MATPOWER

MATPOWER Reference Manual

Release 8.0b1

Ray Zimmerman

Mar 21, 2024

Contents

1 Introduction

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 Functions

2.1 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.

2.1.1 run_mp

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

run_mp() (page 3) - 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 4), *run_cpf()* (page 5), or *run_opf()* (page 5).

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 18) for power flow, *mp.task_cpf* (page 20) for CPF, and *mp.task_opf* (page 21) for OPF
- **d** – data source specification, currently assumed to be a MATPOWER case name or case struct (`mpc`)
- **mpopt** (*struct*) – 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 7)) – 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 4), *run_cpf()* (page 5), *run_opf()* (page 5), *mp.task* (page 7).

2.1.2 run_pf

run_pf(*varargin*)

run_pf() (page 4) - 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 3), 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*) – 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 18)) – 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 3), *mp.task_pf* (page 18).

2.1.3 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 3), 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*) – 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 20)) – 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 3), *mp.task_cpf* (page 20).

2.1.4 run_opf

run_opf(*varargin*)

run_opf() (page 5) 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 3), 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*) – 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_opf* (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_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 3), *mp.task_opf* (page 21).

2.2 Other Functions

2.2.1 mp_table_class

mp_table_class()

mp_table_class() (page 6) - Returns handle to constructor for `table` or *mp_table* (page 155).

Returns a handle to `table` constructor, if it is available, otherwise to *mp_table* (page 155) constructor. Useful for table-based code that is compatible with both MATLAB (using native tables) and Octave (using *mp_table* (page 155) 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 155).

3 Classes

3.1 Task Classes

3.1.1 Core Task Classes

mp.task

class `mp.task`

Bases: `handle`

`mp.task` - MATPOWER task abstract base class.

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

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

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

mp.task Methods:

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

Property Summary

`tag`

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

`name`

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

`dmc`

(`mp.dm_converter`) data model converter object

`dm`

(`mp.data_model`) data model object

`nm`

(`mp.net_model`) network model object

`mm`

(`mp.math_model`) 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` flag, 1 - math model solved, 0 - didn't solve

`message`

(char array) output `message`

`et`

(double) elapsed time (seconds) for `run()` method

Method Summary

`run`(*d*, *mpopt*, *mpx*)

Execute the task.

```
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`)
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output
task (`mp.task`) – task object containing the solved `run()` 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, mpoopt, mpx)
```

Inputs
- **mm** (`mp.math_model`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output
- **mm** (`mp.math_model`) – new or updated mathmatical model object, or empty matrix
- **nm** (`mp.net_model`) – potentially updated network model object
- **dm** (`mp.data_model`) – potentially updated data model object

Called automatically by `run()` 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, mpoopt, mpx)
```

Inputs
- **mm** (`mp.math_model`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

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

Called automatically by `run()` 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, mpoopt, mpx)
```

Inputs
- **mm** (`mp.math_model`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

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

Called automatically by `run()` 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()` 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`)
- **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()` method.

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

Inputs

- **mm** (`mp.math_model`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task (`mp.task`) – 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(mpopt)
task.print_soln(mpopt, 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(mpopt, fd)
```

Inputs

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

Called by `print_soln()` 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`) – 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()` 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 handel*) – handle to default constructor to be used to instantiate the data model converter object

Called by `dm_converter_class()` 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`) – MATPOWER Extensions

Output

dmc (`mp.dm_converter`) – data model converter object, ready to build

Called by `dm_converter_build()` 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`) – MATPOWER Extensions

Output

dmc (`mp.dm_converter`) – data model converter object, ready for use

Called by `run()` 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`) – 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()` 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 handel*) – handle to default constructor to be used to instantiate the data model object

Called by `data_model_class()` 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`) – MATPOWER Extensions

Output

dm (`mp.data_model`) – data model object, ready to build

Called by `data_model_build()` 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_create(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`) – data model converter object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

dm (`mp.data_model`) – data model object, ready for use

Called by `run()` 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()`.

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

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

Outputs
- **dm** (`mp.data_model`) – 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()`.

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

Inputs
- **dm** (`mp.data_model`) – data model object
- **dmc** (`mp.dm_converter`) – data model converter object
- **mpopt** (*struct*) – MATPOWER options struct

Output

dm (`mp.data_model`) – 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`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – 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()` 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`) – 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()` 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`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

nm (`mp.net_model`) – network model object, ready to build

Called by `network_model_build()` 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`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

nm (`mp.net_model`) – network model object, ready for use

Called by `run()` 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()`.

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

Inputs
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output
nm (`mp.net_model`) – 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()`.

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

Inputs
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output
nm (`mp.net_model`) – 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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object

Output
nm (`mp.net_model`) – updated network model object

Called by `network_model_update()`.

`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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object

Output
nm (`mp.net_model`) – updated network model object

Called by `run()` 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – 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()` 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`) – network model object
- **dm** (`mp.data_model`) – 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()` 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct
- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

mm (`mp.math_model`) – mathmatical model object, ready to build

Called by `math_model_build()` 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

- **mpx** (cell array of `mp.extension`) – MATPOWER Extensions

Output

mm (`mp.math_model`) – mathmatical model object, ready for use

Called by `run()` 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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

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

Called by `run()` method.

mp.task_pf

class `mp.task_pf`

Bases: `mp.task`

`mp.task_pf` - 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` - task tag 'PF'
- `name` - task name 'Power Flow'
- `dc` - `true` if using DC network model
- `iterations` - total number of power flow iterations
- `ref` - current ref node indices
- `ref0` - initial ref node indices
- `va_ref0` - initial ref node voltage angles
- `fixed_q_idx` - indices of fixed Q gens
- `fixed_q_qty` - Q output of fixed Q gens

mp.task_pf Methods:

- `run_pre()` - set `dc` property
- `next_dm()` - optionally iterate to enforce generator reactive limits
- `enforce_q_lims()` - implementation of generator reactive limits

- `network_model_class_default()` - select default network model constructor
- `network_model_build_post()` - initialize properties for reactive limits
- `network_model_x_soln()` - correct the voltage angles if necessary
- `math_model_class_default()` - select default math model constructor

See also `mp.task`.

Property Summary

`tag = 'PF'`

`name = 'Power Flow'`

`dc`

`true` if using DC network model (from `mpopt.model`, cached in `run_pre()`)

`iterations`

(integer) total number of power flow `iterations`

`ref`

(integer) current `ref` 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()`.

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

Implement optional iterations to enforce generator reactive limits.

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

Used by `next_dm()` 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` properties, if non-empty AC case with generator reactive limits enforced.

`network_model_x_soln`(*mm*, *nm*)

Call superclass `network_model_x_soln()` 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`

`mp.task_cpf` - 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` - warm start data

mp.task_cpf Methods:

- `task_cpf()` - constructor, inherits from `mp.task_pf` constructor
- `run_pre()` - call superclass `run_pre()` for base and target inputs
- `next_mm()` - handle warm start of continuation iterations
- `dm_converter_class()` - select data model converter class
- `data_model_class_default()` - select default data model constructor
- `data_model_build()` - build base and target data models
- `network_model_build()` - build base and target network models
- `network_model_x_soln()` - update network model solution
- `network_model_update()` - evaluate port injection solution
- `math_model_class_default()` - select default math model constructor
- `math_model_opt()` - add warmstart parameters to math model solve options

See also `mp.task`, `mp.task_pf`.

Constructor Summary

`task_cpf()`

Constructor, inherits from `mp.task_pf` 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()` 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()` 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`

`mp.task_opf` - 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` - `true` if using DC network model

mp.task_opf Methods:

- `run_pre()` - set dc property
- `print_soln_header()` - add printout of objective function value
- `data_model_class_default()` - select default data model constructor
- `data_model_build_post()` - adjust bus voltage limits, if requested

- `network_model_class_default()` - select default network model constructor
- `math_model_class_default()` - select default math model constructor

See also `mp.task`.

Property Summary

`dc`

`true` if using DC network model (from `mpopt.model`, cached in `run_pre()`)

Method Summary

`run_pre`(*d*, *mpopt*)

Set `dc` property after calling superclass `run_pre()`, then check for unsupported AC OPF solver selection.

`print_soln_header`(*mpopt*, *fd*)

Call superclass `print_soln_header()` 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()` 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`, `mp.task_shared_legacy`

`mp.task_pf_legacy` - 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()` - pre-process inputs that are for legacy framework only
- `run_post()` - export results back to data model source
- `legacy_post_run()` - post-process *legacy framework* outputs

See also `mp.task_pf`, `mp.task`, `mp.task_shared_legacy`.

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`)
- **mpopt** (*struct*) – MATPOWER options struct

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

Call `run_pre_legacy()` 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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output

task (`mp.task`) – task object

Calls `mp.dm_converter.export()` 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(mpopt)
```

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`, `mp.task_shared_legacy`

`mp.task_cpf` - 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()` - pre-process inputs that are for legacy framework only
- `run_post()` - export results back to data model source
- `legacy_post_run()` - post-process *legacy framework* outputs

See also `mp.task_cpf`, `mp.task`, `mp.task_shared_legacy`.

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`)
- **mpopt** (*struct*) – MATPOWER options struct

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

Call `run_pre_legacy()` 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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output
task (`mp.task`) – task object

Calls `mp.dm_converter.export()` 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(mpopt)
```

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`, `mp.task_shared_legacy`

`mp.task_opf` - 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()` - pre-process inputs that are for legacy framework only
- `run_post()` - export results back to data model source
- `dm_converter_class_mpc2_default()` - set to `mp.dm_converter_mpc2_legacy`
- `data_model_build_post()` - get data model converter to do more input pre-processing
- `math_model_class_default()` - use legacy math model subclasses
- `legacy_post_run()` - post-process *legacy framework* outputs

See also `mp.task_opf`, `mp.task`, `mp.task_shared_legacy`.

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`)
- **mpopt** (*struct*) – MATPOWER options struct

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

Call `run_pre_legacy()` 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`) – mathmatical model object
- **nm** (`mp.net_model`) – network model object

- **dm** (mp.data_model) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output
task (mp.task) – task object

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

`dm_converter_class_mpc2_default()`

Set to `mp.dm_converter_mpc2_legacy`.

```
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()`.

`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.mm_shared_opf_legacy` (compatible with the legacy `opf_model`), 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(mpopt)
```

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()`, `dcopf_solver()`, and `nlpopf_solver()`.

mp.task_shared_legacy

class `mp.task_shared_legacy`

Bases: `handle`

`mp.task_shared_legacy` - 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()` - handle experimental system-wide ZIP load inputs

See also `mp.task`.

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)
- **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`).

Called by `run_pre()`.

3.2 Data Model Classes

3.2.1 Containers

mp.data_model

class mp.**data_model**

Bases: `mp.element_container`

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

mp.data_model Methods:

- `data_model()` - constructor, assign default data model element classes
- `copy()` - make duplicate of object

- `build()` - create, add, and build element objects
- `count()` - count instances of each element and remove if count is zero
- `initialize()` - initialize (online/offline) status of each element
- `update_status()` - update (online/offline) status based on connectivity, etc
- `build_params()` - extract/convert/calculate parameters for online elements
- `online()` - get number of online elements of named type
- `display()` - display the data model object
- `pretty_print()` - pretty print data model to console or file
- `pp_flags()` - from options, build flags to control pretty printed output
- `pp_section_label()` - construct section header lines for output
- `pp_section_list()` - return list of section tags
- `pp_have_section()` - return true if section exists for object
- `pp_section()` - pretty print the given section
- `pp_get_headers()` - construct pretty printed lines for section headers
- `pp_get_headers_cnt()` - construct pretty printed lines for **cnt** section headers
- `pp_get_headers_ext()` - construct pretty printed lines for **ext** section headers
- `pp_data()` - pretty print the data for the given section
- `set_bus_v_lims_via_vg()` - 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`, `mp.net_model`, `mp.math_model`, `mp.dm_converter`.

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` 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()` 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`)
- **dmc** (`mp.dm_converter`) – data model converter

Create the data model element objects by instantiating each class in the `element_classes` property and adding the resulting object to the `elements` property. Then proceed through the following additional `build()` 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()` is zero.

```
dm.count()
```

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

Called by `build()` 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()` method to `initialize()` statuses and create ID to row index mappings.

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

`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()` 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 retuns 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(mpopt)
dm.pretty_print(mpopt, fd)
[dm, out] = dm.pretty_print(mpopt, 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`) – 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(mpopt)
```

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()`.

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()`.

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

See also `pretty_print()`.

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()`.

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()`.

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()`.

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()`, `pp_get_headers()`.

`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()`, `pp_get_headers()`.

`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()`, `pp_get_headers()`.

`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()`, `pp_section()`.

`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`

`mp.data_model_cpf` - 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()` - constructor, assign default data model element classes

See also `mp.data_model`.

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`

`mp.data_model_opf` - 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()` - constructor, assign default data model element classes
- `pp_flags()` - add flags for **lim** sections
- `pp_section_list()` - append `'lim'` tag for **lim** sections to default list
- `pp_get_headers_other()` - construct headers for **lim** section headers

See also `mp.data_model`.

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()`.

pp_section_list(*out*)

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

See `mp.data_model.pp_section_list()`.

pp_get_headers_other(*section*, *out_s*, *mpopt*)

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

See `mp.data_model.pp_get_headers_other()`.

3.2.2 Elements

mp.dm_element

class `mp.dm_element`

Bases: `handle`

`mp.dm_element` - 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` 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
`uid`	*integer*	unique ID
`name`	*char array*	element name
`status`	*boolean*	true = online, false = offline
`source_uid`	*undefined*	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 - main data table
- nr - total number of rows in table
- n - number of online elements
- ID2i - max(ID) x 1 vector, maps IDs to row indices
- on - n x 1 vector of row indices of online elements
- off - (nr-n) x 1 vector of row indices of offline elements
- i2on - nr x 1 vector mapping row index to index in on/off respectively

mp.dm_element Methods:

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

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

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_idx_prop()` returns *empty*.
2. Cell array with *nc* elements, one for each connection, `cxn_idx_prop()` returns *empty*.
3. Cell array of valid junction element types, `cxn_idx_prop()` 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()`, `cxn_type_prop()`.

