MATPOWER

MATPOWER Developers's Manual

Release 8.1

Ray D. Zimmerman

July 12, 2025

Contents

1 Introduction **1**
- 1.1 Development Environment . . . 1
- 1.2 Conventions . . . 2

2 Notation **3**

3 Architecture Overview **5**
- 3.1 MATPOWER Object Instances . . . 6
- 3.2 MATPOWER Class Hierarchies . . . 7
- 3.3 Two MATPOWER Frameworks . . . 9
- 3.4 MATPOWER Customization . . . 9

4 Task Object **10**
- 4.1 Running a Task . . . 10
- 4.2 Building Model and Converter Objects . . . 10
- 4.3 Iterative Execution . . . 12
- 4.4 Other Methods . . . 12

5 Data Model Object **13**
- 5.1 Data Models . . . 13
- 5.2 Data Model Elements . . . 15

6 Data Model Converter Object **18**
- 6.1 Data Model Converters . . . 18
- 6.2 Data Model Converter Elements . . . 19

7 Network Model Object **22**
- 7.1 Network Model Formulations . . . 22
- 7.2 Network Models . . . 25
- 7.3 Network Model Elements . . . 26

8 Mathematical Model Object **30**
- 8.1 Mathematical Models . . . 31
- 8.2 Mathematical Model Elements . . . 32
- 8.3 Shared Classes . . . 34

9 Customizing MATPOWER **35**

9.1 Default Class Determination 35
9.2 Customization via MATPOWER Options 38
9.3 MATPOWER Extensions 38

10 Acknowledgments **40**

11 References **41**

Bibliography **42**

1 Introduction

The purpose of this *Developer's Manual* is to provide an understanding of the internal design of MATPOWER for users who wish to help with the development of MATPOWER or for those who would like to customize, modify or add to the functionality of MATPOWER in any way.

The MATPOWER User's Manual, on the other hand, is your starting point if you simply want to use MATPOWER without modification or customization.

For reference documentation on each class and function in MATPOWER, see the MATPOWER Reference Manual.

1.1 Development Environment

MATPOWER is implemented in the Matlab language, designed for scientific computing. It requires either MATLAB ®, a commercial product from The MathWorks, or the free, open-source GNU Octave to run.

MATPOWER and its related software packages are developed as open-source projects on GitHub under the MATPOWER Development GitHub organization. Some projects are included in others using git subrepo.

Table 1.1 provides an overview of the various repositories and their relationships to each other. Note that the main **matpower** repository contains all of the others as subrepos, except for **matpower-extras**, which is, however, included when you download the ZIP file for a numbered MATPOWER release.

Table 1.1: MATPOWER GitHub Repositories

Repository	Description
matpower	Main MATPOWER repository. Depends on **mptest** , **mips**, and **mp-opt-model**, which are included as subrepos, along with **most**, and **mp-docs-shared**.
mptest	Functions for implementing unit testing in MATLAB or Octave, with generalized mechanism for testing for optional functionality and corresponding versions, i.e. `have_feature()`. Required by all of the other projects.
mips	**M**ATPOWER **I**nterior **P**oint **S**olver (MIPS), a nonlinear primal-dual interior point solver used as the default solver for AC OPF problems. Also includes a wrapper function for several linear equation solvers. Depends on **mptest**.
mp-opt-model	MP-Opt-Model, an easy-to-use, object-oriented interface for building and solving mathematical programming and optimization problems. Also includes a unified interface for calling numerous LP, QP, mixed-integer and nonlinear solvers, with the ability to switch solvers simply by changing an input option. Depends on **mptest** and **mips**.

continues on next page

Table 1.1 – continued from previous page

Repository	Description
most	**M**ATPOWER **O**ptimal **S**cheduling **T**ool (MOST), a framework for solving generalized steady-state electric power scheduling problems. Depends on **mptest**, **mp-opt-model** and **matpower**.
matpower-extras	MATPOWER Extras, a collection of contributed and/or unsupported MATPOWER-related functions and packages. Note that some of the extras have their own separate repositories and are actually included here as subrepos. Depends on **mptest** and **matpower**.
mp-docs-shared	Defines common resources used for the Sphinx documentation and included as a subrepo in `docs/sphinx/source` in all of the projects.

In general, each repository has two permanent branches, `master` and `release`, where `release` points to the latest stable release and `master` contains any unreleased but hopefully stable updates. Each numbered release also has an associated git tag.

1.2 Conventions

Because MATPOWER is intended to run unmodified on either MATLAB or GNU Octave, it is important to stick to syntax and functionality that are supported by both.

We use `classdef` syntax supported by both to define classes and, in methods, we use `obj` as the variable name representing the object. Most of the classes are defined in the `mp` package/namespace.

All classes, methods, properties, and functions include a help section that can be accessed by the `help` and `doc` commands *and* processed by Sphinx to produce HTML and PDF reference documentation. For a class, it summarizes the purpose and overall functionality provided by the class along with lists of the properties and methods. For a function or method, it describes the inputs, outputs and what the function or method does. The `run_mp()` function and the `mp.task` class provide examples of this reference documentation. *Hint: Click the GitHub icon in the upper right corner of the reference manual page to see the source.*

All functionality should be covered by at least one of the automated tests.

See the MATPOWER Contributors Guide for more information on contributing to the MATPOWER project.

2 Notation

This section introduces and summarizes the mathematical notation used throughout this manual.

This notation is consistent with what was used in the MP-Element technical note, *MATPOWER Technical Note 5* [TN5] where you can find more detail.

Styles

x, θ	real scalars
$\mathrm{x}, \mathrm{\theta}$	complex scalars
$\boldsymbol{x}, \boldsymbol{\theta}$	real vectors
$\mathbf{x}, \mathbf{\theta}$	complex vectors
$\boldsymbol{X}, \boldsymbol{\Theta}$	real matrices
$\mathbf{X}, \mathbf{\Theta}$	complex matrices
$x, \mathrm{x}, \boldsymbol{x}, \mathbf{x}, \boldsymbol{X}, \mathbf{X}$	variables, functions
$\underline{x}, \underline{\mathrm{x}}, \underline{\boldsymbol{x}}, \underline{\mathbf{x}}, \underline{\boldsymbol{X}}, \underline{\mathbf{X}}$	constants, parameters[1]
$\hat{\boldsymbol{x}}, \hat{\mathbf{x}}, \hat{\boldsymbol{X}}, \hat{\mathbf{X}}$	selected rows of interest of $\boldsymbol{x}, \mathbf{x}, \boldsymbol{X}, \mathbf{X}$, respectively[2]

Operators

$\lceil \mathbf{a} \rfloor$	diagonal matrix with vector $\mathbf{a}$ on the diagonal
$\{\boldsymbol{A}\}_{\times n}$	the set $\{\boldsymbol{A}, \boldsymbol{A}, \ldots, \boldsymbol{A}\}$ where $\boldsymbol{A}$ is repeated n times
$\mathcal{A} = \{\boldsymbol{A}_i\}_{i=1}^{n}$	set of n matrices (or vectors) indexed by i
$\mathrm{diag}(\boldsymbol{A}) = [a_{11} \ \ldots \ a_{nn}]^{\mathsf{T}}$	matrix-to-vector diagonal operator
$\lceil \mathcal{A} \rfloor$	block diagonal matrix with the set $\mathcal{A}$ of matrices or vectors on the block diagonal
$\mathbf{A}^{\mathsf{T}}$	(non-conjugate) transpose of matrix $\mathbf{A}$
a^*, $\mathbf{a}^*$, $\mathbf{A}^*$	complex conjugate of a, $\mathbf{a}$, and $\mathbf{A}$, respectively
$\Re\{\mathbf{a}\}$, $\Im\{\mathbf{a}\}$	real and imaginary parts of $\mathbf{a}$, respectively
$\mathbf{a}^n$	element-wise exponent[3] for vector $\mathbf{a}$
$\mathbf{A}^n$	matrix exponent[Page 4, 3] for matrix $\mathbf{A}$
$a^{\boldsymbol{b}}$, $a^{\boldsymbol{B}}$	element-wise exponent[Page 4, 3] for vector $\boldsymbol{b}$ and matrix $\boldsymbol{B}$, respectively
$\mathrm{f}(\boldsymbol{x}), \mathbf{f}(\boldsymbol{x})$	scalar, vector functions of $\boldsymbol{x}$, respectively

continues on next page

[1] Constants and parameters are underlined, with the following exceptions: constants e and j, p, q, m and n when used as dimensions, and i, j, and k as indices.

