf attribute), 21
 va_start (mp.dme_bus attribute), 52
 vec2symmat() (mp.dme_line3p method), 192
 vec2symmat_stacked() (mp.mme_line3p method), 199

`vertcat()` (*mp_table* method), 162
`violated_q_lims()` (*mp.dme_gen* method), 55
`vm1_setpoint` (*mp.dme_gen3p* attribute), 188
`vm1_start` (*mp.dme_bus3p* attribute), 186
`vm2_fcn()` (*mp.form_acc* method), 88
`vm2_fcn()` (*mp.mme_buslink_opf_acc* method), 205
`vm2_hess()` (*mp.form_acc* method), 88
`vm2_hess()` (*mp.mme_buslink_opf_acc* method), 205
`vm2_setpoint` (*mp.dme_gen3p* attribute), 188
`vm2_start` (*mp.dme_bus3p* attribute), 186
`vm3_setpoint` (*mp.dme_gen3p* attribute), 188
`vm3_start` (*mp.dme_bus3p* attribute), 186
`vm_control` (*mp.dme_bus* attribute), 52
`vm_control` (*mp.dme_bus3p* attribute), 187
`vm_lb` (*mp.dme_bus* attribute), 52
`vm_setpoint` (*mp.dme_gen* attribute), 55
`vm_setpoint_fr` (*mp.dme_legacy_dcline* attribute), 209
`vm_setpoint_to` (*mp.dme_legacy_dcline* attribute), 209
`vm_start` (*mp.dme_bus* attribute), 52
`vm_ub` (*mp.dme_bus* attribute), 52
`voltage_constraints()` (*mp.mme_buslink_pf_ac* method), 202
`voltage_constraints()` (*mp.mme_buslink* method), 200

W

`warmstart` (*mp.task_cpf* attribute), 23

X

`x` (*mp.dme_branch* attribute), 49
`x` (*mp.dme_xfmr3p* attribute), 193
`x2vz()` (*mp.nm_element* method), 114
`xt_3p` (class in *mp*), 182
`xt_legacy_dcline` (class in *mp*), 206
`xt_oval_cap_curve` (class in *mp*), 215
`xt_reserves` (class in *mp*), 177

Y

`Y` (*mp.form_ac* attribute), 78
`yc` (*mp.dme_line3p* attribute), 191
`ys` (*mp.dme_line3p* attribute), 191

Z

`z` (*mp.net_model_dc* attribute), 110
`zg_x_update()` (*mp.math_model_pf_acps* method), 131
`zgausspf()` (built-in function), 271
`zones` (*mp.dme_reserve_zone* attribute), 180