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()`, `cxn_type_prop()`.

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()`

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()`.

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()`, `cxn_idx_prop()`.

`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()` 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()`.

`dm_converter_element`(*dmc*, *name*)

Get corresponding data model converter element.

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

Inputs

- **dmc** (`mp.dm_converter`) – data model converter object
- **name** (*char array*) – name of element type

Output

dmce (`mp.dmc_element`) – 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`) – `copy()` 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`) – data model

Output

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

Called for each element by the `count()` method of `mp.data_model` 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`) – 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()`.

Called for each element by the `initialize()` method of `mp.data_model` 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`) – data model

Called by `initialize()`. 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`) – 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()` method of `mp.data_model` 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`) – 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()` method of `mp.data_model` 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()`, `initialize()`, `build_params()`.

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

Input

dm (`mp.data_model`) – 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 retuns 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`) – 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()` 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()`.

See also `pp_have_section_cnt()`, `pp_have_section_sum()`, `pp_have_section_ext()`, `pp_have_section_det()`.

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()` 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()` 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()` 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()` 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()` 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()`
- **...** – see `pretty_print()` for details of other inputs

Implementation handled by section-specific *pp_data* methods or `pp_data_other()`.

See also `pp_data_cnt()`, `pp_data_sum()`, `pp_data_ext()`, `pp_data_det()`.

`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()`.

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()`.

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()`.

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()`.

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()`.

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()`.

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()`.

pp_get_title_det(*mpopt*, *pp_args*)

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

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

Inputs
see `pretty_print()` for details

Output

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

Called by pp_get_headers_det() 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().

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().

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() for each row.

See also pp_data(), pp_data_row_det().

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() for details of other inputs

Output

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

Called by pp_data_det() for each row.

mp.dme_branch

class mp.**dme_branch**

Bases: mp.dm_element

mp.dme_branch - 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
`bus_fr`	*integer*	bus ID (`uid`) of "from" bus
`bus_to`	*integer*	bus ID (`uid`) of "to" bus
`r`	*double*	per unit series resistance
`x`	*double*	per unit series reactance
`g_fr`	*double*	per unit shunt conductance at "from" end
`b_fr`	*double*	per unit shunt susceptance at "from" end
`g_to`	*double*	per unit shunt conductance at "to" end
`b_to`	*double*	per unit shunt susceptance at "to" end
`sm_ub_a`	*double*	long term apparent power rating (MVA)
`sm_ub_b`	*double*	short term apparent power rating (MVA)
`sm_ub_c`	*double*	emergency apparent power rating (MVA)
`cm_ub_a`	*double*	long term current magnitude rating (MVA equivalent at 1 p.u. voltage)
`cm_ub_b`	*double*	short term current magnitude rating (MVA equivalent at 1 p.u. voltage)
`cm_ub_c`	*double*	emergency current magnitude rating (MVA equivalent at 1 p.u. voltage)
`vad_lb`	*double*	voltage angle difference lower bound
`vad_ub`	*double*	voltage angle difference upper bound
`tm`	*double*	transformer off-nominal turns ratio
`ta`	*double*	transformer phase-shift angle (degrees)
`pl_fr`	*double*	active power injection at "from" end
`ql_fr`	*double*	reactive power injection at "from" end
`pl_to`	*double*	active power injection at "to" end
`ql_to`	*double*	reactive power injection 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`

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

rate_a

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

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(*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(*mpopt*, *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`, `mp.dme_shared_opf`

`mp.dme_branch_opf` - Data model element for branch for OPF.

To parent class `mp.dme_branch`, 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
mu_flow_fr_u	*double*	shadow price on flow constraint at “from” end *(u/MVA)*[1]
mu_flow_to_u	*double*	shadow price on flow constraint at “to” end *(u/MVA)*[Page 48, 1]
mu_vad_lb	*double*	shadow price on lower bound of voltage angle difference constraint *(u/degree)*[1]
mu_vad_ub	*double*	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(*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_data_row_lim(*dm*, *k*, *out_e*, *mpopt*, *fd*, *pp_args*)

mp.dme_bus

class mp.**dme_bus**

Bases: `mp.dm_element`

`mp.dme_bus` - 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	*double*	base voltage *(kV)*
type	*integer*	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
area	*integer*	area number
zone	*integer*	loss zone
vm_lb	*double*	voltage magnitude lower bound *(p.u.)*
vm_ub	*double*	voltage magnitude upper bound *(p.u.)*
va	*double*	voltage angle *(degrees)*
vm	*double*	voltage magnitude *(p.u.)*

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

Property Summary

`type`
: node `type` 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`(*mpopt*, *pp_args*)

`pp_data_ext`(*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*)

`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`, `mp.dme_shared_opf`

`mp.dme_bus_opf` - Data model element for bus for OPF.

To parent class `mp.dme_bus`, 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`	*double*	active power nodal price, i.e. shadow price on active power balance constraint *(u/MW)*[1]
`lam_q`	*double*	reactive power nodal price, i.e. shadow price on reactive power balance constraint *(u/MVAr)*[1]
`mu_vm_lb`	*double*	shadow price on voltage magnitude lower bound *(u/p.u.)*[1]
`mu_vm_ub`	*double*	shadow price on voltage magnitude upper bound *(u/p.u.)*[1]

Method Summary

main_table_var_names()

export_vars()

export_vars_offline_val()

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*)

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

mp.dme_gen

class `mp.dme_gen`

Bases: `mp.dm_element`

`mp.dme_gen` - 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`	*integer*	bus ID (`uid`)
`vm_setpoint`	*double*	voltage magnitude setpoint *(p.u.)*
`pg_lb`	*double*	active power output lower bound *(MW)*
`pg_ub`	*double*	active power output upper bound *(MW)*
`qg_lb`	*double*	reactive power output lower bound *(MVAr)*
`qg_ub`	*double*	reactive power output upper bound *(MVAr)*
`pg`	*double*	active power output *(MW)*
`qg`	*double*	reactive power output *(MVAr)*
`startup_cost_cold`	*double*	cold startup cost *(USD)*
`pc1`	*double*	lower active power output of PQ capability curve *(MW)*
`pc2`	*double*	upper active power output of PQ capability curve *(MW)*
`qc1_lb`	*double*	lower bound on reactive power output at `pc1` *(MVAr)*
`qc1_ub`	*double*	upper bound on reactive power output at `pc1` *(MVAr)*
`qc2_lb`	*double*	lower bound on reactive power output at `pc2` *(MVAr)*
`qc2_ub`	*double*	upper bound on reactive power output at `pc2` *(MVAr)*

Property Summary

`bus`
: `bus` 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`, `mp.dme_shared_opf`

`mp.dme_gen_opf` - Data model element for generator for OPF.

To parent class `mp.dme_gen`, 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
cost_pg	*mp.cost_table*	active power cost *(u/MW)*[1]
cost_qg	*mp.cost_table*	reactive power cost *(u/MVAr)*[Page 53, 1]
mu_pg_lb	*double*	shadow price on active power output lower bound *(u/MW)*[1]
mu_pg_ub	*double*	shadow price on active power output upper bound *(u/MW)*[1]
mu_qg_lb	*double*	shadow price on reactive power output lower bound *(u/MVAr)*[1]
mu_qg_ub	*double*	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`.

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(*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_load

class mp.**dme_load**

Bases: `mp.dm_element`

`mp.dme_load` - 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:

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

Name	Type	Description
bus	*integer*	bus ID (uid)
pd	*double*	p_p, active constant power demand *(MW)*
qd	*double*	q_p, reactive constant power demand *(MVAr)*
pd_i	*double*	p_i, active nominal[1] constant current demand *(MW)*
qd_i	*double*	q_i, reactive nominal[Page 54, 1] constant current demand *(MVAr)*
pd_z	*double*	p_z, active nominal[1] constant impedance demand *(MW)*
qd_z	*double*	q_z, reactive nominal[1] constant impedance demand *(MVAr)*
p	*double*	p, total active demand *(MW)*
q	*double*	q, total reactive demand *(MVAr)*

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

$$\begin{aligned} \mathrm{s} &= \mathrm{s}_p + \mathrm{s}_i|\mathrm{v}| + \mathrm{s}_z|\mathrm{v}|^2 \\ p + jq &= (p_p + jq_p) + (p_i + jq_i)|\mathrm{v}| + (p_z + jq_z)|\mathrm{v}|^2 \end{aligned} \tag{3.1}$$

Property Summary

bus

bus 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()

label()

labels()

cxn_type()

cxn_idx_prop()

main_table_var_names()

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

count(*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_load_cpf

class mp.**dme_load_cpf**

Bases: `mp.dme_load`

`mp.dme_load_cpf` - Data model element for load for CPF.

To parent class `mp.dme_load`, 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`, `mp.dme_shared_opf`

`mp.dme_load_opf` - Data model element for load for OPF.

To parent class `mp.dme_load`, adds pretty-printing for **lim** sections.

mp.dme_shunt_cpf

class mp.**dme_shunt_cpf**

Bases: `mp.dme_shunt`

`mp.dme_shunt_cpf` - Data model element for shunt for CPF.

To parent class `mp.dme_shunt`, 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_shunt

class mp.**dme_shunt**

Bases: `mp.dm_element`

`mp.dme_shunt` - 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`	*integer*	bus ID (`uid`)
`gs`	*double*	g_s, shunt conductance, specified as nominal[1] active power demand *(MW)*
`bs`	*double*	b_s, shunt susceptance, specified as nominal[1] reactive power injection *(MVAr)*
`p`	*double*	p, total active power absorbed *(MW)*
`q`	*double*	q, total reactive power absorbed *(MVAr)*

Property Summary

bus

`bus` 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()`

`count(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`, `mp.dme_shared_opf`

`mp.dme_shunt_opf` - Data model element for shunt for OPF.

To parent class `mp.dme_shunt`, 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` - Mixin class for OPF **data model element** objects.

For all elements of `mp.data_model_opf`, 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`

`mp.dm_converter` - 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`) 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()` - return char array identifier for data source/format
- `copy()` - make duplicate of object
- `build()` - create and add element objects
- `import()` - import data from a data source into a data model
- `export()` - export data from a data model to a data source
- `init_export()` - initialize a data source for export
- `save()` - save data source to a file
- `display()` - 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`, `mp.task`.

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` property and adding the resulting object to the `elements` property.

`import`(*dm*, *d*)

Import data from a data source into a data model.

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

Inputs
- **dm** (`mp.data_model`) – data model
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

Output
dm (`mp.data_model`) – updated data model

Calls the `import()` 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`) – data model
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

Output
d – updated data source

Calls the `export()` 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`) – data model

Output
d – new empty data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

Creates a new data source of the appropriate type in preparation for calling `export()`.

`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`)

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 retuns 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`

`mp.dm_converter_mpc2` - 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()` - constructor
- `format_tag()` - return char array identifier for data source/format (`'mpc2'`)
- `import()` - import data from a MATPOWER case struct into a data model
- `export()` - export data from a data model to a MATPOWER case struct
- `save()` - save MATPOWER case struct to a file

See also `mp.dm_converter`.

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`

`mp.dm_converter_mpc2_legacy` - Legacy MATPOWER **data model converter** for MATPOWER case v2.

Adds to `mp.dm_converter_mpc2` the ability to handle legacy user customization.

mp.dm_converter_mpc2_legacy Methods:

- `legacy_user_mod_inputs()` - pre-process legacy inputs for use-defined customization
- `legacy_user_nln_constraints()` - pre-process legacy inputs for user-defined nonlinear constraints

See also `mp.dm_converter`, `mp.dm_converter_mpc2`, `mp.task_opf_legacy`.

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`, `fparm`, `H1`, `Cw`, `z0`, `zl`, `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()` method.

3.3.2 Elements

mp.dmc_element

class `mp.dmc_element`

Bases: `handle`

`mp.dmc_element`- 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` 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()` method.

mp.dmc_element Methods:

- `name()` - get name of element type, e.g. `'bus'`, `'gen'`
- `data_model_element()` - get corresponding data model element
- `data_field()` - get name of field in data source corresponding to default data table
- `data_subs()` - get subscript reference struct for accessing data source

- `data_exists()` - check if default field exists in data source
- `get_import_spec()` - get import specification
- `get_export_spec()` - get export specification
- `get_import_size()` - get dimensions of data to be imported
- `get_export_size()` - get dimensions of data to be exported
- `table_var_map()` - get variable map for import/export
- `import()` - import data from data source into data model element
- `import_table_values()` - import table values for given import specification
- `get_input_table_values()` - get values to insert in data model element table
- `import_col()` - extract and optionally modify values from data source column
- `export()` - export data from data model element to data source
- `export_table_values()` - export table values for given import specification
- `init_export_data()` - initialize data source for export from data model element
- `default_export_data_table()` - create default (empty) data table for data source
- `default_export_data_nrows()` - get number of rows `default_export_data_table()`
- `export_col()` - 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`.

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, name)
```

Inputs
- **dm** (`mp.data_model`) – data model object
- **name** (*char array*) – name of element type

Output
dme (`mp.dm_element`) – 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()` 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`)

Output
TorF (*boolean*) – true if field exists

Check if value returned by `data_field()` 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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

Output
spec (*struct*) – import specification, with keys:

- `'subs'` - subscript reference struct for accessing data source, as returned by `data_subs()`
- `'nr'`, `'nc'`, `'r'` - number of rows, number of columns, row index vector, as returned by `get_import_size()`
- `'vmap'` - variable map, as returned by `table_var_map()`

See also `get_export_spec()`.

get_export_spec(*dme*, *d*)

Get export specification.

```
spec = dmce.get_export_spec(dme, d)
```

Inputs
- **dme** (`mp.dm_element`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

Output
spec (*struct*) – export specification, see `get_import_spec()`

See also `get_import_spec()`.

`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`)

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`) – 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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)

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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **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`) – updated data model element object

See also `export()`.

`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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **spec** (*struct*) – import specification, see `get_import_spec()`
- **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`) – updated data model element object

Called by `import()`.

`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`)
- **spec** (*struct*) – import specification, see `get_import_spec()`
- **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()`.

`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`)
- **spec** (*struct*) – import specification, see `get_import_spec()`
- **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()`.

`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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **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()`.

`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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **spec** (*struct*) – export specification, see `get_export_spec()`
- **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()`.

`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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **spec** (*struct*) – export specification, see `get_export_spec()`

Output

d – updated data source

Called by `export_table_values()`.

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()`

Output
- **dt** – data table for data source, type depends on implementing subclass

Called by `init_export_data()`.

default_export_data_nrows(*spec*)

Get number of rows for `default_export_data_table()`.