[2] Obtained by multiplying by matrix $\underline{\boldsymbol{J}}$ or $\underline{\boldsymbol{J}}_{\boldsymbol{k}}$.

Table 2.2 – continued from previous page

$\mathrm{f}_{\boldsymbol{x}}, \mathbf{f}_{\boldsymbol{x}}$	transpose of gradient of f, Jacobian of $\mathbf{f}$, respectively, w.r.t. $\boldsymbol{x}$
$\mathrm{f}_{\boldsymbol{xx}}, \mathbf{f}_{\boldsymbol{xx}}(\boldsymbol{\lambda})$	Hessian of f, Jacobian of $\mathbf{f}_{\boldsymbol{x}}{}^{\mathsf{T}}\boldsymbol{\lambda}$, respectively, w.r.t. $\boldsymbol{x}$

Constants and Dimensions

e, j	constants, e is base of natural log (≈ 2.71828), j is $\sqrt{-1}$
n_k, n_n, n_p, n_p^k	number of elements, nodes, ports, ports for element k, respectively
$n_{\mathbf{x}}, n_{\mathbf{v}}, n_{\mathbf{z}}$	dimension of vector $\mathbf{x}$, $\mathbf{v}$, $\mathbf{z}$, respectively.
$\mathbf{1}_n, [\,\mathbf{1}_n\,]$	$n \times 1$ vector of all ones, $n \times n$ identity matrix
$\mathbf{0}$	appropriately-sized vector or matrix of all zeros

Variables

v_i	complex voltage at node/port i
u_i, w_i	real and imaginary parts of voltage at node/port i, $\mathrm{v}_i = u_i + jw_i$
ν_i, θ_i	voltage magnitude and angle at node/port i, $\mathrm{v}_i = \nu_i e^{j\theta_i}$
$\mathbf{v}$	column vector of complex voltages v_i
$\mathbf{e}$	column vector $\mathbf{v}$ with elements scaled to unit magnitude, $\mathbf{e} = e^{j\boldsymbol{\theta}}$
$\boldsymbol{u}, \boldsymbol{w}$	column vectors of real (u_i) and imaginary (w_i) parts of voltage, respectively, $\mathbf{v} = \boldsymbol{u} + j\boldsymbol{w}$
$\boldsymbol{\nu}, \boldsymbol{\theta}$	column vectors of voltage magnitudes ν_i and angles θ_i, respectively, $\mathbf{v} = [\,\boldsymbol{\nu}\,]\,\mathbf{e} = [\,\boldsymbol{\nu}\,]\,e^{j\boldsymbol{\theta}}$
$\boldsymbol{\Lambda}$	column vector of inverse of complex voltages $\frac{1}{\mathrm{v}_i}$, $\boldsymbol{\Lambda} = \mathbf{v}^{-1}$
$\boldsymbol{z}$	column vector of real non-voltage state variables z_i
$\mathbf{z}$	column vector of complex non-voltage state variables z_i
$\boldsymbol{z}_r, \boldsymbol{z}_i$	column vectors of real and imaginary parts of $\mathbf{z} = \boldsymbol{z}_r + j\boldsymbol{z}_i$

Parameters

$\underline{\boldsymbol{J}_{\boldsymbol{k}}}$	matrix formed by taking selected rows, indexed by vector $\boldsymbol{k}$, from an identity matrix[4]
$\underline{\mathbf{Y}}$	AC model admittance matrix
$\underline{\mathbf{L}}$	linear coefficient (of $\mathbf{z}$) for affine complex current injections
$\underline{\mathbf{i}}$	vector of constant complex current injections
$\underline{\mathbf{M}}$	linear coefficient (of $\mathbf{v}$) for affine complex power injections
$\underline{\mathbf{N}}$	linear coefficient (of $\mathbf{z}$) for affine complex power injections
$\underline{\mathbf{s}}$	vector of constant complex power injections
$\underline{\boldsymbol{B}}$	DC model susceptance matrix
$\underline{\boldsymbol{K}}$	linear coefficient (of $\boldsymbol{z}$) for affine active power injections
$\underline{\boldsymbol{p}}$	vector of constant active power injections
$\underline{\boldsymbol{C}}$	element-node incidence matrix for a given port
$\underline{\boldsymbol{D}}$	element-variable incidence matrix for a given state variable
$\underline{\boldsymbol{A}}$	combined incidence matrix $\underline{\boldsymbol{A}} = \begin{bmatrix} \underline{\boldsymbol{C}} & \mathbf{0} \\ \mathbf{0} & \underline{\boldsymbol{D}} \end{bmatrix}$

[3] Superscripts may also be used as indices, indicated by context.
[4] Often used simply as $\underline{\boldsymbol{J}}$ without the subscript.

3 Architecture Overview

A new *object-oriented MATPOWER core architecture* (MP-Core), designed around the concept of a generic system **element**,[1] was introduced in MATPOWER 8.0, along with two frameworks for employing this new MP-Core in MATPOWER. This chapter gives an overview of this architecture.

MATPOWER's primary function is to solve steady-state electric power system simulation and optimization problems, such as power flow, continuation power flow and optimal power flow. At the top level of MP-Core is a **task** object that constructs the various layers of modeling for the desired problem type and formulation, solves the problem, and propogates the solution back through the modeling layers to the user.

This architecture employs an explicit three-layer modeling structure designed to decouple from one another (1) the user-visible element parameters and quantities, (2) the network connections, states and flows, and (3) the mathematical problem being solved. The three layers are referred to, resepectively, as the **data**, **network**, and **mathematical** (or **math**) modeling layers as shown in Figure 3.1.

data model layer	defines user-visible element parameters and quantities
network model layer	defines network connections and states (e.g. voltages, injections) and relationship between states and flows
mathematical model layer	defines mathematical model to solve (e.g. variables, costs, constraints)

Figure 3.1: MATPOWER Model Layers

The data model layer is further decoupled from any particular data format, such as the legacy MATPOWER case struct (`mpc`) and case file formats, by introducing a data conversion service (**data model converter**) to convert data between the data model and specific external data formats.

[1] Hence the name *MP-Element* used early on in the development cycle.

Each modeling layer, plus the data conversion service, is organized around a collection of **element** objects, one for each **element type**, enclosed in a **container** object. An element type corresponds to a particular type of device (e.g. bus, generator, transmission line) or some other attribute or service (e.g. transmission interface, reserve requirement) in the system. This structure provides extraordinary flexibility by allowing the user to customize the environment by adding new, or modifying existing, element types independently from the rest.

3.1 MATPOWER Object Instances

In any given MATPOWER run, a set of object instances are created and used to solve the problem. The structure of these object instances in the *object-oriented MATPOWER core architecture* (MP-Core) is show in Figure 3.2. The classes for the various objects may be specific to (1) the type of problem being solved, (2) the problem formulation, (3) the data source, and for individual elements, (4) the type of element. The labels in the white circles in the figure are used by convention throughout the codebase in variable and class names for the corresponding type of object.