```
nr = default_export_data_nrows(spec)
```

Input
- **spec** (*struct*) – export specification, see `get_export_spec()`

Output
- **nr** (*integer*) – number of rows

Called by `default_export_data_table()`.

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`) – data model element object
- **d** – data source, type depends on the implementing subclass (e.g. MATPOWER case struct for `mp.dm_converter_mpc2`)
- **spec** (*struct*) – export specification, see `get_export_spec()`
- **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()`.

mp.dmce_branch_mpc2

class mp.**dmce_branch_mpc2**

Bases: `mp.dmc_element`

`mp.dmce_branch_mpc2` - 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`

`mp.dmce_bus_mpc2` - 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`

`mp.dmce_gen_mpc2` - Data model converter element for generator for MATPOWER case v2.

Property Summary

`pwl1`

indices 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`

`mp.dmce_load_mpc2` - 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`

`mp.dmce_shunt_mpc2` - 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` - 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` and `mp.form`.

`mp.form` 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()` - get char array w/name of formulation
- `form_tag()` - get char array w/short label of formulation
- `model_params()` - get cell array of names of model parameters
- `model_vvars()` - get cell array of names of voltage state variables
- `model_zvars()` - get cell array of names of non-voltage state variables
- `get_params()` - get network model element parameters
- `find_form_class()` - get name of network element object's formulation subclass

See also `mp.nm_element`.

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 properies 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 netork model elements parent classes, the `mp.form` subclass, that is not a subclass of `mp.nm_element`, with the longest inheritance path back to `mp.form`.

mp.form_ac

`class mp.form_ac`

Bases: `mp.form`

`mp.form_ac` - 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` and `mp.form_ac`.

`mp.form_ac` defines the complex port injections as functions of the state variables $\mathbf{x}$, that is, the complex voltages $\mathbf{v}$ and non-voltage states $\mathbf{z}$. The are defined in terms of 3 compoents, the linear current injection and linear power injection components,

$$\begin{aligned} \mathbf{i}^{lin}(\mathbf{x}) &= \left[\begin{array}{cc} \underline{\mathbf{Y}} & \underline{\mathbf{L}} \end{array}\right] \mathbf{x} + \underline{\mathbf{i}} \\ &= \underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}} \end{aligned} \tag{3.2}$$

$$\begin{aligned} \mathbf{s}^{lin}(\mathbf{x}) &= \left[\begin{array}{cc} \underline{\mathbf{M}} & \underline{\mathbf{N}} \end{array}\right] \mathbf{x} + \underline{\mathbf{s}} \\ &= \underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}}, \end{aligned} \tag{3.3}$$

and an arbitrary nonlinear injection component represented by $\mathbf{s}^{nln}(\mathbf{x})$ or $\mathbf{i}^{nln}(\mathbf{x})$. The full complex power and current port injection functions implemented by `mp.form_ac`, are respectively

$$\begin{aligned} \mathbf{g}^S(\mathbf{x}) &= \left[{}^{\diagdown}\mathbf{v}_{\diagdown}\right] \left(\mathbf{i}^{lin}(\mathbf{x})\right)^* + \mathbf{s}^{lin}(\mathbf{x}) + \mathbf{s}^{nln}(\mathbf{x}) \\ &= \left[{}^{\diagdown}\mathbf{v}_{\diagdown}\right] \left(\underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}}\right)^* + \underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}} + \mathbf{s}^{nln}(\mathbf{x}) \end{aligned} \tag{3.4}$$

$$\begin{aligned} \mathbf{g}^I(\mathbf{x}) &= \mathbf{i}^{lin}(\mathbf{x}) + \left[{}^{\diagdown}\mathbf{s}^{lin}(\mathbf{x})_{\diagdown}\right]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x}) \\ &= \underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}} + \left[{}^{\diagdown}\underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}}_{\diagdown}\right]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x}) \end{aligned} \tag{3.5}$$

where $\underline{\mathbf{Y}}$, $\underline{\mathbf{L}}$, $\underline{\mathbf{M}}$, $\underline{\mathbf{N}}$, $\underline{\mathbf{i}}$, and $\underline{\mathbf{s}}$, along with $\mathbf{s}^{nln}(\mathbf{x})$ or $\mathbf{i}^{nln}(\mathbf{x})$, 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` - $n_p n_k \times n_n$ matrix $\underline{\mathbf{Y}}$ of model parameters
- `L` - $n_p n_k \times n_\mathbf{z}$ matrix $\underline{\mathbf{L}}$ of model parameters
- `M` - $n_p n_k \times n_n$ matrix $\underline{\mathbf{M}}$ of model parameters
- `N` - $n_p n_k \times n_\mathbf{z}$ matrix $\underline{\mathbf{N}}$ of model parameters
- `i` - $n_p n_k \times 1$ vector $\underline{\mathbf{i}}$ of model parameters
- `s` - $n_p n_k \times 1$ vector $\underline{\mathbf{s}}$ of model parameters
- `params_ncols` - specify number of columns for each parameter
- `inln` - function to compute $\mathbf{i}^{nln}(\mathbf{x})$
- `snln` - function to compute $\mathbf{s}^{nln}(\mathbf{x})$
- `inln_hess` - function to compute Hessian of $\mathbf{i}^{nln}(\mathbf{x})$
- `snln_hess` - function to compute Hessian of $\mathbf{s}^{nln}(\mathbf{x})$

mp.form_dc Methods:

- `model_params()` - get network model element parameters (`{'Y', 'L', 'M', 'N', 'i', 's'}`)
- `model_zvars()` - get cell array of names of non-voltage state variables (`{'zr', 'zi'}`)
- `port_inj_current()` - compute port current injections from network state
- `port_inj_power()` - compute port power injections from network state
- `port_inj_current_hess()` - compute Hessian of port current injections
- `port_inj_power_hess()` - compute Hessian of port power injections
- `port_inj_current_jac()` - abstract method to compute voltage-related Jacobian terms
- `port_inj_current_hess_v()` - abstract method to compute voltage-related Hessian terms
- `port_inj_current_hess_vz()` - abstract method to compute voltage-related Hessian terms
- `port_inj_power_jac()` - abstract method to compute voltage-related Jacobian terms
- `port_inj_power_hess_v()` - abstract method to compute voltage-related Hessian terms
- `port_inj_power_hess_vz()` - abstract method to compute voltage-related Hessian terms
- `port_apparent_power_lim_fcn()` - compute port squared apparent power injection constraints
- `port_active_power_lim_fcn()` - compute port active power injection constraints
- `port_active_power2_lim_fcn()` - compute port squared active power injection constraints
- `port_current_lim_fcn()` - compute port squared current injection constraints
- `port_apparent_power_lim_hess()` - compute port squared apparent power injection Hessian
- `port_active_power_lim_hess()` - compute port active power injection Hessian
- `port_active_power2_lim_hess()` - compute port squared active power injection Hessian
- `port_current_lim_hess()` - compute port squared current injection Hessian
- `aux_data_va_vm()` - abstract method to return voltage angles/magnitudes from auxiliary data

See also `mp.form`, `mp.form_acc`, `mp.form_acp`, `mp.form_dc`, `mp.nm_element`.

Property Summary

Y = []

(double) $n_p n_k \times n_n$ matrix $\underline{\mathbf{Y}}$ of model parameter coefficients for $\mathbf{v}$

L = []

(double) $n_p n_k \times n_\mathbf{z}$ matrix $\underline{\mathbf{L}}$ of model parameter coefficients for $\mathbf{z}$

M = []

(double) $n_p n_k \times n_n$ matrix $\underline{\mathbf{M}}$ of model parameter coefficients for $\mathbf{v}$

N = []

(double) $n_p n_k \times n_\mathbf{z}$ matrix $\underline{\mathbf{N}}$ of model parameter coefficients for $\mathbf{z}$

i = []

(double) $n_p n_k \times 1$ vector $\underline{\mathbf{i}}$ of model parameters

s = []

(double) $n_p n_k \times 1$ vector $\underline{\mathbf{s}}$ 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 => $n_\mathbf{z}$ 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 $\mathbf{s}^{nln}(\mathbf{x})$

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()`.

model_zvars()

Get cell array of names of non-voltage state variables, i.e. `{'zr', 'zi'}`.

See `mp.form.model_zvars()`.

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 `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, $\mathbf{g}^I_{\boldsymbol{\theta}}$ (polar) or $\mathbf{g}^I_{\boldsymbol{u}}$ (cartesian)
- **Iv2** (*complex double*) – Jacobian of port complex current injections w.r.t 2nd voltage component, $\mathbf{g}^I_{\boldsymbol{\nu}}$ (polar) or $\mathbf{g}^I_{\boldsymbol{w}}$ (cartesian)
- **Izr** (*complex double*) – Jacobian of port complex current injections w.r.t real part of non-voltage state, $\mathbf{g}^I_{\boldsymbol{z}_r}$
- **Izi** (*complex double*) – Jacobian of port complex current injections w.r.t imaginary part of non-voltage state, $\mathbf{g}^I_{\boldsymbol{z}_i}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`.

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)
[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, $\mathbf{g}^S_{\boldsymbol{\theta}}$ (polar) or $\mathbf{g}^S_{\boldsymbol{u}}$ (cartesian)

- **Sv2** (*complex double*) – Jacobian of port complex power injections w.r.t 2nd voltage component, $\mathbf{g}^S_{\boldsymbol{\nu}}$ (polar) or $\mathbf{g}^S_{\boldsymbol{w}}$ (cartesian)
- **Szr** (*complex double*) – Jacobian of port complex power injections w.r.t real part of non-voltage state, $\mathbf{g}^S_{\boldsymbol{z}_r}$
- **Szi** (*complex double*) – Jacobian of port complex power injections w.r.t imaginary part of non-voltage state, $\mathbf{g}^S_{\boldsymbol{z}_i}$.

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_current()`.

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 muliplying 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 $\mathbf{g}^I_{\mathbf{xx}}(\boldsymbol{\lambda})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_current()`.

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 muliplying the transpose of the port power 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 power injections corresponding to specified $\boldsymbol{\lambda}$, namely $\mathbf{g}^S_{\mathbf{xx}}(\boldsymbol{\lambda})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`.

`port_inj_current_jac`(*n*, *v_*, *Y*, *M*, *invdiagvic*, *diagSlincJ*)

Abstract method to compute voltage-related Jacobian terms.

Called by `port_inj_current()` to compute voltage-related Jacobian terms. See `mp.form_acc.port_inj_current_jac()` and `mp.form_acp.port_inj_current_jac()` for details.

`port_inj_current_hess_v`(*x_*, *lam*, *v_*, *z_*, *diaginvic*, *Y*, *M*, *diagSlincJ*, *dlamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_current_hess()` to compute voltage-related Hessian terms. See `mp.form_acc.port_inj_current_hess_v()` and `mp.form_acp.port_inj_current_hess_v()` for details.

`port_inj_current_hess_vz`(*x_*, *lam*, *v_*, *z_*, *diaginvic*, *N*, *dlamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_current_hess()` to compute voltage/non-voltage-related Hessian terms. See `mp.form_acc.port_inj_current_hess_vz()` and `mp.form_acp.port_inj_current_hess_vz()` 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()` to compute voltage-related Jacobian terms. See `mp.form_acc.port_inj_power_jac()` and `mp.form_acp.port_inj_power_jac()` for details.

`port_inj_power_hess_v`(*x_*, *lam*, *v_*, *z_*, *diagvi*, *Y*, *M*, *diagIlincJ*, *dlamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_power_hess()` to compute voltage-related Hessian terms. See `mp.form_acc.port_inj_power_hess_v()` and `mp.form_acp.port_inj_power_hess_v()` for details.

`port_inj_power_hess_vz`(*x_*, *lam*, *v_*, *z_*, *diagvi*, *L*, *dlamJ*)

Abstract method to compute voltage-related Hessian terms.

Called by `port_inj_power_hess()` to compute voltage/non-voltage-related Hessian terms. See `mp.form_acc.port_inj_power_hess_vz()` and `mp.form_acp.port_inj_power_hess_vz()` 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()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **nm** (`mp.net_model`) – 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, $\boldsymbol{h}^{\text{flow}}(\boldsymbol{x})$

- **dh** (*double*) – constraint Jacobian, $\boldsymbol{h}_{\boldsymbol{x}}^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`.

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()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **nm** (`mp.net_model`) – 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, $\boldsymbol{h}^{\text{flow}}(\boldsymbol{x})$
- **dh** (*double*) – constraint Jacobian, $\boldsymbol{h}_{\boldsymbol{x}}^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`.

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()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **nm** (`mp.net_model`) – 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, $\boldsymbol{h}^{\text{flow}}(\boldsymbol{x})$
- **dh** (*double*) – constraint Jacobian, $\boldsymbol{h}_{\boldsymbol{x}}^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`.

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()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$

- **nm** (`mp.net_model`) – 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, $\boldsymbol{h}^{\text{flow}}(\boldsymbol{x})$
- **dh** (*double*) – constraint Jacobian, $\boldsymbol{h}_{\boldsymbol{x}}^{\text{flow}}$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_current()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\mu}$. Results are based on the complex outputs of `port_inj_power()` and `port_inj_power_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** (`mp.net_model`) – network model object
- **idx** (*integer*) – *(optional)* vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

d2H (*double*) – sparse constraint Hessian matrix, $\boldsymbol{h}_{\boldsymbol{xx}}^{\text{flow}}(\boldsymbol{\mu})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`, `port_inj_power_hess()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\mu}$. Results are based on the complex outputs of `port_inj_power()` and `port_inj_power_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** (`mp.net_model`) – network model object
- **idx** (*integer*) – *(optional)* vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

d2H (*double*) – sparse constraint Hessian matrix, $\boldsymbol{h}_{\boldsymbol{xx}}^{\text{flow}}(\boldsymbol{\mu})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`, `port_inj_power_hess()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\mu}$. Results are based on the complex outputs of `port_inj_power()` and `port_inj_power_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** (`mp.net_model`) – network model object
- **idx** (*integer*) – *(optional)* vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

d2H (*double*) – sparse constraint Hessian matrix, $\boldsymbol{h}^{\text{flow}}_{\boldsymbol{xx}}(\boldsymbol{\mu})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_power()`, `port_inj_power_hess()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\mu}$. Results are based on the complex outputs of `port_inj_current()` and `port_inj_current_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$
- **lam** (*double*) – vector $\boldsymbol{\mu}$ of multipliers, one for each port
- **nm** (`mp.net_model`) – network model object
- **idx** (*integer*) – *(optional)* vector of indices of ports of interest, if empty or missing, returns results corresponding to all ports

Outputs

d2H (*double*) – sparse constraint Hessian matrix, $\boldsymbol{h}^{\text{flow}}_{\boldsymbol{xx}}(\boldsymbol{\mu})$

For details on the derivations of the formulas used, see *MATPOWER Technical Note 5*.

See also `port_inj_current()`, `port_inj_current_hess()`.

`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()` and `mp.form_acp.aux_data_va_vm()`.

mp.form_acc

class mp.**form_acc**

Bases: `mp.form_ac`

`mp.form_acc` - 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** repesentation 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` and `mp.form_acc`.

Provides implementation of evaluation of voltage-related Jacobian and Hessian terms needed by some `mp.form_ac` methods.

mp.form_dc Methods:

- `form_name()` - get char array w/name of formulation (`'AC-cartesian'`)
- `form_tag()` - get char array w/short label of formulation (`'acc'`)
- `model_vvars()` - get cell array of names of voltage state variables (`{'vr', 'vi'}`)
- `port_inj_current_jac()` - compute voltage-related terms of current injection Jacobian
- `port_inj_current_hess_v()` - compute voltage-related terms of current injection Hessian
- `port_inj_current_hess_vz()` - compute voltage/non-voltage-related terms of current injection Hessian
- `port_inj_power_jac()` - compute voltage-related terms of power injection Jacobian
- `port_inj_power_hess_v()` - compute voltage-related terms of power injection Hessian
- `port_inj_power_hess_vz()` - compute voltage/non-voltage-related terms of power injection Hessian
- `aux_data_va_vm()` - return voltage angles/magnitudes from auxiliary data
- `va_fcn()` - compute voltage angle constraints and Jacobian
- `va_hess()` - compute voltage angle Hessian
- `vm2_fcn()` - compute squared voltage magnitude constraints and Jacobian
- `vm2_hess()` - 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`, `mp.form_ac`, `mp.form_acp`, `mp.nm_element`.

Method Summary

form_name()

Get user-readable name of formulation, i.e. `'AC-cartesian'`.

See `mp.form.form_name()`.

form_tag()

Get short label of formulation, i.e. `'acc'`.

See `mp.form.form_tag()`.

model_vvars()

Get cell array of names of voltage state variables, i.e. {'vr', 'vi'}.

See mp.form.model_vvars().

port_inj_current_jac(*n*, *v_*, *Y*, *M*, *invdiagvic*, *diagSlincJ*)

Compute voltage-related terms of current injection Jacobian.