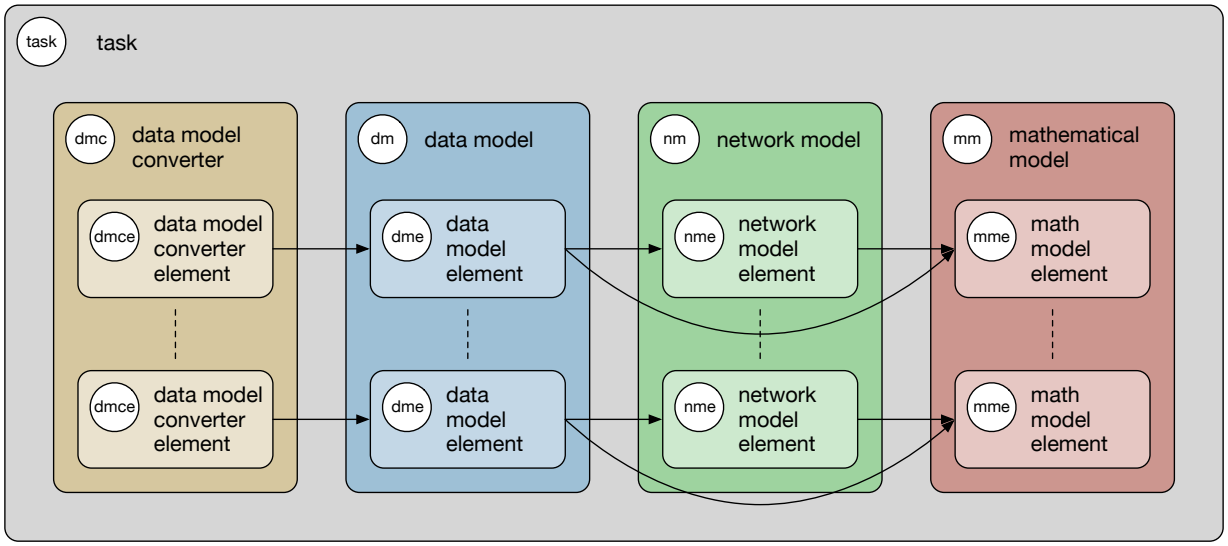

Figure 3.2: MATPOWER Object Instances

A single **task** object is created to manage the overall process. The task is specific to the type of problem being solved, e.g. power flow (PF), continuation power flow (CPF), or optimal power flow (OPF), and it has a `run()` method that sets up and solves the correspnding problem. For example, the following runs an OPF for the 9-bus case.

```
mpopt = mpoption('verbose', 2);     % set MATPOWER options
task = mp.task_opf();               % create task object for OPF
task.run('case9', mpopt);           % create and run task for 'case9'
```

The steps shown in Listing 3.1 are roughly equivalent to those performed when the task is run. It defines the classes used to construct each of the model objects, as well as the data model converter. In this example, the classes are defined explicitly, but in the actual code they are returned by calls to corresponding methods, allowing them to be overridden by subclasses.

The task then creates the data model converter object that corresponds to the data source provided, followed by the three main model objects. The data model is created from the specified data source with the help of the data model converter, and is then used to create the network model. The math model is then created using both the data and network models. After solving itself, the math model is also used to update the states of the other two model objects.

Listing 3.1: Basic steps performed by the task's run() method[Page 7, 2]

```
% define classes used to construct model objects and data model converter
dmc_class = @mp.dm_converter_mpc2;  % data model convert class, MATPOWER case format v2
dm_class = @mp.data_model_opf;      % data model class for OPF
nm_class = @mp.net_model_acp;       % network model class for AC polar
mm_class = @mp.math_model_opf_acps; % math model class for AC polar power OPF

% create objects
dmc = dmc_class().build();              % create data model converter
dm = dm_class().build('case9', dmc);    % create data model for 'case9'
nm = nm_class().build(dm);              % create network model
mm = mm_class().build(nm, dm, mpopt);   % create math model

% find solution
opt = mm.solve_opts(nm, dm, mpopt);     % get solver options
mm.solve(opt);                          % solve math model
nm = mm.network_model_x_soln(nm);       % update network model state with soln
nm.port_inj_soln();                     % use network model to compute flows
dm = mm.data_model_update(nm, dm, mpopt);   % update data model with soln
```

Each of the four main objects created by the task consists of a container object holding a set of corresponding element objects. That is, the data model contains a set of data model elements, the network model, a set of network model elements, etc., one for each element type. Each element type is associated with a **name**, that is a valid struct field name used to identify the corresponding element in each container object. The list of element classes for a given container is defined by the container class, but can be modified after the container's construction and before calling its `build()` method.

The build process of a given container object simply loops through its set of elements, building each one, possibly with access to the respective element of the other model layers. For example, when building the network model (`nm`), a network model element (`nme`) is constructed for each type of element, pulling its data from the corresponding data model element (`dme`). For example, the network model element for generators pulls its data from the data model element for generators.

This process is described in more detail in Chapters 5–8.

3.2 MATPOWER Class Hierarchies

A summary of the class inheritance structure in MP-Core is represented in Figure 3.3, showing class name conventions, with abstract classes displayed with a single border and concrete classes with a double border. A significant portion of MP-Core functionality is implemented in abstract base classes, greatly reducing the effort involved in customization.

Subclasses in these hierarchies are distinguished from one another by various attributes. For example, task classes are distinguished by the type of *task* or problem being solved (e.g. PF, CPF, OPF), data model converters by the *data format* (e.g. MATPOWER case v2, PSS/E RAW), data models by the *task*, network models by the *formulation* (e.g. DC, AC polar, AC cartesian), mathematical models by the *task* and *formulation*. That goes for both the container classes and their respective element classes, which are also distinguished by the corresponding *element type* (e.g. bus, generator, transmission line).

The `mp.element_container` is a mixin class providing shared functionality for the four container types mentioned above, implementing a set of elements, which can be addressed by both index and name and supplying the properties `elements` and `element_classes`.

[2] This code should execute successfully from the command line without modification.