```
[Iu, Iw] = nme.port_inj_current_jac(n, v_, Y, M, invdiagvic, diagSlincJ)
```

Called by mp.form_ac.port_inj_current() 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.

```
[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,
↪diagSlincJ, dlamJ)
```

Called by mp.form_ac.port_inj_current_hess() 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() 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.

```
[Su, Sw] = nme.port_inj_power_jac(...)
```

Called by mp.form_ac.port_inj_power() 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.

```
[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, diagIlincJ,
↪ dlamJ)
```

Called by mp.form_ac.port_inj_power_hess() 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()` 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)
```

Connverts 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, $\boldsymbol{g}(\boldsymbol{x})$
- **dg** (*double*) – constraint Jacobian, $\boldsymbol{g}_{\boldsymbol{x}}$

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\lambda}$.

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in `xx{1}`, imaginary part in `xx{2}`
- **lam** (*double*) – vector $\boldsymbol{\lambda}$ of multipliers, one for each constraint
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all

Outputs

d2G (*double*) – sparse constraint Hessian, $\boldsymbol{g}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$

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, $\boldsymbol{g}(\boldsymbol{x})$
- **dg** (*double*) – constraint Jacobian, $\boldsymbol{g_x}$

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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\lambda}$.

Inputs

- **xx** (*1 x 2 cell array*) – real part of complex voltage in `xx{1}`, imaginary part in `xx{2}`
- **lam** (*double*) – vector $\boldsymbol{\lambda}$ of multipliers, one for each constraint
- **idx** (*integer*) – index of subset of voltages of interest to include in constraint; if empty, include all

Outputs

d2G (*double*) – sparse constraint Hessian, $\boldsymbol{g_{xx}}(\boldsymbol{\lambda})$

mp.form_acp

class mp.form_acp

Bases: `mp.form_ac`

`mp.form_acp` - 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** repesentation 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` and `mp.form_acp`.

Provides implementation of evaluation of voltage-related Jacobian and Hessian terms needed by some `mp.form_ac` methods.

mp.form_dc Methods:

- `form_name()` - get char array w/name of formulation (`'AC-polar'`)
- `form_tag()` - get char array w/short label of formulation (`'acp'`)
- `model_vvars()` - get cell array of names of voltage state variables (`{'va', 'vm'}`)

- `port_inj_current_jac()` - compute voltage-related terms of current injection Jacobian
- `port_inj_current_hess_v()` - compute voltage-related terms of current injection Hessian
- `port_inj_current_hess_vz()` - compute voltage/non-voltage-related terms of current injection Hessian
- `port_inj_power_jac()` - compute voltage-related terms of power injection Jacobian
- `port_inj_power_hess_v()` - compute voltage-related terms of power injection Hessian
- `port_inj_power_hess_vz()` - compute voltage/non-voltage-related terms of power injection Hessian
- `aux_data_va_vm()` - 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`, `mp.form_ac`, `mp.form_acc`, `mp.nm_element`.

Method Summary

form_name()

Get user-readable name of formulation, i.e. `'AC-polar'`.

See `mp.form.form_name()`.

form_tag()

Get short label of formulation, i.e. `'acp'`.

See `mp.form.form_tag()`.

model_vvars()

Get cell array of names of voltage state variables, i.e. `{'va', 'vm'}`.

See `mp.form.model_vvars()`.

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()` 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()` 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()` 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()` 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, diagIlincJ,
↪ dlamJ)
```

Called by `mp.form_ac.port_inj_power_hess()` 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()` 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`

`mp.form_dc` - 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` and `mp.form_dc`.

`mp.form_dc` defines the port active power injection as a linear function of the state variables $\boldsymbol{x}$, that is, the voltage angles $\boldsymbol{\theta}$ and non-voltage states $\boldsymbol{z}$, as

$$\begin{aligned} \boldsymbol{g}^P(\boldsymbol{x}) &= \begin{bmatrix} \underline{\boldsymbol{B}} & \underline{\boldsymbol{K}} \end{bmatrix} \boldsymbol{x} + \underline{\boldsymbol{p}} \\ &= \underline{\boldsymbol{B}}\boldsymbol{\theta} + \underline{\boldsymbol{K}}\boldsymbol{z} + \underline{\boldsymbol{p}}, \end{aligned} \tag{3.6}$$

where $\underline{\boldsymbol{B}}$, $\underline{\boldsymbol{K}}$, and $\underline{\boldsymbol{p}}$ 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:

- `B` - $n_p n_k \times n_n$ matrix $\underline{\boldsymbol{B}}$ of model parameters
- `K` - $n_p n_k \times n_z$ matrix $\underline{\boldsymbol{K}}$ of model parameters
- `p` - $n_p n_k \times 1$ vector $\underline{\boldsymbol{p}}$ of model parameters
- `params_ncols` - specify number of columns for each parameter

mp.form_dc Methods:

- `form_name()` - get char array w/name of formulation (`'DC'`)
- `form_tag()` - get char array w/short label of formulation (`'dc'`)
- `model_params()` - get network model element parameters (`{'B', 'K', 'p'}`)
- `model_vvars()` - get cell array of names of voltage state variables (`{'va'}`)
- `model_zvars()` - get cell array of names of non-voltage state variables (`{'z'}`)
- `port_inj_power()` - compute port power injections from network state

See also `mp.form`, `mp.form_ac`, `mp.nm_element`.

Property Summary

`B = []`

(double) $n_p n_k \times n_n$ matrix $\underline{\boldsymbol{B}}$ of model parameter coefficients for $\boldsymbol{\theta}$

`K = []`

(double) $n_p n_k \times n_z$ matrix $\underline{\boldsymbol{K}}$ of model parameter coefficients for $\boldsymbol{z}$

`p = []`

(double) $n_p n_k \times 1$ vector $\underline{\boldsymbol{p}}$ 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().

form_tag()

Get short label of formulation, i.e. 'dc'.

See mp.form.form_tag().

model_params()

Get cell array of names of model parameters, i.e. {'B', 'K', 'p'}.

See mp.form.model_params().

model_vvars()

Get cell array of names of voltage state variables, i.e. {'va'}.

See mp.form.model_vvars().

model_zvars()

Get cell array of names of non-voltage state variables, i.e. {'z'}.

See mp.form.model_zvars().

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 $\boldsymbol{x}$
- **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, $\boldsymbol{g}^P(\boldsymbol{x})$

mp.net_model

class mp.**net_model**

Bases: `mp.nm_element`, `mp.element_container`, `mp_idx_manager`

`mp.net_model` - 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`) objects and *is itself* a network model element. All network model classes inherit from `mp.net_model` and therefore also from `mp.element_container`, mp_idx_manager, and `mp.nm_element`. Concrete network model classes are also formulation-specific, inheriting from a corresponding subclass of `mp.form`.

By convention, network model variables are named `nm` and network model class class names begin with `mp.net_model`.

mp.net_model Properties:

- `the_np` - total number of ports
- `the_nz` - total number of non-voltage states
- `nv` - total number of (real) voltage variables
- `node` - mp_idx_manager data for nodes
- `port` - mp_idx_manager data for ports
- `state` - mp_idx_manager data for non-voltage states

mp.net_model Methods:

- `name()` - return name of this network element type (`'network'`)
- `np()` - return number of ports for this network element
- `nz()` - return number of *(possibly complex)* non-voltage states for this network element
- `build()` - create, add, and build network model element objects
- `add_nodes()` - elements add nodes, then add corresponding voltage variables
- `add_states()` - elements add states, then add corresponding state variables
- `build_params()` - build incidence matrices, parameters, add ports for each element
- `stack_matrix_params()` - form network matrix parameter by stacking corresponding element parameters
- `stack_vector_params()` - form network vector parameter by stacking corresponding element parameters
- `add_vvars()` - add voltage variable(s) for each network node
- `add_zvars()` - add non-voltage state variable(s) for each network state
- `def_set_types()` - define node, state, and port set types for mp_idx_manager
- `init_set_types()` - initialize structures for tracking/indexing nodes, states, ports
- `display()` - display the network model object
- `add_node()` - add named set of nodes
- `add_port()` - add named set of ports

- `add_state()` - add named set of states
- `set_type_idx_map()` - map node/port/state index back to named set & index within set
- `set_type_label()` - create a user-readable label to identify a node, port, or state
- `add_var()` - add a set of variables to the model
- `params_var()` - return initial value, bounds, and variable type for variables
- `get_node_idx()` - get index information for named node set
- `get_port_idx()` - get index information for named port set
- `get_state_idx()` - get index information for named state set
- `node_types()` - get node type information
- `ensure_ref_node()` -
- `set_node_type_ref()` - make the specified node a reference node
- `set_node_type_pv()` - make the specified node a PV node
- `set_node_type_pq()` - 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`, `mp.nm_element`, `mp.task`, `mp.data_model`, `mp.math_model`.

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()` network model element objects.

```
nm.build(dm)
```

Input

dm (`mp.data_model`) – 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()`, `add_states()`, `build_params()`.

add_nodes(*nm*, *dm*)

Elements add nodes, then add corresponding voltage variables.

```
nm.add_nodes(nm, dm)
```

Inputs

- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – 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()`, `add_states()`.

add_states(*nm*, *dm*)

Elements add states, then add corresponding state variables.