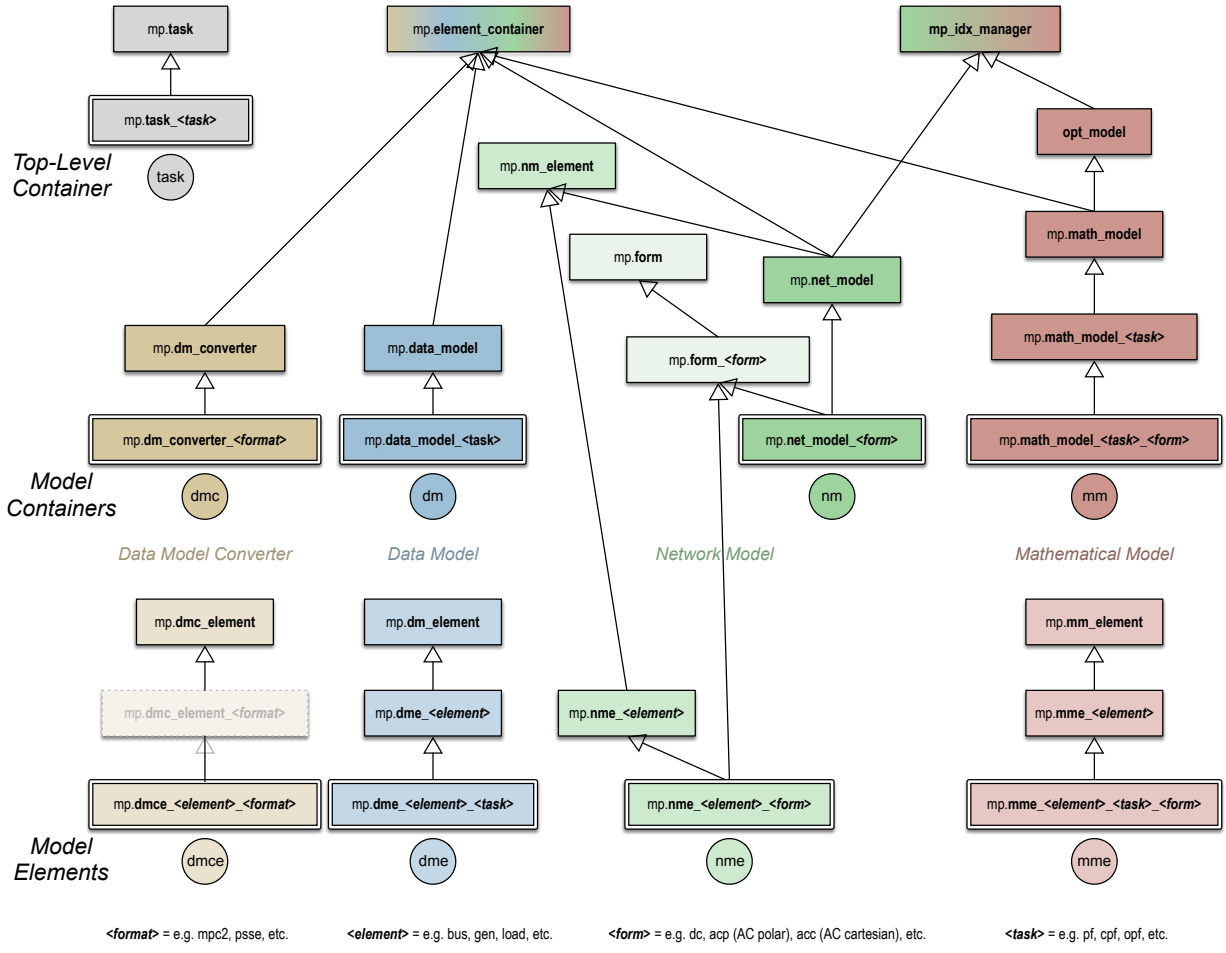

Figure 3.3: MATPOWER Class Hierarchies

Other mixin classes are also sometimes used when certain functionality and implementation is shared across classes in ways that do not match the primary inheritance paths.

3.3 Two MATPOWER Frameworks

MATPOWER currently provides two approaches to utilizing the object-oriented MATPOWER core architecture.

The first, which we call the **legacy MATPOWER framework**, wraps MP-Core objects inside the legacy user interface, with its inherent limitations, in order to provide backward compatibility for legacy user customization mechanisms. This allows MP-Core to be used internally to implement all of the legacy PF, CPF and OPF functionality and, even more importantly, to be validated by MATPOWER's extensive legacy test suite.

The second approach, which we call the **flexible MATPOWER framework**, involves an object-oriented design with a new customization architecture, able to make the full scope of flexibility of MP-Core accessible to the end user. For example, this framework is required to take advantage of new modeling capabilities to add multiphase unbalanced and hybrid models. It provides its own version of the top-level user functions, namely `run_pf()`, `run_cpf()`, and `run_opf()` *(note the underscores in the names)*.

One of the primary differences between the two frameworks is that the legacy framework converts the MATPOWER case data to internal format, removing offline equipment and renumbering buses consecutively using the legacy `ext2int()` function, *before* creating the task object and running it. After solving, it converts the case back to the external format using `int2ext()` before returning the result. This conversion is required for the legacy user callback mechanisms, but is not necessary for MP-Core itself, so it is not included in the flexible framework.

3.4 MATPOWER Customization

The primary motivation behind the design of MP-Core was to facilitate customization, both for the end user and for the developer who wants to add new capabilities to MATPOWER itself. Given the object-oriented architecture, this is possible by simply subclassing existing classes to modify or override their behavior or adding completely new classes, which can often inherit significant functionality from existing abstract base classes.

The flexible MATPOWER framework includes a mechanism for defining and using **MATPOWER extensions** (see Chapter 9.3). A MATPOWER extension is essentially a collection of modifications and additions to be made to the set default classes used to construct the task, model and model element objects.

4 Task Object

The task object is the one that builds and manages the model objects in order to solve the problem of interest. The `mp.task` base class implements much of the functionality, with PF, CPF and OPF subclasses, namely `mp.task_pf`, `mp.task_cpf`, and `mp.task_opf`, respectively, specifying the model classes to use and implementing other problem-specific functionality. The typical usage pattern is simply to construct the task object for the problem of interest, then call its `run()` method, passing in a struct of input data, a MATPOWER options struct, and an optional cell array of MATPOWER extensions.

4.1 Running a Task

Most of the action related to the task object occurs in the `run()` method. In a typical case, as illustrated in Listing 3.1, it simply builds the objects for the three model layers sequentially, solves the math model, and uses the results to update the network and data models. However, the actual `run()` method also allows for each model layer to iterate with a modified instance of the model as shown in the flowchart in Figure 4.1. This can be used, for example, to iteratively update and re-solve a power flow in order to automatically satisfy the generator reactive power limits.

4.2 Building Model and Converter Objects

Each of the build steps, marked with the stars in Figure 4.1, consists of the following sub-steps:

1. Determine the class for the corresponding container object. There is a default, defined by a task method, but it can be overridden by a task subclass, or modified by user options or extensions.
2. Construct the container object.
3. Determine the set of classes for the individual element objects. The container class defines the defaults, but they can also be modified by user options or extensions.
4. Call the container object's `build()` method to construct the element objects and complete the build process.

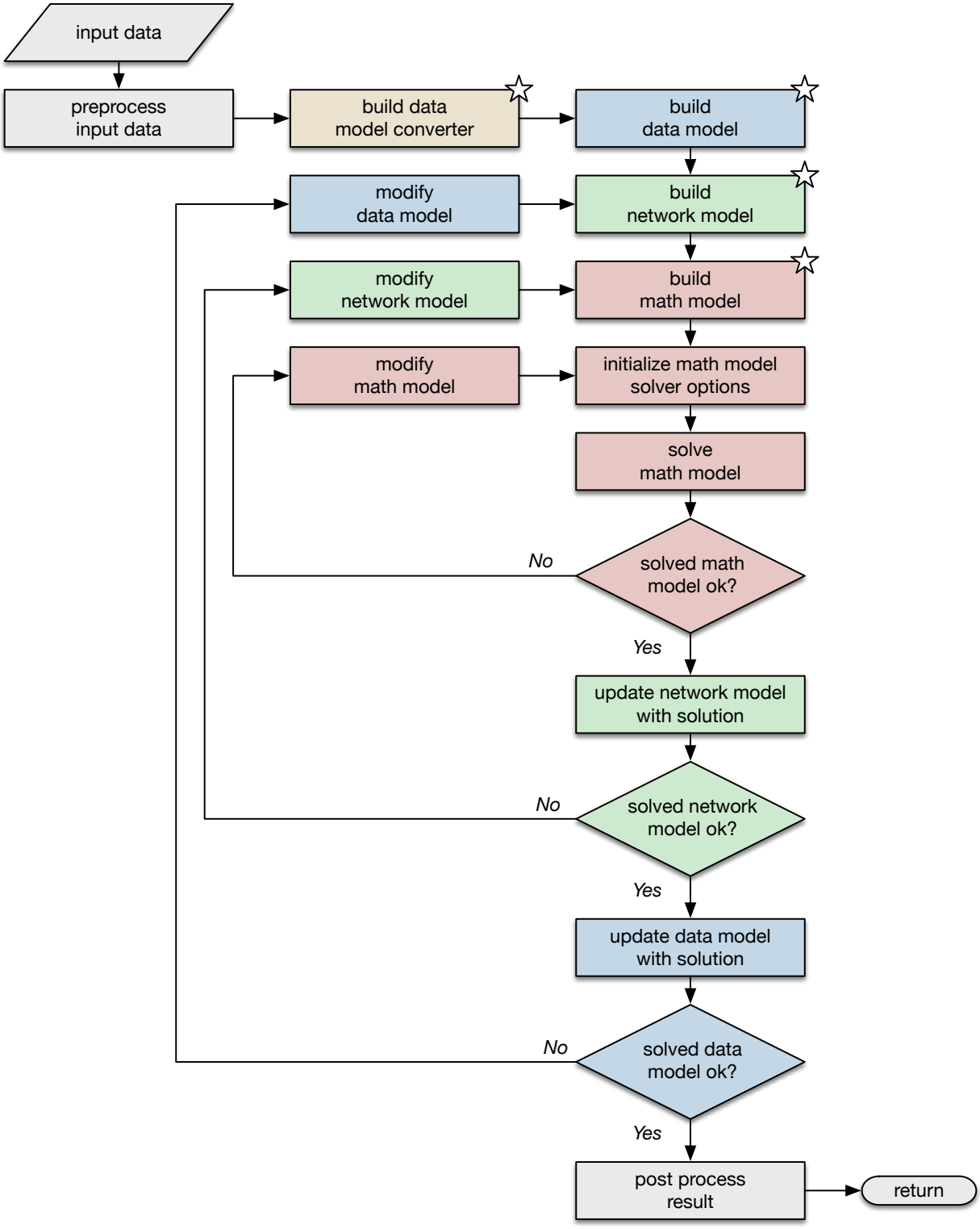

Figure 4.1: Flowchart of task `run()` method

4.3 Iterative Execution

As mentioned above, the `run()` method allows for an iterative solution at any of the three modeling layers. This is accomplished by overridding the `next_dm()`, `next_nm()`, or `next_mm()` methods, respectively, for the data, network or math models. By default, these methods return an empty matrix, indicating that iteration should terminate. On the other hand, if a modified model object is returned, it triggers a new iteration with the modified model.

This feature is used by both PF and CPF to implement enforcement of certain constraints, such as generator reactive power limits.

4.4 Other Methods

A task also has a `print_soln()` method for pretty printing the solution to the console and a `save_soln()` method for saving the saved case to a file.

5 Data Model Object

The data model is essentially the internal representation of the input data provided by the user for the given simulation or optimization run and the output 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.

5.1 Data Models

A data model object is primarily a container for data model element objects. All data model classes inherit from `mp.data_model` and therefore also from `mp.element_container`, and may be task-specific, as shown in Figure 5.1. For a simple power flow problem, `mp.data_model` is used directly as a concrete class. For CPF and OPF problems, subclasses are used. In the case of CPF, `mp.data_model_cpf` encapsulates both the base and the target cases. In the case of the OPF, `mp.data_model_opf` includes additional input data, such as generator costs, and output data, such as nodal prices and shadow prices on line flow contraints.

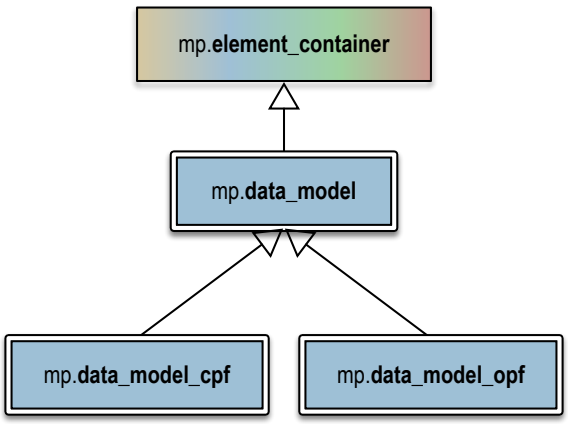

Figure 5.1: Data Model Classes

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

5.1.1 Building a Data Model

There are two steps to building a data model. The first is to call the constructor of the desired data model class, without arguments. This initializes the `element_classes` property with a list of data model element classes. This list can be modified before the second step, which is to call the `build()` method, passing in the data and a corresponding data model converter object.

```
dmc = mp.dm_converter_mpc2().build();
dm = mp.data_model();
dm.build('case9', dmc);
```

The `build()` method proceeds through the following stages sequentially, looping through each element at every stage.

1. **Create** – Instantiate each element object and add it to the `elements` property of the `dm`.
2. **Import** – Use the corresponding data model converter element to read the data into each element's table(s).
3. **Count** – Determine the number of instances of each element present in the data, store it in the element's `nr` property, and remove the element type from `elements` if the count is 0.
4. **Initialize** – Initialize the (online/offline) status of each element and create a mapping of ID to row index in the `ID2i` element property.
5. **Update status** – 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`.
6. **Build parameters** – 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.

5.1.2 System Level Parameters

There are a few system level parameters such as the system per-unit power base that are stored in data model properties. Balanced single-phase model elements, typical in transmission systems, use an MVA base found in `base_mva`. Unbalanced three-phase model elements, typical in distribution systems, use a kVA base found in `base_kva`. Models with both types of elements, therefore, use both properties.

5.1.3 Printing a Data Model

The `mp.data_model` provides a `pretty_print()` method for displaying 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** - count, 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.

5.2 Data Model Elements

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.[1] So, for example, the data model element for generators contains a table with the generator data for all generators in the system, where each table row corresponds to an individual generator.

By convention, data model element variables are named `dme` and data model element class names begin with `mp.dme`. Figure 5.2 shows the inheritance relationships between a few example data model element classes. Here the `mp.dme_bus`, `mp.dme_gen` and `mp.dme_load` classes are used for PF and CPF runs, while the OPF requires task-specific subclasses of each.

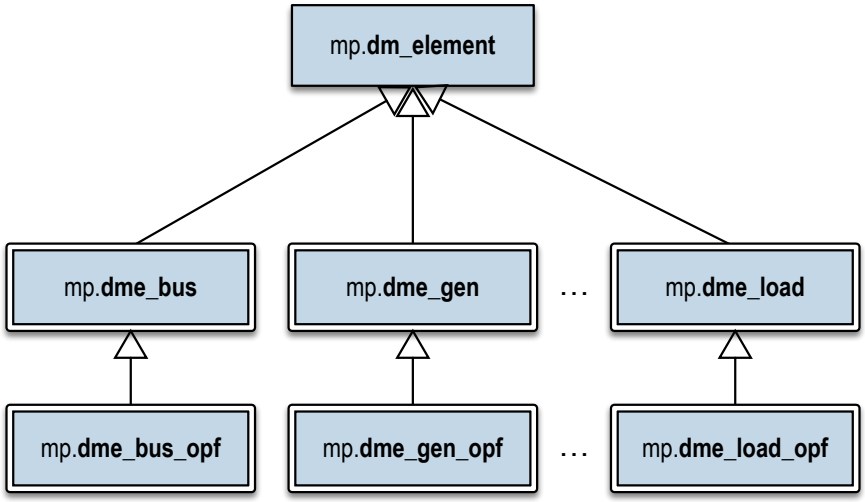

Figure 5.2: Data Model Element Classes

5.2.1 Data Tables

Typically, a data model element has at least one main table, stored in the `tab` property. Each row in the table corresponds to an individual element and the columns are the parameters. In general, MATPOWER attempts to follow the parameter naming conventions outlined in *The Common Electric Power Transmission System Model* (CTM) [CTM]. The following parameters (table columns) are shared by all data model elements.

- **uid** – positive integer serving as a unique identifier for the element
- **name** – optional string identifier for the element
- **status** – 0 or 1, on/off-line status of the element
- **source_uid** – implementation specific *(e.g. sometimes used to map back to a specific record in the source data)*

[1] Implemented using the built-in `table` and included `mp_table` classes, respectively, under MATLAB and GNU Octave. See also `mp_table_class()`.

5.2.2 Properties

The table below includes additional properties, besides the main table `tab`, found in all data model elements.

Table 5.1: Data Model Element Properties

Property	Description
`nr`	number of rows in the table, i.e. the *total* number of elements of the type
`n`	number of *online* elements of the type
`on`	vector of row indices of online elements
`off`	vector of row indices of offline elements
`ID2i`	$M \times 1$ vector mapping IDs to row indices, where M is the largest ID value
`i2on`	$n_r \times 1$ vector mapping row indices to the corresponding index into the `on` vector *(for online elements)* or `off` vector *(for offline elements)*
`tab`	main data table

5.2.3 Methods

A data model element also has a `name()` method that returns the name of the element type under which it is entered in the data model (container) object. For example, the name returned for the `mp.dme_gen` class is `'gen'`, which means the corresponding data model element object is found in `dm.elements.gen`.

There are also methods named `label()` and `labels()` which return user visible names for singular and plural instances of the element used when pretty-printing. For `mp.dme_gen`, for example, these return `'Generator'` and `'Generators'`, respectively.

Note: Given that these name and label methods simply return a character array, it might seem logical to implement them as Constant properties instead of methods, but this would prevent the value from being overridden by a subclass, in effect precluding the option to create a new element type that inherits from an existing one.

The `main_table_var_names()` method returns a cell array of variable names defining the columns of the main data table. These are used by the corresponding data model converter element to import the data.

There are also methods that correspond to the build steps for the data model container object, `count()`, `initialize()`, `init_status()`, `update_status()`, and `build_params()`, as well as those for pretty printing output, `pretty_print()`, etc.