```
nm.add_states(nm, dm)
```

Inputs

- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – data model object

Each element can add its states, then corresponding non-voltage state variables are added for each state.

See also `add_zvars()`, `add_nodes()`.

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`) – network model object
- **dm** (`mp.data_model`) – 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **idx** (*integer*) – index for name and indexed variables *(currently unused here)*

Also updates the `nv` property.

See also `add_zvars()`, `add_nodes()`.

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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **idx** (*cell array*) – indices for named and indexed variables *(currently unused here)*

See also `add_vvars()`, `add_states()`.

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 retuns 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`) – 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`) – 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 =

  4×1 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, vl, vu, vt)
nm.add_var(vtype, name, N, v0, vl, vu)
nm.add_var(vtype, name, N, v0, vl)
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)
[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`) – network model object
- **dm** (`mp.data_model`) – 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`
  - `mp.NODE_TYPE.PV`
  - `mp.NODE_TYPE.PQ`
- **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`, `ensure_ref_node()`.

`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`) – 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
  - mp.NODE_TYPE.PV
  - mp.NODE_TYPE.PQ

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`) – 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`.

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`) – 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`.

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`) – 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`.

mp.net_model_ac

class `mp.`**net_model_ac**

Bases: `mp.net_model`

`mp.net_model_ac` - Abstract base class for MATPOWER AC **network model** objects.

Explicitly a subclass of `mp.net_model` and implicitly assumed to be a subclass of `mp.form_ac` 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()` - add non-voltage state variable set types for mp_idx_manager
- `build_params()` - build incidence matrices, parameters, add ports for each element
- `port_inj_nln()` - compute general nonlinear port injection functions and Jacobians
- `port_inj_nln_hess()` - compute general nonlinear port injection Hessian
- `nodal_complex_current_balance()` - compute nodal complex current balance constraints
- `nodal_complex_power_balance()` - compute nodal complex power balance constraints
- `nodal_complex_current_balance_hess()` - compute nodal complex current balance Hessian
- `nodal_complex_power_balance_hess()` - compute nodal complex power balance Hessian
- `port_inj_soln()` - compute the network port power injections at the solution
- `get_va()` - get node voltage angle vector

See also `mp.net_model`, `mp.form`, `mp.form_ac`, `mp.nm_element`.

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()`, `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`) – network model object
- **dm** (`mp.data_model`) – 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, $\mathbf{s}^{nln}(\mathbf{x})$ or $\mathbf{i}^{nln}(\mathbf{x})$, 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, gzr, gzi] = nm.port_inj_nln(si, x_, sysx, idx)
```

Compute and assemble the functions, and optionally Jacobians, for the general nonlinear injection functions $\mathbf{s}^{nln}(\mathbf{x})$ and $\mathbf{i}^{nln}(\mathbf{x})$ 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 $\mathbf{s}^{nln}(\mathbf{x})$

– 'I' for complex current $\mathbf{i}^{nln}(\mathbf{x})$

- **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

- **g** (*complex double*) – nonlinear injection function, $\mathbf{s}^{nln}(\mathbf{x})$ (or $\mathbf{i}^{nln}(\mathbf{x})$)
- **gv1** (*complex double*) – Jacobian w.r.t. 1st voltage variable, $\mathbf{s}^{nln}_{\boldsymbol{\theta}}$ or $\mathbf{s}^{nln}_{\boldsymbol{u}}$ (or $\mathbf{i}^{nln}_{\boldsymbol{\theta}}$ or $\mathbf{i}^{nln}_{\boldsymbol{u}}$)
- **gv2** (*complex double*) – Jacobian w.r.t. 2nd voltage variable, $\mathbf{s}^{nln}_{\boldsymbol{\nu}}$ or $\mathbf{s}^{nln}_{\boldsymbol{w}}$ (or $\mathbf{i}^{nln}_{\boldsymbol{\nu}}$ or $\mathbf{i}^{nln}_{\boldsymbol{w}}$)
- **gzr** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $\mathbf{s}^{nln}_{z_r}$ (or $\mathbf{i}^{nln}_{z_r}$)
- **gzi** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $\mathbf{s}^{nln}_{z_i}$ (or $\mathbf{i}^{nln}_{z_i}$)

See also `port_inj_nln_hess()`.

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 $\mathbf{s}^{nln}(\mathbf{x})$ and $\mathbf{i}^{nln}(\mathbf{x})$ 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 muliplying the transpose of the corresponding Jacobian by a vector $\boldsymbol{\lambda}$.

Inputs

- **si** ('S' or 'I') – select power or current injection function:
  - 'S' for complex power $\mathbf{s}^{nln}(\mathbf{x})$
  - 'I' for complex current $\mathbf{i}^{nln}(\mathbf{x})$
- **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

Output

H (*complex double*) – sparse Hessian matrix, $\mathbf{s}^{nln}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$ or $\mathbf{i}^{nln}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$

See also `port_inj_nln()`.

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()` 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, $\mathbf{g}^{\mathrm{kcl}}(\boldsymbol{x})$
- **Gv1** (*complex double*) – Jacobian w.r.t. 1st voltage variable, $\mathbf{g}^{\mathrm{kcl}}_{\boldsymbol{\theta}}$ or $\mathbf{g}^{\mathrm{kcl}}_{\boldsymbol{u}}$
- **Gv2** (*complex double*) – Jacobian w.r.t. 2nd voltage variable, $\mathbf{g}^{\mathrm{kcl}}_{\boldsymbol{\nu}}$ or $\mathbf{g}^{\mathrm{kcl}}_{\boldsymbol{w}}$

- **Gzr** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $\mathbf{g}^{\text{kcl}}_{z_r}$
- **Gzi** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $\mathbf{g}^{\text{kcl}}_{z_i}$

See also `mp.form_ac.port_inj_current()`, `nodal_complex_current_balance_hess()`.

`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()` 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, $\mathbf{g}^{\text{kcl}}(\boldsymbol{x})$
- **Gv1** (*complex double*) – Jacobian w.r.t. 1st voltage variable, $\mathbf{g}^{\text{kcl}}_{\boldsymbol{\theta}}$ or $\mathbf{g}^{\text{kcl}}_{\boldsymbol{u}}$
- **Gv2** (*complex double*) – Jacobian w.r.t. 2nd voltage variable, $\mathbf{g}^{\text{kcl}}_{\boldsymbol{\nu}}$ or $\mathbf{g}^{\text{kcl}}_{\boldsymbol{w}}$
- **Gzr** (*complex double*) – Jacobian w.r.t. real non-voltage variable, $\mathbf{g}^{\text{kcl}}_{z_r}$
- **Gzi** (*complex double*) – Jacobian w.r.t. imaginary non-voltage variable, $\mathbf{g}^{\text{kcl}}_{z_i}$

See also `mp.form_ac.port_inj_power()`, `nodal_complex_power_balance_hess()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\lambda}$. Based on `mp.form_ac.port_inj_current_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$ (full system state)
- **lam** (*double*) – vector $\boldsymbol{\lambda}$ of multipliers, one for each node

Output

d2G (*complex double*) – sparse Hessian matrix, $\mathbf{g}^{\text{kcl}}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$

See also `mp.form_ac.port_inj_current_hess()`, `nodal_complex_current_balance()`.

`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 muliplying the transpose of the constraint Jacobian by a vector $\boldsymbol{\lambda}$. Based on `mp.form_ac.port_inj_power_hess()`.

Inputs

- **x_** (*complex double*) – state vector $\mathbf{x}$ (full system state)
- **lam** (*double*) – vector $\boldsymbol{\lambda}$ of multipliers, one for each node

Output

d2G (*complex double*) – sparse Hessian matrix, $\mathbf{g}^{\text{kcl}}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$

See also `mp.form_ac.port_inj_power_hess()`, `nodal_complex_power_balance()`.

`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`, `mp.form_acc`

`mp.net_model_acc` - 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`.

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()` - constructor, assign default network model element classes
- `def_set_types()` - add voltage state variable set types for mp_idx_manager
- `initial_voltage_angle()` - get vector of initial node voltage angles

See also `mp.net_model_ac`, `mp.net_model`, `mp.form_acc`, `mp.form_ac`, `mp.form`, `mp.nm_element`.

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`.

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(), mp.net_model.def_set_types(), 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, mp.form_acp

mp.net_model_acp - 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.

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() - constructor, assign default network model element classes
- def_set_types() - add voltage state variable set types for mp_idx_manager
- initial_voltage_angle() - get vector of initial node voltage angles

See also mp.net_model_ac, mp.net_model, mp.form_acp, mp.form_ac, mp.form, mp.nm_element.

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`.

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()`, `mp.net_model.def_set_types()`, `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`, `mp.form_dc`

`mp.net_model_dc` - 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`.

mp.net_model_dc Properties:

- `va` - vector of voltage states (voltage angles $\boldsymbol{\theta}$)
- `z` - vector of non-voltage states $\boldsymbol{z}$

mp.net_model_dc Methods:

- `net_model_dc()` - constructor, assign default network model element classes
- `def_set_types()` - add voltage and non-voltage variable set types for mp_idx_manager
- `build_params()` - build incidence matrices, parameters, add ports for each element
- `port_inj_soln()` - compute the network port injections at the solution

See also `mp.net_model`, `mp.form_dc`, `mp.form`, `mp.nm_element`.

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`.

Property Summary

va = []

(double) vector of voltage states (voltage angles $\boldsymbol{\theta}$)

z = []

(double) vector of non-voltage states $\boldsymbol{z}$

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()`, `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`) – network model object
- **dm** (`mp.data_model`) – 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 - 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 and also, like the container, from a formulation-specific subclass of mp.form. 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 - number of elements of this type
- C - stacked sparse element-node incidence matrices
- D - stacked sparse incidence matrices for z-variables
- soln - struct for storing solved states, quantities

mp.mm_element Methods:

- name() - get name of element type, e.g. 'bus', 'gen'
- np() - number of ports per element of this type
- nn() - number of nodes per element, created by this element type
- nz() - number of non-voltage state variables per element of this type
- data_model_element() - get the corresponding data model element
- math_model_element() - get the corresponding math model element
- count() - get number of online elements in dm, set nk
- add_nodes() - add nodes to network model
- add_states() - add non-voltage states to network model
- add_vvars() - add real-valued voltage variables to network object
- add_zvars() - add real-valued non-voltage state variables to network object
- build_params() - build model parameters from data model
- get_nv_() - get number of *(possibly complex)* voltage variables
- x2vz() - get port voltages and non-voltage states from combined state vector
- node_indices() - construct node indices from data model element connection info
- incidence_matrix() - construct stacked incidence matrix from set of index vectors
- node_types() - get node type information
- set_node_type_ref() - make the specified node a reference node
- set_node_type_pv() - make the specified node a PV node
- set_node_type_pq() - make the specified node a PQ node

- `display()` - 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`.

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, name)
```

Inputs
- **dm** (`mp.data_model`) – data model object
- **name** (*char array*) – name of element type

Output
dme (`mp.dm_element`) – data model element object

`math_model_element`(*mm*, *name*)

Get the corresponding math model element.

```
mme = nme.math_model_element(mm, name)
```

Inputs
- **mm** (`mp.math_model`) – math model object
- **name** (*char array*) – name of element type

Output
mme (`mp.mm_element`) – 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`) – 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`) – network model object
- **dm** (`mp.data_model`) – data model object

Add nodes to the network model object, based on value *nn* returned by `nn()`. Calls the network model's `add_node()` *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`) – network model object
- **dm** (`mp.data_model`) – data model object

Add non-voltage states to the network model object, based on value *nz* returned by `nz()`. Calls the network model's `add_state()` *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`) – network model object
- **dm** (`mp.data_model`) – 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`).

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()` 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`) – network model object
- **dm** (`mp.data_model`) – 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`).

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()` 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`) – network model object
- **dm** (`mp.data_model`) – 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()`, `node_indices()`.

get_nv_(*sysx*)

Get number of *(possibly complex)* voltage variables.

```
nv_ = nme.get_nv_(sysx)
```

Input

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)*

Output

nv_ (*integer*) – number of *(possibly complex)* voltage variables in the state variable `x_`, 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`) – network model object
- **dm** (`mp.data_model`) – 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()` 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()` 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()` 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()`, `mp.dm_element.cxn_type()`, `mp.dm_element.cxn_idx_prop()`, `mp.dm_element.cxn_type_prop()`.

`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* x *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`) – network model object
- **dm** (`mp.data_model`) – 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`
  - `mp.NODE_TYPE.PV`
  - `mp.NODE_TYPE.PQ`
- **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`.

`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`) – 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`.

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`) – 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`.

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`) – 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`.

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 retuns 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`

`mp.nme_branch` - 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`

`mp.nme_branch_ac` - Network model element abstract base class for branch for AC formulations.

Implements building of the admittance parameter $\underline{\mathbf{Y}}$ for branches.

Method Summary

build_params(*nm*, *dm*)

Builds the admittance parameter $\underline{\mathbf{Y}}$ for branches.

mp.nme_branch_acc

class mp.**nme_branch_acc**

Bases: `mp.nme_branch_ac`, `mp.form_acc`

`mp.nme_branch_acc` - 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`.

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`, `mp.form_acp`

`mp.nme_branch_acp` - Network model element for branch for AC polar voltage formulations.

Inherits from `mp.form_acp`.

mp.nme_branch_dc

class mp.**nme_branch_dc**

Bases: `mp.nme_branch`, `mp.form_dc`

`mp.nme_branch_dc` - Network model element for branch for DC formulations.

Implements building of the branch parameters $\underline{\boldsymbol{B}}$ and $\underline{\boldsymbol{p}}$, and inherits from `mp.form_dc`.

Method Summary

build_params(*nm*, *dm*)

mp.nme_bus

class mp.**nme_bus**

Bases: mp.nm_element

mp.nme_bus - 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, mp.form_acc

mp.nme_bus_acc - Network model element for bus for AC cartesian voltage formulations.

Adds voltage variables Vr and Vi to the network model and inherits from mp.form_acc.

Method Summary

add_vvars(*nm*, *dm*, *idx*)

mp.nme_bus_acp

class mp.**nme_bus_acp**

Bases: mp.nme_bus, mp.form_acp

mp.nme_bus_acp - Network model element for bus for AC cartesian polar formulations.

Adds voltage variables Va and Vm to the network model and inherits from mp.form_acp.

Method Summary

add_vvars(*nm*, *dm*, *idx*)

mp.nme_bus_dc

`class mp.nme_bus_dc`

Bases: `mp.nme_bus`, `mp.form_dc`

`mp.nme_bus_dc` - Network model element for bus for DC formulations.

Adds voltage variable `Va` to the network model and inherits from `mp.form_dc`.

Method Summary

add_vvars(*nm*, *dm*, *idx*)

mp.nme_gen

`class mp.nme_gen`

Bases: `mp.nm_element`

`mp.nme_gen` - 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`

`mp.nme_gen_ac` - Network model element abstract base class for generator for AC formulations.

Adds non-voltage state variables `Pg` and `Qg` to the network model and builds the parameter $\underline{\mathbf{N}}$.

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`, `mp.form_acc`

`mp.nme_gen_acc` - Network model element for generator for AC cartesian voltage formulations.

Inherits from `mp.form_acc`.

mp.nme_gen_acp

class `mp.nme_gen_acp`

Bases: `mp.nme_gen_ac`, `mp.form_acp`

`mp.nme_gen_acp` - Network model element for generator for AC polar voltage formulations.

Inherits from `mp.form_acp`.

mp.nme_gen_dc

class `mp.nme_gen_dc`

Bases: `mp.nme_gen`, `mp.form_dc`

`mp.nme_gen_dc` - Network model element for generator for DC formulations.

Adds non-voltage state variable `Pg` to the network model, builds the parameter $\underline{\boldsymbol{K}}$, and inherits from `mp.form_dc`.

Method Summary

`add_zvars`(*nm*, *dm*, *idx*)

`build_params`(*nm*, *dm*)

mp.nme_load

class `mp.nme_load`

Bases: `mp.nm_element`

`mp.nme_load` - 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`

`mp.nme_load_ac` - Network model element abstract base class for load for AC formulations.

Builds the parameters $\underline{\mathbf{s}}$ and $\underline{\mathbf{Y}}$ and nonlinear functions $\mathbf{s}^{nln}(\mathbf{x})$ and $\mathbf{i}^{nln}(\mathbf{x})$.

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`, `mp.form_acc`

`mp.nme_load_acc` - Network model element for load for AC cartesian voltage formulations.

Inherits from `mp.form_acc`.

mp.nme_load_acp

class mp.**nme_load_acp**

Bases: `mp.nme_load_ac`, `mp.form_acp`

`mp.nme_load_acp` - Network model element for load for AC polar voltage formulations.

Inherits from `mp.form_acp`.

mp.nme_load_dc

class mp.**nme_load_dc**

Bases: `mp.nme_load`, `mp.form_dc`

`mp.nme_load_dc` - Network model element for load for DC formulations.

Builds the parameter $\underline{\boldsymbol{p}}$ and inherits from `mp.form_dc`.

Method Summary

build_params(*nm*, *dm*)

mp.nme_shunt

class mp.**nme_shunt**

Bases: mp.nm_element

mp.nme_shunt - 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

mp.nme_shunt_ac - Network model element abstract base class for shunt for AC formulations.

Builds the parameter $\underline{\mathbf{Y}}$.

Method Summary

build_params(*nm*, *dm*)

mp.nme_shunt_acc

class mp.**nme_shunt_acc**

Bases: mp.nme_shunt_ac, mp.form_acc

mp.nme_shunt_acc - Network model element for shunt for AC cartesian voltage formulations.

Inherits from mp.form_acc.

mp.nme_shunt_acp

class mp.**nme_shunt_acp**

Bases: mp.nme_shunt_ac, mp.form_acp

mp.nme_shunt_acp - Network model element for shunt for AC polar voltage formulations.

Inherits from mp.form_acp.

mp.nme_shunt_dc

`class mp.nme_shunt_dc`

Bases: `mp.nme_shunt`, `mp.form_dc`

`mp.nme_shunt_dc` - Network model element for shunt for DC formulations.

Builds the parameter $\underline{p}$ and inherits from `mp.form_dc`.

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`, `opt_model`

`mp.math_model` - 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`) objects and it is also an MP-Opt-Model (opt_model) object. All math model classes inherit from `mp.math_model` and therefore also from `mp.element_container`, 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` - auxiliary data relevant to the model

mp.math_model Methods:

- `task_tag()` - returns task tag, e.g. `'PF'`, `'OPF'`
- `task_name()` - returns task name, e.g. `'Power Flow'`, `'Optimal Power Flow'`
- `form_tag()` - returns network formulation tag, e.g. `'dc'`, `'acps'`
- `form_name()` - returns network formulation name, e.g. `'DC'`, `'AC-polar-power'`
- `build()` - create, add, and build math model element objects
- `display()` - display the math model object
- `add_aux_data()` - builds auxiliary data and adds it to the model
- `build_base_aux_data()` - builds base auxiliary data, including node types & variable initial values

- `add_vars()` - add variables to the model
- `add_system_vars()` - add system variables to the model
- `add_constraints()` - add constraints to the model
- `add_system_constraints()` - add system constraints to the model
- `add_node_balance_constraints()` - add node balance constraints to the model
- `add_costs()` - add costs to the model
- `add_system_costs()` - add system costs to the model
- `solve_opts()` - return an options struct to pass to the solver
- `update_nm_vars()` - update network model variables from math model solution
- `data_model_update()` - update data model from math model solution
- `network_model_x_soln()` - 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`, `mp.data_model`, `mp.net_model`.

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*, *mpopt*)

Create, add, and `build()` math model element objects.