5.2.4 Connections

As described in the *Network Model Object* (page 22) section, the network model consists of elements with **nodes**, and elements with **ports** that are connected to those nodes. The corresponding data model elements, on the other hand, contain the information defining how these port-node connections are made in the network model, for example, to link generators and loads to single buses, and branches to pairs of buses.

A **connection** in this context refers to a mapping of a set of ports of an element of type *A* (e.g. *from bus* and *to bus* ports of a *branch*) to a set of nodes created by elements of type *B* (e.g. *bus*). We call the node-creating elements **junction** elements. A single connection links all type *A* elements to corresponding type *B* junction elements. For example, a three-phase branch could define two connections, a *from bus* connection and a *to bus* connection, where each connection defines a mapping of 3 ports per branch to the 3 nodes of each corresponding bus.

A data model element class defines its connections by implementing a couple of methods. The `cxn_type()` method returns the name of the junction element(s) for the connection(s). The `cxn_idx_prop()` method returns the name(s)

of the property(ies) containing the indices of the corresponding junction elements. For example, if `cxn_type()` for a branch element class returns `'bus'` and `cxn_idx_prop()` returns `{'fbus', 'tbus'}`, that means it is defining two connections, both to `'bus'` elements. The `fbus` and `tbus` properties of the branch object are each vectors of indices into the set of `'bus'` objects, and will be used automatically to generate the connectivity for the network model.

It is also possible to define a connection where the junction element type is instance-specific. For example, if you had two types of buses, and a load element that could connect to either type, then each load would have to indicate both which type of bus and which bus of that type it is connected to. This is done by having `cxn_type()` return a cell array of the valid junction element types and `cxn_type_prop()` return the name(s) of the property(ies) containing vector(s) of indices into the junction element type cell array.

6 Data Model Converter Object

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.

6.1 Data Model Converters

A data model converter object is primarily a container for data model converter element objects. All data model converter classes inherit from `mp.dm_converter` and therefore also from `mp.element_container` and they are specific to the type or format of the data source, as shown in Figure 6.1. In this example, the PSS/E RAW format converter has not yet been implemented, but is shown here for illustration.

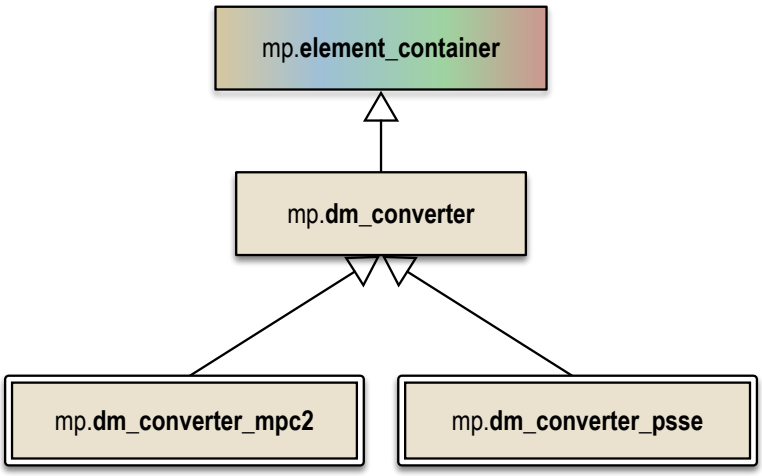

Figure 6.1: Data Model Converter Classes

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

6.1.1 Building a Data Model Converter

A data model converter object is created in two steps. The first is to call the constructor of the desired data model converter class, without arguments. This initializes the `element_classes` property with a list of data model converter element classes. This list can be modified before the second step, which is to call the `build()` method, also without parameters, which simply instantiates and adds the set of element objects indicated in `element_classes`. Once it has been created, it is ready to be used for its two primary functions, namely *import* and *export*.

```
dmc = mp.dm_converter_mpc2();
dmc.build();
```

6.1.2 Importing Data

The `import()` method is called automatically by the `build()` method of the data model object. It takes a data model object and a data source and updates the data model by looping through its element objects and calling each element's own `import()` method to import the element's data from the data source into the corresponding data model element. For a MATPOWER case struct it would like like this.

```
mpc = loadcase('case9');
dm = dmc.import(dm, mpc);
```

6.1.3 Exporting Data

Conversely, the `export()` method takes the same inputs but returns an updated data source, once again looping through its element objects and calling each element's own `export()` method to export data from the corresponding data model element to the respective portion of the data source.

```
mpc = dmc.export(dm, mpc);
```

Calling `export()` without passing in a data source will initialize one from scratch.

```
mpc = dmc.export(dm);
```

6.2 Data Model Converter Elements

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`. Figure 6.1 shows the inheritance relationships between a few example data model converter element classes. Here the PSS/E classes have not yet been implemented, but are shown here for illustration.

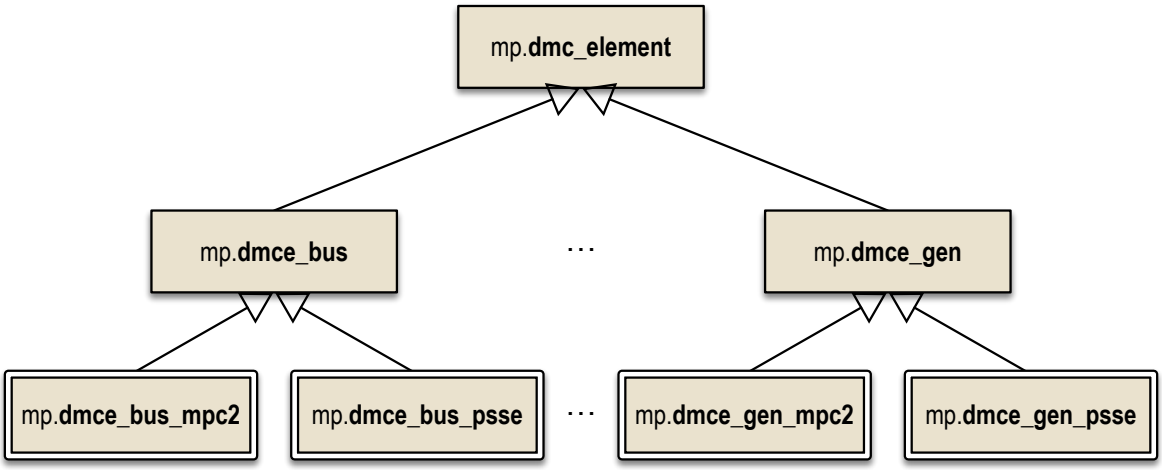

Figure 6.2: Data Model Converter Element Classes

6.2.1 Data Import Specifications

The default `import()` method for a data model converter element first calls the `get_import_spec()` method to get a struct containing the specifications that define the details of the import process. This specification is then passed to `import_table_values()` to import the data.

The import specifications include things like where to find the data in the data source, the number of rows, number of columns, and possibly a row index vector for rows of interest,[1] and a map defining how to import each column of the main data table.

This map `vmap` is a struct returned by the `table_var_map()` method with fields matching the table column names for the corresponding data model element `dme`. For example, if `vn` contains a variable, that is column, name, then `vmap.(vn) = <value>` defines how that data table column will be imported or initialized, as summarized in Table 6.1 for different types of values.

Table 6.1: Variable Map Values

<value>	Description
`{'IDs'}`	Assign consecutive IDs starting at 1.
`{'col', c}` *or* `{'col', c, sf}` *or* `{'col', c, sf_fcn}`	Copy the data directly from column *c* of data source, optionally scaling it by a numerical scale factor *sf*, or by the value returned by the function handle *sf_fcn*, called as *sf_fcn(dmce, vn)*.
`{'cell', val}`	Create a cell array with each element initialized with *val*.
`{'num', n}`	Create a numeric vector with each element initialized with numeric scalar *n*.
`{'fcn', ifn}` *or* `{'fcn', ifn, efn}`	Assign the values returned by the import function handle in *ifn*, where the optional *efn* is the corresponding export function. The import and export functions are called as *ifn(dmce, d, spec, vn)* and *efn(dmce, dme, d, spec, vn, ridx)*, respectively, where *d* is the data source, *spec* is the import/export specification, and *ridx* is an optional vector of row indices.

The `table_var_map()` in `mp.dmc_element` 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.

[1] For example, when extracting loads from a bus matrix, where only certain buses have corresponding loads.

6.2.2 Data Export Specifications

The default `export()` method first calls the `get_export_spec()` method to get a struct containing the specifications that define the details of the export process. This specification is then passed to `export_table_values()` to export the data.

The export of data from a data model element back to the original data format is handled by the same variable map as the input, by default.

The `init_export_data()` method is used to initialize the relevant output data structure before exporting to it, if the `data_exists()` method returns false.

7 Network Model Object

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. This network model with a unified structure is *the key* to a flexible modular design where model elements can simply define a few parameters and all of the mathematics involved in computing injections and their derivatives for a given state is handled automatically.

One of the unique features of the network model is that the network model object, which contains network model elements, **is a** network model element itself. Each network model element can optionally define the following:

- **nodes** to serve as network connection points
- **ports** for connecting to network nodes
- **states** which fully capture the element's operating condition

There are two types of states that make up the element's full state variable `x`, voltage states `v` associated with each port, and optional non-voltage states `z`. The network model object inherits from `mp_idx_manager` from MP-Opt-Model to track and index the nodes, ports, and states added by its elements, and the corresponding voltage and non-voltage state variables.

A given network model implements a specific network model **formulation**. Figure 7.1 shows the structure of an AC network model for an element with n_p connection ports and n_z non-voltage states.

The MP-Element technical note, *MATPOWER Technical Note 5* [TN5], includes a lot more detail on the network model and especially on the mathematics involved in the model formulations.

7.1 Network Model Formulations

Each concrete network model element class, including the container class, inherits from a specific subclass of `mp.form`. That is, it implements a specific network model formulation. For example, Figure 7.2 shows the corresponding classes for the three network model formulations currently implemented, (1) DC, (2) AC with cartesian voltage representation, and (3) AC with polar voltage representation.

By convention, network model formulation class names begin with `mp.form`. It is the formulation class that defines the network model's parameters and methods for accessing them. It also defines the form of the state variables, real or complex, and methods for computing injections as a function of the state, and in the case of nonlinear formulations, corresponding derivatives as well.

All formulations share a common structure, illustrated in Figure 7.1, with ports, corresponding voltage states, non-voltage states, and functions of predefined form for computing port injections from the state.

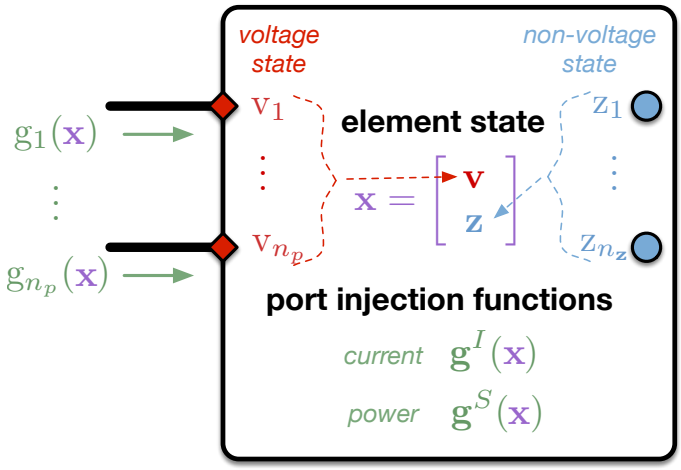

Figure 7.1: AC Model for Element with n_p Ports

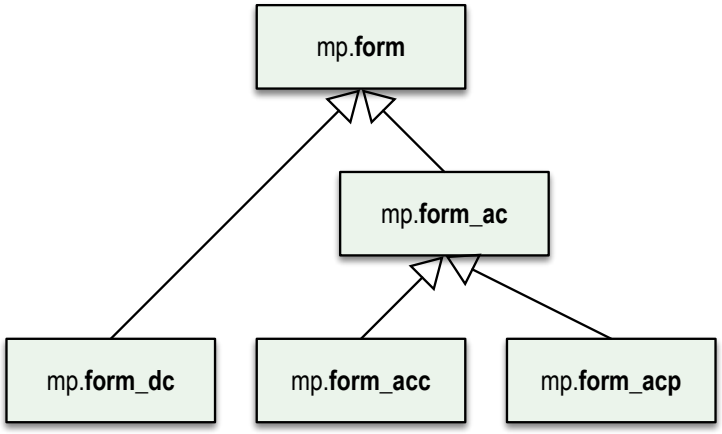

Figure 7.2: Network Model Formulation Classes

7.1.1 DC Formulation

For the DC formulation, the state vector $\boldsymbol{x}$ is real valued and the port injection function is defined in terms of active power injections. The state begins with the $n_p \times 1$ vector $\boldsymbol{\theta}$ of voltage angles at the n_p ports, and may include an $n_{\boldsymbol{z}} \times 1$ real vector of additional state variables $\boldsymbol{z}$, for a total of $n_{\boldsymbol{x}}$ state variables.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{\theta} \\ \boldsymbol{z} \end{bmatrix} \tag{7.1}$$

The port injection function in this case defines the active power port injections as a linear function of a set of parameters $\underline{\boldsymbol{B}}$, $\underline{\boldsymbol{K}}$ and $\underline{\boldsymbol{p}}$, where $\underline{\boldsymbol{B}}$ is an $n_p \times n_p$ susceptance matrix, $\underline{\boldsymbol{K}}$ is an $n_p \times n_{\boldsymbol{z}}$ matrix coefficient for a linear power injection function, and $\underline{\boldsymbol{p}}$ is an $n_p \times 1$ constant power injection.

$$\begin{aligned} \boldsymbol{g}^P(\boldsymbol{x}) &= \begin{bmatrix} \underline{\boldsymbol{B}} & \underline{\boldsymbol{K}} \end{bmatrix} \boldsymbol{x} + \underline{\boldsymbol{p}} \\ &= \underline{\boldsymbol{B}}\boldsymbol{\theta} + \underline{\boldsymbol{K}}\boldsymbol{z} + \underline{\boldsymbol{p}} \end{aligned} \tag{7.2}$$

7.1.2 AC Formulations

For the AC formulations, the state vector $\mathbf{x}$ is complex valued and there are two port injection functions, one for complex power injections and one for current injections, as shown in Figure 7.1. The state begins with the $n_p \times 1$ vector $\mathbf{v}$ of complex voltages at the n_p ports, and may include an $n_{\mathbf{z}} \times 1$ real vector of additional state variables $\mathbf{z}$, for a total of $n_{\mathbf{x}}$ state variables.

$$\mathbf{x} = \begin{bmatrix} \mathbf{v} \\ \mathbf{z} \end{bmatrix} \tag{7.3}$$

The port injection functions for the model, both complex power injection $\mathbf{g}^S(\mathbf{x})$ and complex current injection $\mathbf{g}^I(\mathbf{x})$, are defined by three terms, a linear current injection component $\mathbf{i}^{lin}(\mathbf{x})$, a linear power injection component $\mathbf{s}^{lin}(\mathbf{x})$, and an arbitrary nonlinear component, $\mathbf{s}^{nln}(\mathbf{x})$ or $\mathbf{i}^{nln}(\mathbf{x})$, respectively.