```
mm.build(nm, dm, mpopt);
```

Inputs

- **nm** (`mp.net_model`) – network model object
- **dm** (`mp.data_model`) – 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 retuns 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`) – network model object
- **dm** (`mp.data_model`) – 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()`, is defined in `mp.mm_shared_pfcpf` (and in `mp.math_model_opf`) 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`) – network model object
- **dm** (`mp.data_model`) – 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system variables, then calls the `add_vars()` 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) – network model object
- **dm** (mp.data_model) – 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()` method. Other variables can be added by `add_system_vars()`. 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) – network model object
- **dm** (mp.data_model) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system constraints, then calls the `add_constraints()` 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) – network model object
- **dm** (mp.data_model) – 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()` method. Other constraints can be added by `add_system_constraints()`. In this base class, it simply calls `add_node_balance_constraints()`.

`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) – network model object
- **dm** (mp.data_model) – 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) – network model object
- **dm** (mp.data_model) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Adds system costs, then calls the `add_costs()` 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`) – network model object
- **dm** (`mp.data_model`) – 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()` method. Other variables can be added by `add_system_costs()`. 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`) – network model object
- **dm** (`mp.data_model`) – 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`) – 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`) – network model object
- **dm** (`mp.data_model`) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Output
dm (`mp.data_model`) – updated data model object

Calls the `data_model_update()` 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`) – network model object

Output
nm (`mp.net_model`) – updated network model object

Calls convert_x_m2n() to which is defined in a subclass of in `mp.mm_shared_pfcpf` (and of `mp.math_model_opf`) 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`

`mp.math_model_pf` - 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`

`mp.math_model_pf_ac` - 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`, `mp.mm_shared_pfcpf_acci`

`mp.math_model_pf_acci` - 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`, `mp.mm_shared_pfcpf_accs`

`mp.math_model_pf_accs` - 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`, `mp.mm_shared_pfcpf_acpi`

`mp.math_model_pf_acpi` - 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`, `mp.mm_shared_pfcpf_acps`

`mp.math_model_pf_acps` - 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`, `mp.mm_shared_pfcpf_dc`

`mp.math_model_pf_dc` - 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()` 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` - 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`, `mp.mm_shared_pfcpf_acci`

`mp.math_model_cpf_acci` - 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`, `mp.mm_shared_pfcpf_accs`

`mp.math_model_cpf_accs` - 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` - 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`, `mp.mm_shared_pfcpf_acpi`

`mp.math_model_cpf_acpi` - 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`, `mp.mm_shared_pfcpf_acps`

`mp.math_model_cpf_acps` - 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`

`mp.math_model_opf` - 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(mm, nm, dm, x0)`

`interior_va(nm, dm)`

mp.math_model_opf_ac

class `mp.math_model_opf_ac`

Bases: `mp.math_model_opf`

`mp.math_model_opf_ac` - 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`

`mp.math_model_opf_acc` - Abstract base class for AC cartesian OPF **math model** objects.

Provides formulation-specific and OPF-specific subclasses for elements.

Implements `convert_x_m2n()` 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`

`mp.math_model_opf_acci` - 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`, `mp.mm_shared_opf_legacy`

`mp.math_model_opf_acci_legacy` - OPF **math model** for AC-cartesian-current formulation w/legacy extensions.

Provides formluation-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`

`mp.math_model_opf_accs` - 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`, `mp.mm_shared_opf_legacy`

`mp.math_model_opf_accs_legacy` - OPF **math model** for AC-cartesian-power formulation w/legacy extensions.

Provides formluation-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`

`mp.math_model_opf_acp` - Abstract base class for AC polar OPF **math model** objects.

Provides formulation-specific and OPF-specific subclasses for elements.

Implements `convert_x_m2n()` 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`

`mp.math_model_opf_acpi` - 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`, `mp.mm_shared_opf_legacy`

`mp.math_model_opf_acpi_legacy` - OPF **math model** for AC-polar-current formulation w/legacy extensions.

Provides formluation-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`

`mp.math_model_opf_acps` - 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`, `mp.mm_shared_opf_legacy`

`mp.math_model_opf_acps_legacy` - OPF **math model** for AC-polar-power formulation w/legacy extensions.

Provides formluation-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`

`mp.math_model_opf_dc` - 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()` 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`, `mp.mm_shared_opf_legacy`

`mp.math_model_opf_dc` - OPF **math model** for DC formulation w/legacy extensions.

Provides formluation-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` - 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`

`mp.mm_shared_pfcpf_ac` - 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()` - 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` - 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 formuation.

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`

`mp.mm_shared_pfcpf_acc` - 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 formuation.

Method Summary

convert_x_m2n(*mmx*, *nm*, *only_v*)

`convert_x_m2n()` - 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`, `mp.mm_shared_pfcpf_ac_i`

`mp.mm_shared_pfcpf_acci` - 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 formuation.

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

mp.mm_shared_pfcpf_accs - 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 formuation.

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

mp.mm_shared_pfcpf_acp - 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 formuation.

Method Summary

convert_x_m2n(*mmx*, *nm*, *only_v*)

convert_x_m2n() - 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, mp.mm_shared_pfcpf_ac_i

mp.mm_shared_pfcpf_acpi - 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 formuation.

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`

`mp.mm_shared_pfcpf_acps` - 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 formuation.

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`

`mp.mm_shared_pfcpf_dc` - 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()` - 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()` - 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 - 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

def_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() - 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() - 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() - 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` - 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`. 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()` - get name of element type, e.g. `'bus'`, `'gen'`
- `data_model_element()` - get corresponding data model element
- `network_model_element()` - get corresponding network model element
- `add_vars()` - add math model variables for this element
- `add_constraints()` - add math model constraints for this element
- `add_costs()` - add math model costs for this element
- `data_model_update()` - update the corresponding data model element
- `data_model_update_off()` - update offline elements in corresponding data model element
- `data_model_update_on()` - 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`.

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, name)
```

Inputs

- **dm** (mp.data_model) – data model object
- **name** (*char array*) – name of element type

Output

dme (mp.dm_element) – data model element object

network_model_element(*nm*, *name*)

Get corresponding network model element.

```
nme = mme.network_model_element(nm, name)
```

Inputs

- **nm** (mp.net_model) – network model object
- **name** (*char array*) – name of element type

Output

nme (mp.nm_element) – 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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – 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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – 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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – 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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – data model object
- **mpopt** (*struct*) – MATPOWER options struct

Call `data_model_update_off()` then `data_model_update_on()` to update the data model for this element based on the math model solution.

See also `data_model_update_off()`, `data_model_update_on()`.

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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – 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()`.

See also `data_model_update()`, `data_model_update_on()`.

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) – mathmatical model object
- **nm** (mp.net_model) – network model object
- **dm** (mp.data_model) – 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()`, `data_model_update_off()`.

mp.mme_branch

class `mp.mme_branch`

Bases: `mp.mm_element`

`mp.mme_branch` - 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`

`mp.mme_branch_pf_ac` - 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`

`mp.mme_branch_pf_dc` - 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`

`mp.mme_branch_opf` - 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`

`mp.mme_branch_opf_ac` - 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`

`mp.mme_branch_opf_acc` - 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`

`mp.mme_branch_opf_acp` - 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`

`mp.mme_branch_opf_dc` - 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`

`mp.mme_bus` - 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`

`mp.mme_bus_pf_ac` - 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`

`mp.mme_bus_pf_dc` - 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*, *mpopt*)

mp.mme_bus_opf_ac

class `mp.mme_bus_opf_ac`

Bases: `mp.mme_bus`

`mp.mme_bus_opf_ac` - 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`

`mp.mme_bus_opf_acc` - 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*, *mpopt*)

interior_x0(*mm*, *nm*, *dm*, *x0*)

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

mp.mme_bus_opf_acp

class `mp.mme_bus_opf_acp`

Bases: `mp.mme_bus_opf_ac`

`mp.mme_bus_opf_acp` - 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`

`mp.mme_bus_opf_dc` - 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

mp.mme_gen - 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

mp.mme_gen_pf_ac - 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

mp.mme_gen_pf_dc - 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`

`mp.mme_gen_opf` - 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` parameters with fields:

- `poly_p` - polynomial costs for active power, struct returned by `mp.cost_table.poly_params()`, 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 actve & reactive struct returned by `mp.cost_table.pwl_params()`, 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`

`mp.mme_gen_opf_ac` - 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()`

has_pq_cap(*gen*, *upper_lower*)
from legacy `hasPQcap()`

disp_load_constant_pf_constraint(*dm*)
from legacy `makeAvl()`

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`

`mp.mme_gen_opf_dc` - 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`

`mp.mme_load` - 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`

`mp.mme_load_pf_ac` - 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*, *mpopt*)

mp.mme_load_pf_dc

class mp.**mme_load_pf_dc**

Bases: `mp.mme_load`

`mp.mme_load_pf_dc` - 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*, *mpopt*)

mp.mme_load_cpf

class mp.**mme_load_cpf**

Bases: `mp.mme_load_pf_ac`

`mp.mme_load_cpf` - 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`

`mp.mme_shunt` - 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`

`mp.mme_shunt_pf_ac` - 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`

`mp.mme_shunt_pf_dc` - 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`

`mp.mme_shunt_cpf` - 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 155) - 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 156) - construct object
- *istable()* (page 156) - true for *mp_table* (page 155) objects
- *size()* (page 156) - dimensions of table
- *isempty()* (page 156) - true if table has no columns or no rows
- *end()* (page 156) - used to index last row or variable/column
- *subsref()* (page 156) - indexing a table to retrieve data
- *subsasgn()* (page 157) - indexing a table to assign data
- *horzcat()* (page 157) - concatenate tables horizontally
- *vertcat()* (page 157) - concatenate tables vertically

- `display()` (page 158) - 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 156) 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>
```

(continues on next page)

(continued from previous page)

```
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
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 retuns 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 158) - Class that acts like a table but isn't one.

Addresses two issues with inheriting from **table** classes (`table`) or *mp_table* (page 155)).

1. In MATLAB, `table` is a sealed class, so you cannot inherit from it. You can, however, use a subclass of *mp_table* (page 155), 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 155) in Octave (at least up through 8.4.0), you cannot nest a subclass of *mp_table* (page 155) inside another *mp_table* (page 155) 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 155) 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 155), listed below) to the contained table object.

The class of the contained table object is either `table` or *mp_table* (page 155) and is determined by *mp_table_class()* (page 6).

Limitations

1. The Octave bug mentioned above also affects tables that inherit from *mp_table_subclass* (page 158). That is, such tables can be nested inside tables of type `table` or *mp_table* (page 155), but not inside tables that are or inherit from *mp_table_subclass* (page 158).
2. In MATLAB, when nesting an *mp_table_subclass* (page 158) object within another *mp_table_subclass* (page 158) 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 155).

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 159) - return the table stored in `tab`
- *set_table()* (page 159) - assign a table to `tab`
- `istable()` - true for *mp_table* (page 155) 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 155), *mp_table_class()* (page 6).

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`

`mp.cost_table` - 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` where that syntax is available, making the code more readable.

mp.cost_table Methods:

- `cost_table()` - construct object
- `poly_params()` - create struct of polynomial parameters from `mp.cost_table`
- `pwl_params()` - create struct of piecewise linear parameters from `mp.cost_table`
- `max_pwl_cost()` - get maximum cost component used to specify pwl costs

An `mp.cost_table` has the following columns:

Name	Type	Description
`poly_n`	*integer*	n_{poly}, number of coefficients in polynomial cost curve, $f_{\text{poly}}(x) = c_0 + c_1 x ... + c_N x^N$, where $n_{\text{poly}} = N + 1$
`poly_co`	*double*	matrix of coefficients c_j, of polynomial cost $f_{\text{poly}}(x)$, where c_j is found in column $j+1$
`pwl_n`	*double*	n_{pwl}, number of data points $(x_1, f_1), (x_2, f_2), ..., (x_N, f_N)$ defining a piecewise linear cost curve, $f_{\text{pwl}}(x)$ where $N = n_{\text{pwl}}$
`pwl_qty`	*double*	matrix of *quantiy* coordinates x_j for piecwise linear cost $f_{\text{pwl}}(x)$, where x_j is found in column j
`pwl_cos`	*double*	matrix of *cost* coordinates f_j for piecwise linear cost $f_{\text{pwl}}(x)$, where f_j is found in column j

See also `mp.cost_table_utils`, `mp_table_subclass`.

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`) – the cost table object

Method Summary

`poly_params`(*idx*, *pu_base*)

```
p = poly_params(obj, idx, pu_base)
```

Inputs
- **obj** (`mp.cost_table`) – 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 polynmial 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()`.

`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`) – 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()`.

`max_pwl_cost`()

```
maxc = max_pwl_cost(obj)
```

Input

obj (`mp.cost_table`) – 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()`.

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` - Static methods for `mp.cost_table`.

Contains the implementation of some methods that would ideally belong in `mp.cost_table`.

Within classes that inherit from `mp_table_subclass`, such as `mp.cost_table`, 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` methods that **do** allow access to that table via normal table subscripting syntax.

mp.cost_table_util Methods:

- `poly_params()` - create struct of polynomial parameters from `mp.cost_table`
- `pwl_params()` - create struct of piecewise linear parameters from `mp.cost_table`
- `max_pwl_cost()` - get maximum cost component used to specify pwl costs

See also `mp.cost_table`.

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()`. See `mp.cost_table.poly_params()` 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()`. See `mp.cost_table.pwl_params()` for details.

static max_pwl_cost(*cost*)

```
maxc = mp.cost_table_utils.max_pwl_cost(cost)
```

Implementation for `mp.cost_table.max_pwl_cost()`. See `mp.cost_table.max_pwl_cost()` for details.

3.6.5 mp.element_container

class `mp.element_container`

Bases: `handle`

`mp.element_container` - 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` - cell array of element constructors
- `elements` - a `mp.mapped_array` to hold the element objects

mp.element_container Methods:

- `modify_element_classes()` - modify an existing set of element constructors

See also `mp.mapped_array`.

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`) 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 - 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');

% check for named element
ma.has_name('color');
```

mp.mapped_array Methods:

- mapped_array() - constructor
- copy() - create a duplicate of the mapped array object
- length() - return number of elements in mapped array
- size() - return dimensions of mapped array
- add_names() - add or modify names of elements
- add_elements() - append elements to the end of the mapped array
- delete_elements() - delete elements from the mapped array
- has_name() - return true if the name exists in the mapped array
- name2idx() - return the index corresponding to a name
- subsref() - called when indexing a mapped array to retrieve data
- subsasgn() - called when indexing a mapped array to assign data
- display() - 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()`.

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()`.

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 corrsponding 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 retuns an object of this class.

3.6.7 mp.NODE_TYPE

class `mp.NODE_TYPE`

`mp.NODE_TYPE` - Defines enumerated type for node types.

mp.NODE_TYPE Properties:

- `PQ` - PQ node (= 1)
- `PV` - PV node (= 2)
- `REF` - reference node (= 3)
- `NONE` - isolated node (= 4)

mp.NODE_TYPE Methods:

- `is_valid()` - 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`.

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

3.7 MATPOWER Extension Classes

3.7.1 Base

mp.extension

class mp.extension

Bases: `handle`

`mp.extension` - 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()` - 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()` - return element class modifiers for data model converter elements
- `dm_element_classes()` - return element class modifiers for data model elements
- `nm_element_classes()` - return element class modifiers for network model elements
- `mm_element_classes()` - 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` - adds fixed zonal reserves to OPF
- `mp.xt_3p` - adds example prototype unbalanced three-phase elements for AC PF, CPF, and OPF

See also `mp.task`, `mp.dm_converter`, `mp.data_model`, `mp.net_model`, `mp.math_model`, `mp.dmc_element`, `mp.dm_element`, `mp.nm_element`, `mp.mm_element`.

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
dmc_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`

`mp.xt_reserves` - 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()` - add two classes to data model converter elements
- `dm_element_classes()` - add two classes to data model elements
- `mm_element_classes()` - 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`.

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`
- `mp.dmce_reserve_zone_mpc2`

`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`
- `mp.dme_reserve_zone`

`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`
- `mp.mme_reserve_zone`

Other classes belonging to `mp.xt_reserves` extension:

mp.dmce_reserve_gen_mpc2

class mp.**dmce_reserve_gen_mpc2**

Bases: `mp.dmc_element`

`mp.dmce_reserve_gen_mpc2` - 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`

`mp.dmce_reserve_zone_mpc2` - 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`, `mp.dme_shared_opf`

`mp.dme_reserve_gen` - 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
`gen`	*integer*	ID (`uid`) of corresponding generator
`cost`	*double*	reserve cost *(u/MW)*[1]
`qty`	*double*	available reserve quantity *(MW)*
`ramp10`	*double*	10-minute ramp rate *(MW)*
`r`	*double*	r, reserve allocation *(MW)*
`r_lb`	*double*	lower bound on reserve allocation *(MW)*
`r_ub`	*double*	upper bound on reserve allocation *(MW)*
`total_cost`	*double*	total cost of allocated reserves *(u)*[1]
`prc`	*double*	reserve price *(u/MVAr)*[1]
`mu_lb`	*double*	shadow price on r lower bound *(u/MW)*[1]
`mu_ub`	*double*	shadow price on r upper bound *(u/MW)*[1]
`mu_pg_ub`	*double*	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(*mpopt*, *pp_args*)

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

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

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*)

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*)

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

mp.dme_reserve_zone

class `mp.`**`dme_reserve_zone`**

Bases: `mp.dm_element`, `mp.dme_shared_opf`

`mp.dme_reserve_zone` - 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
`req`	*double*	zonal reserve requirement *(MW)*
`zones`	*integer*	matrix defining generators included in the zone
`prc`	*double*	zonal reserve price *(u/MW)*[1]

Property Summary

zones

zone map for online `zones` / 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(*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.mme_reserve_gen

class mp.**mme_reserve_gen**

Bases: `mp.mm_element`

`mp.mme_reserve_gen` - 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*, *mpopt*)

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

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

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

mp.mme_reserve_zone

class mp.**mme_reserve_zone**

Bases: `mp.mm_element`

`mp.mme_reserve_zone` - 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*, *mpopt*)

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

3.7.3 Three-Phase Prototype Extension

mp.xt_3p

class mp.**xt_3p**

Bases: `mp.extension`

`mp.xt_3p` - MATPOWER extension to add unbalanced three-phase elements.

For AC power flow, continuation power flow, and optimial 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()` - add six classes to data model converter elements
- `dm_element_classes()` - add six classes to data model elements
- `nm_element_classes()` - add six classes to network model elements
- `mm_element_classes()` - 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`.

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`
- `mp.dmce_gen3p_mpc2`
- `mp.dmce_load3p_mpc2`
- `mp.dmce_line3p_mpc2`
- `mp.dmce_xfmr3p_mpc2`

- mp.dmce_buslink_mpc2

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
- mp.dme_gen3p
- mp.dme_load3p
- mp.dme_line3p
- mp.dme_xfmr3p
- mp.dme_buslink

For 'OPF' tasks, adds the classes:

- mp.dme_bus3p_opf
- mp.dme_gen3p_opf
- mp.dme_load3p_opf
- mp.dme_line3p_opf
- mp.dme_xfmr3p_opf
- mp.dme_buslink_opf

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
- mp.nme_gen3p_acc
- mp.nme_load3p
- mp.nme_line3p
- mp.nme_xfmr3p
- mp.nme_buslink_acc

For *polar* voltage formulations, adds the classes:

- mp.nme_bus3p_acp
- mp.nme_gen3p_acp
- mp.nme_load3p
- mp.nme_line3p
- mp.nme_xfmr3p
- mp.nme_buslink_acp

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
- mp.mme_gen3p
- mp.mme_line3p
- mp.mme_xfmr3p
- mp.mme_buslink_pf_acc *(cartesian)* or mp.mme_buslink_pf_acp *(polar)*

For 'OPF' tasks, adds the classes:

- mp.mme_bus3p_opf_acc *(cartesian)* or mp.mme_bus3p_opf_acp *(polar)*
- mp.mme_gen3p_opf
- mp.mme_line3p_opf
- mp.mme_xfmr3p_opf
- mp.mme_buslink_opf_acc *(cartesian)* or mp.mme_buslink_opf_acp *(polar)*

Data model converter element classes belonging to mp.xt_3p extension:

mp.dmce_bus3p_mpc2

class `mp.dmce_bus3p_mpc2`

Bases: `mp.dmc_element`

`mp.dmce_bus3p_mpc2` - 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`

`mp.dmce_gen3p_mpc2` - 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`

`mp.dmce_load3p_mpc2` - 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`

`mp.dmce_line3p_mpc2` - 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`

`mp.dmce_xfmr3p_mpc2` - 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`

`mp.dmce_buslink_mpc2` - 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` extension:

mp.dme_bus3p

class mp.**dme_bus3p**

Bases: `mp.dm_element`

`mp.dme_bus3p` - 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`	*integer*	bus type (1 = PQ, 2 = PV, 3 = ref, 4 = isolated)
`base_kv`	*double*	base voltage *(kV)*
`vm1`	*double*	phase 1 voltage magnitude *(p.u.)*
`vm2`	*double*	phase 2 voltage magnitude *(p.u.)*
`vm3`	*double*	phase 3 voltage magnitude *(p.u.)*
`va1`	*double*	phase 1 voltage angle *(degrees)*
`va2`	*double*	phase 2 voltage angle *(degrees)*
`va3`	*double*	phase 3 voltage angle *(degrees)*

Property Summary

type
node `type` 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(*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_gen3p

class mp.**dme_gen3p**

Bases: `mp.dm_element`

`mp.dme_gen3p` - 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`	*integer*	bus ID (`uid`) of 3-phase bus
`vm1_setpoint`	*double*	phase 1 voltage magnitude setpoint *(p.u.)*
`vm2_setpoint`	*double*	phase 2 voltage magnitude setpoint *(p.u.)*
`vm3_setpoint`	*double*	phase 3 voltage magnitude setpoint *(p.u.)*
`pg1`	*double*	phase 1 active power output *(kW)*
`pg2`	*double*	phase 2 active power output *(kW)*
`pg3`	*double*	phase 3 active power output *(kW)*
`qg1`	*double*	phase 1 reactive power output *(kVAr)*
`qg2`	*double*	phase 2 reactive power output *(kVAr)*
`qg3`	*double*	phase 3 reactive power output *(kVAr)*

Property Summary

bus

`bus` 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(*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_load3p

class mp.**dme_load3p**

Bases: `mp.dm_element`

`mp.dme_load3p` - 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`	*integer*	bus ID (`uid`) of 3-phase bus
`pd1`	*double*	phase 1 active power demand *(kW)*
`pd2`	*double*	phase 2 active power demand *(kW)*
`pd3`	*double*	phase 3 active power demand *(kW)*
`pf1`	*double*	phase 1 power factor
`pf2`	*double*	phase 2 power factor
`pf3`	*double*	phase 3 power factor

Property Summary

bus

`bus` 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(*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_line3p

class mp.**dme_line3p**

Bases: `mp.dm_element`

`mp.dme_line3p` - 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 *(?)*
`pl1_fr`	*double*	phase 1 active power injection at "from" end *(kW)*
`ql1_fr`	*double*	phase 1 reactive power injection at "from" end *(kVAr)*
`pl2_fr`	*double*	phase 2 active power injection at "from" end *(kW)*
`ql2_fr`	*double*	phase 2 reactive power injection at "from" end *(kVAr)*
`pl3_fr`	*double*	phase 3 active power injection at "from" end *(kW)*
`ql3_fr`	*double*	phase 3 reactive power injection at "from" end *(kVAr)*
`pl1_to`	*double*	phase 1 active power injection at "to" end *(kW)*
`ql1_to`	*double*	phase 1 reactive power injection at "to" end *(kVAr)*
`pl2_to`	*double*	phase 2 active power injection at "to" end *(kW)*
`ql2_to`	*double*	phase 2 reactive power injection at "to" end *(kVAr)*
`pl3_to`	*double*	phase 3 active power injection at "to" end *(kW)*
`ql3_to`	*double*	phase 3 reactive power injection at "to" end *(kVAr)*

The line construction table in the `lc_tab` property is defined as a table with the following columns:

Name	Type	Description
`id`	*integer*	unique line construction ID, referenced from `lc` column of main data table
`r`	*double*	6 resistence parameters for forming symmetric 3x3 series impedance matrix
`x`	*double*	6 reactance parameters for forming symmetric 3x3 series impedance matrix
`c`	*double*	6 susceptance parameters for forming symmetric 3x3 shunt susceptance matrix

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

`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`

`mp.dme_xfmr3p` - 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
`bus_fr`	*integer*	bus ID (`uid`) of "from" 3-phase bus
`bus_to`	*integer*	bus ID (`uid`) of "to" 3-phase bus
`r`	*double*	series resistance *(p.u.)*
`x`	*double*	series reactance *(p.u.)*
`base_kva`	*double*	transformer kVA base *(kVA)*
`base_kv`	*double*	transformer kV base *(kV)*
`pl1_fr`	*double*	phase 1 active power injection at "from" end *(kW)*
`ql1_fr`	*double*	phase 1 reactive power injection at "from" end *(kVAr)*
`pl2_fr`	*double*	phase 2 active power injection at "from" end *(kW)*
`ql2_fr`	*double*	phase 2 reactive power injection at "from" end *(kVAr)*
`pl3_fr`	*double*	phase 3 active power injection at "from" end *(kW)*
`ql3_fr`	*double*	phase 3 reactive power injection at "from" end *(kVAr)*
`pl1_to`	*double*	phase 1 active power injection at "to" end *(kW)*
`ql1_to`	*double*	phase 1 reactive power injection at "to" end *(kVAr)*
`pl2_to`	*double*	phase 2 active power injection at "to" end *(kW)*
`ql2_to`	*double*	phase 2 reactive power injection at "to" end *(kVAr)*
`pl3_to`	*double*	phase 3 active power injection at "to" end *(kW)*
`ql3_to`	*double*	phase 3 reactive power injection at "to" end *(kVAr)*

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()`

`cxn_type()`

`cxn_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(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_buslink

class `mp.dme_buslink`

Bases: `mp.dm_element`

`mp.dme_buslink` - 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	*integer*	bus ID (uid) of single phase bus
bus3p	*integer*	bus ID (uid) of 3-phase bus

Property Summary

bus
bus index vector (all buslinks)

bus3p
bus3p 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(*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_bus3p_opf

class `mp.dme_bus3p_opf`

Bases: `mp.dme_bus3p`, `mp.dme_shared_opf`

`mp.dme_bus3p_opf` - Data model element for 3-phase bus for OPF.

To parent class `mp.dme_bus3p`, adds pretty-printing for **lim** sections.

mp.dme_gen3p_opf

class `mp.dme_gen3p_opf`

Bases: `mp.dme_gen3p`, `mp.dme_shared_opf`

`mp.dme_gen3p_opf` - Data model element for 3-phase generator for OPF.

To parent class `mp.dme_gen3p`, adds pretty-printing for **lim** sections.

mp.dme_load3p_opf

class `mp.dme_load3p_opf`

Bases: `mp.dme_load3p`, `mp.dme_shared_opf`

`mp.dme_load3p_opf` - Data model element for 3-phase load for OPF.

To parent class `mp.dme_load3p`, adds pretty-printing for **lim** sections.

mp.dme_line3p_opf

class `mp.dme_line3p_opf`

Bases: `mp.dme_line3p`, `mp.dme_shared_opf`

`mp.dme_line3p_opf` - Data model element for 3-phase line for OPF.

To parent class `mp.dme_line3p`, adds pretty-printing for **lim** sections.

mp.dme_xfmr3p_opf

class `mp.dme_xfmr3p_opf`

Bases: `mp.dme_xfmr3p`, `mp.dme_shared_opf`

`mp.dme_xfmr3p_opf` - Data model element for 3-phase transformer for OPF.

To parent class `mp.dme_xfmr3p`, adds pretty-printing for **lim** sections.

mp.dme_buslink_opf

class mp.**dme_buslink_opf**

Bases: `mp.dme_buslink`, `mp.dme_shared_opf`

`mp.dme_buslink_opf` - Data model element for 1-to-3-phase buslink for OPF.

To parent class `mp.dme_buslink`, adds pretty-printing for **lim** sections.

Network model element classes belonging to `mp.xt_3p` extension:

mp.nme_bus3p

class mp.**nme_bus3p**

Bases: `mp.nm_element`

`mp.nme_bus3p` - 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()` method.

Method Summary

name()

nn()

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()` method of `mp.net_model`.

mp.nme_bus3p_acc

class mp.**nme_bus3p_acc**

Bases: `mp.nme_bus3p`, `mp.form_acc`

`mp.nme_bus3p_acc` - 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`.

Method Summary

add_vvars(*nm*, *dm*, *idx*)

mp.nme_bus3p_acp

class mp.**nme_bus3p_acp**

Bases: `mp.nme_bus3p`, `mp.form_acp`

`mp.nme_bus3p_acp` - 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`.

Method Summary

add_vvars(*nm*, *dm*, *idx*)

mp.nme_gen3p

class mp.**nme_gen3p**

Bases: `mp.nm_element`

`mp.nme_gen3p` - 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 $\underline{\mathbf{N}}$.

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`, `mp.form_acc`

`mp.nme_gen3p_acc` - Network model element for 3-phase generator, AC cartesian voltage formulation.

Inherits from `mp.form_acc`.

mp.nme_gen3p_acp

class mp.**nme_gen3p_acp**

Bases: `mp.nme_gen3p`, `mp.form_acp`

`mp.nme_gen3p_acp` - Network model element for 3-phase generator, AC polar voltage formulation.

Inherits from `mp.form_acp`.

mp.nme_load3p

class mp.**nme_load3p**

Bases: `mp.nm_element`, `mp.form_acp`

`mp.nme_load3p` - 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 $\underline{s}$ and inherits from `mp.form_acp`.

Method Summary

name()

np()

build_params(*nm*, *dm*)

mp.nme_line3p

class mp.**nme_line3p**

Bases: `mp.nm_element`, `mp.form_acp`

`mp.nme_line3p` - 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 $\underline{\mathbf{Y}}$ for 3-phase lines and inherits from `mp.form_acp`.

Method Summary

name()

np()

build_params(*nm*, *dm*)

vec2symmat_stacked(*vv*)

mp.nme_xfmr3p

class mp.**nme_xfmr3p**

Bases: `mp.nm_element`, `mp.form_acp`

`mp.nme_xfmr3p` - 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 $\underline{\mathbf{Y}}$ for 3-phase transformers and inherits from `mp.form_acp`.

Method Summary

name()

np()

build_params(*nm*, *dm*)

mp.nme_buslink

class mp.**nme_buslink**

Bases: `mp.nm_element`

`mp.nme_buslink` - 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 `Plink` and `Qlink` to the network model, builds the parameter $\underline{\mathbf{N}}$, 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, mp.form_acc

mp.nme_buslink_acc - Network model element for 1-to-3-phase buslink, AC cartesian voltage formulation.

Inherits from mp.form_acc.

mp.nme_buslink_acp

class mp.**nme_buslink_acp**

Bases: mp.nme_buslink, mp.form_acp

mp.nme_buslink_acp - Network model element for 1-to-3-phase buslink, AC polar voltage formulation.

Inherits from mp.form_acp.

Mathematical model element classes belonging to mp.xt_3p extension:

mp.mme_bus3p

class mp.**mme_bus3p**

Bases: mp.mm_element

mp.mme_bus3p - 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

mp.mme_gen3p - 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

mp.mme_line3p - 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

mp.mme_xfmr3p - 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

mp.mme_buslink - 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`

`mp.mme_buslink_pf_ac` - 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`

`mp.mme_buslink_pf_acc` - 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*, *mpopt*)

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`

`mp.mme_buslink_pf_acp` - 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*, *mpopt*)

mp.mme_bus3p_opf_acc

class `mp.mme_bus3p_opf_acc`

Bases: `mp.mme_bus3p`

`mp.mme_bus3p_opf_acc` - 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`

`mp.mme_bus3p_opf_acp` - 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`

`mp.mme_gen3p_opf` - 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`

`mp.mme_line3p_opf` - 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`

`mp.mme_xfmr3p_opf` - 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

mp.mme_buslink_opf - 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

mp.mme_buslink_opf_acc - 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*, *mpopt*)

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

mp.mme_buslink_opf_acp - 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*, *mpopt*)

3.7.4 Legacy DC Line Extension

mp.xt_legacy_dcline

class mp.**xt_legacy_dcline**

Bases: `mp.extension`

`mp.xt_legacy_dcline` - MATPOWER extension to add legacy DC line elements.

For AC and DC power flow, continuation power flow, and optimial 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()` - add a class to data model converter elements
- `dm_element_classes()` - add a class to data model elements
- `nm_element_classes()` - add a class to network model elements
- `mm_element_classes()` - 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`.

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`

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`

For `'OPF'` tasks, adds the class:
- `mp.dme_legacy_dcline_opf`

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`

For AC *cartesian* voltage formulations, adds the class:
- `mp.nme_legacy_dcline_acc`

For AC *polar* voltage formulations, adds the class:
- `mp.nme_legacy_dcline_acp`

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` *(DC formulation)* or
- `mp.mme_legacy_dcline_pf_ac` *(AC formulation)*

For `'OPF'` tasks, adds the class:
- `mp.mme_legacy_dcline_opf_dc` *(DC formulation)* or
- `mp.mme_legacy_dcline_opf_ac` *(AC formulation)*

Data model converter element class belonging to `mp.xt_legacy_dcline` extension:

mp.dmce_legacy_dcline_mpc2

class mp.**dmce_legacy_dcline_mpc2**

Bases: `mp.dmc_element`

`mp.dmce_legacy_dcline_mpc2` - 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` extension:

mp.dme_legacy_dcline

class `mp.dme_legacy_dcline`

Bases: `mp.dm_element`

`mp.dme_legacy_dcline` - Data model element for legacy DC line.

Implements the data element model for legacy DC line elements, with linear line losses.

$$p_{\mathrm{loss}} = \underline{l}_0 + \underline{l}_1 p_{\mathrm{fr}}$$

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" bus
`bus_to`	*integer*	bus ID (`uid`) of "to" bus
`loss0`	*double*	$\underline{l}_0$, constant term of loss function (MW)
`loss1`	*double*	$\underline{l}_1$, linear coefficient of loss function (MW/MW)
`vm_setpoint_fr`	*double*	per unit "from" bus voltage magnitude setpoint
`vm_setpoint_to`	*double*	per unit "to" bus voltage magnitude setpoint
`p_fr_lb`	*double*	lower bound on MW flow at "from" port
`p_fr_ub`	*double*	upper bound on MW flow at "from" port
`q_fr_lb`	*double*	lower bound on MVAr injection into "from" bus
`q_fr_ub`	*double*	upper bound on MVAr injection into "from" bus
`q_to_lb`	*double*	lower bound on MVAr injection into "to" bus
`q_to_ub`	*double*	upper bound on MVAr injection into "to" bus
`p_fr`	*double*	MW flow at "from" end ("from" –> "to")
`q_fr`	*double*	MVAr injection into "from" bus
`p_to`	*double*	MW flow at "to" end ("from" –> "to")
`q_to`	*double*	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(*mpopt*, *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`, `mp.dme_shared_opf`

`mp.dme_legacy_dcline_opf` - Data model element for legacy DC line for OPF.

To parent class `mp.dme_legacy_dcline`, 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
`cost_pg`	*mp.cost_table*	cost of active power flow *(u/MW)*[1]
`mu_p_fr_lb`	*double*	shadow price on MW flow lower bound at "from" end *(u/MW)*[1]
`mu_p_fr_ub`	*double*	shadow price on MW flow upper bound at "from" end *(u/MW)*[1]
`mu_q_fr_lb`	*double*	shadow price on lower bound of MVAr injection at "from" bus *(u/degree)*[1]
`mu_q_fr_ub`	*double*	shadow price on upper bound of MVAr injection at "from" bus *(u/degree)*[1]
`mu_q_to_lb`	*double*	shadow price on lower bound of MVAr injection at "to" bus *(u/degree)*[1]
`mu_q_to_ub`	*double*	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(*mpopt*, *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*, *mpopt*, *pp_args*)

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

Network model element classes belonging to `mp.xt_legacy_dcline` extension:

mp.nme_legacy_dcline

class `mp.nme_legacy_dcline`

Bases: `mp.nm_element`

`mp.nme_legacy_dcline` - 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`

`mp.nme_legacy_dcline_ac` - 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 $\underline{\mathbf{N}}$.

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`, `mp.form_acc`

`mp.nme_legacy_dcline_acc` - Network model element for legacy DC line for for AC cartesian voltage formulations.

Inherits from `mp.form_acc`.

mp.nme_legacy_dcline_acp

class mp.**nme_legacy_dcline_acp**

Bases: `mp.nme_legacy_dcline_ac`, `mp.form_acp`

`mp.nme_legacy_dcline_acp` - Network model element for legacy DC line for for AC polar voltage formulations.

Inherits from `mp.form_acp`.

mp.nme_legacy_dcline_dc

class mp.**nme_legacy_dcline_dc**

Bases: `mp.nme_legacy_dcline`, `mp.form_dc`

`mp.nme_legacy_dcline_dc` - 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 $\underline{\boldsymbol{K}}$.

Method Summary

add_zvars(*nm*, *dm*, *idx*)

build_params(*nm*, *dm*)

Mathematical model element classes belonging to `mp.xt_legacy_dcline` extension:

mp.mme_legacy_dcline

class mp.**mme_legacy_dcline**

Bases: `mp.mm_element`

`mp.mme_legacy_dcline` - 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_ac**

Bases: `mp.mme_legacy_dcline`

`mp.mme_legacy_dcline_pf_ac` - 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_dc**

Bases: `mp.mme_legacy_dcline`

`mp.mme_legacy_dcline_pf_dc` - 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_opf**

Bases: `mp.mme_legacy_dcline`

`mp.mme_legacy_dcline_opf` - 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` 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 actve 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

mp.mme_legacy_dcline_opf_ac - 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

mp.mme_legacy_dcline_opf_dc - 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*)

Index

A

B

C

D

E

F

K

L

M

N

O

P

Q

R

S

X

Y

Z