The linear current and power injection components are expressed in terms of the six parameters, $\underline{\mathbf{Y}}$, $\underline{\mathbf{L}}$, $\underline{\mathbf{M}}$, $\underline{\mathbf{N}}$, $\underline{\mathbf{i}}$, and $\underline{\mathbf{s}}$. The admittance matrix $\underline{\mathbf{Y}}$ and linear power coefficient matrix $\underline{\mathbf{M}}$ are $n_p \times n_p$, linear coefficient matrices $\underline{\mathbf{L}}$ and $\underline{\mathbf{N}}$ are $n_p \times n_{\mathbf{z}}$, and $\underline{\mathbf{i}}$ and $\underline{\mathbf{s}}$ are $n_p \times 1$ vectors of constant current and power injections, respectively.

$$\begin{aligned} \mathbf{i}^{lin}(\mathbf{x}) &= \begin{bmatrix} \underline{\mathbf{Y}} & \underline{\mathbf{L}} \end{bmatrix} \mathbf{x} + \underline{\mathbf{i}} \\ &= \underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}} \end{aligned} \tag{7.4}$$

$$\begin{aligned} \mathbf{s}^{lin}(\mathbf{x}) &= \begin{bmatrix} \underline{\mathbf{M}} & \underline{\mathbf{N}} \end{bmatrix} \mathbf{x} + \underline{\mathbf{s}} \\ &= \underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}} \end{aligned} \tag{7.5}$$

Note that the arbitrary *nonlinear* injection component, represented by either $\mathbf{s}^{nln}(\mathbf{x})$ or $\mathbf{i}^{nln}(\mathbf{x})$, corresponds to a single set of injections represented either as a complex power injection or as a complex current injection, but not both. Since the functions represent the same set of injections, they are not additive components, but rather must be related to one another by the following relationship.

$$\mathbf{s}^{nln}(\mathbf{x}) = \begin{bmatrix} \diagdown \\ \mathbf{v} \\ \diagdown \end{bmatrix} \left(\mathbf{i}^{nln}(\mathbf{x})\right)^* \tag{7.6}$$

Complex Power Injections

Then the port injection function for complex power can be written as follows.

$$\begin{aligned}\mathbf{g}^S(\mathbf{x}) &= \left[\diagup \mathbf{v} \diagdown\right] \left(\mathbf{i}^{lin}(\mathbf{x})\right)^* + \mathbf{s}^{lin}(\mathbf{x}) + \mathbf{s}^{nln}(\mathbf{x}) \\ &= \left[\diagup \mathbf{v} \diagdown\right] \left(\underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}}\right)^* + \underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}} + \mathbf{s}^{nln}(\mathbf{x})\end{aligned} \tag{7.7}$$

Complex Current Injections

Similarly, the port injection function for complex current can be written as follows.

$$\begin{aligned}\mathbf{g}^I(\mathbf{x}) &= \mathbf{i}^{lin}(\mathbf{x}) + \left[\diagup \mathbf{s}^{lin}(\mathbf{x}) \diagdown\right]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x}) \\ &= \underline{\mathbf{Y}}\mathbf{v} + \underline{\mathbf{L}}\mathbf{z} + \underline{\mathbf{i}} + \left[\diagup \underline{\mathbf{M}}\mathbf{v} + \underline{\mathbf{N}}\mathbf{z} + \underline{\mathbf{s}} \diagdown\right]^* \mathbf{\Lambda}^* + \mathbf{i}^{nln}(\mathbf{x})\end{aligned} \tag{7.8}$$

The derivatives of $\mathbf{s}^{nln}$ and $\mathbf{i}^{nln}$ are assumed to be provided explicitly, and the derivatives of the other terms of (7.7) and (7.8) are derived in [TN5].

7.2 Network Models

A network model object is primarily a container for network model 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` as shown in Figure 7.3.

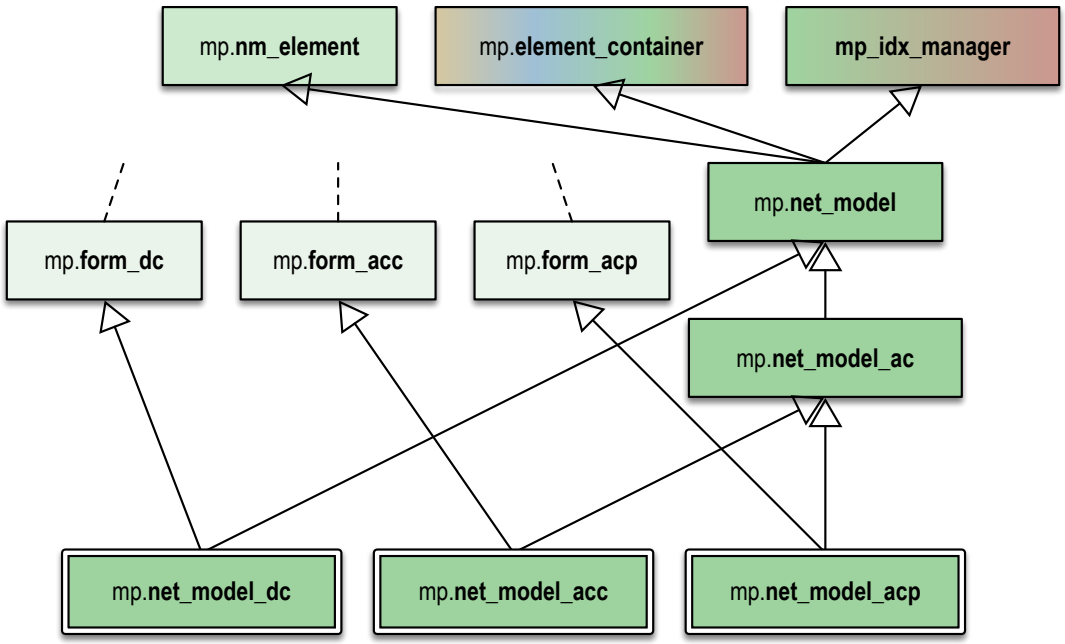

Figure 7.3: Network Model Classes

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

7.2.1 Building a Network Model

A network model object is created in two steps. The first is to call the constructor of the desired network model class, without arguments. This initializes the `element_classes` property with a list of network model element classes. This list can be modified before the second step, which is to call the `build()` method, passing in the data model object.

```
nm = mp.net_model_acp();
nm.build(dm);
```

The `build()` method proceeds through the following stages sequentially, looping through each element at each stage.

1. **Create** – Instantiate each element object.
2. **Count and add** - For each element object, determine the number of online elements from the corresponding data model element and, if nonzero, store it in the object and add the object to the `elements` property of the `nm`.
3. **Add nodes** – Allow each element to add network nodes, then add voltage variables for each node.
4. **Add states** – Allow each element to add non-voltage states, then add non-voltage variables for each state.
5. **Build parameters** – Construct the formulation-specific model parameters for each element, including mappings of element port to network node and element non-voltage state to system non-voltage variable. Add ports to the container object for each element to track per-element port indexing.

7.2.2 Node Types

Most problems require that certain nodes be given special treatment depending on their *type*. For example, in the power flow problem, there is typically a single **reference** node, some **PV** nodes, with the rest being **PQ** nodes.

In the current design, each node-creating network model element class implements a `node_types()` method that returns information about the types of the nodes it creates. The container object `node_types()` method assembles that information for the full set of network nodes. It can also optionally, assign a new reference node if one does not exist. There are also methods, namely `set_node_type_ref()`, `set_node_type_pv()`, `set_node_type_pq()`, for setting the type of a network node and having the relevant elements update their corresponding data model elements.

7.3 Network Model Elements

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, e.g. $\underline{\boldsymbol{B}}$, $\underline{\boldsymbol{K}}$ and $\underline{\boldsymbol{p}}$ from (7.2) for DC and $\underline{\mathbf{Y}}$, $\underline{\mathbf{L}}$, $\underline{\mathbf{M}}$, $\underline{\mathbf{N}}$, $\underline{\mathbf{i}}$, and $\underline{\mathbf{s}}$ from (7.4) and (7.5) for AC.

So, for example, in a system with 1000 in-service transmission lines, the $\underline{\mathbf{Y}}$ parameter in the corresponding AC network model element object would be a 2000×2000 matrix for an aggregate 2000-port element, representing the 1000 two-port transmission lines.

By convention, network model element variables are named `nme` and network model element class names begin with `mp.nme`. Figure 7.4 shows the inheritance relationships between a few example network model element classes. Here the `mp.nme_bus_acp` and `mp.nme_gen_acp` classes are used for all problems with an AC polar formulation, while the AC cartesian and DC formulations use their own respective subclasses.

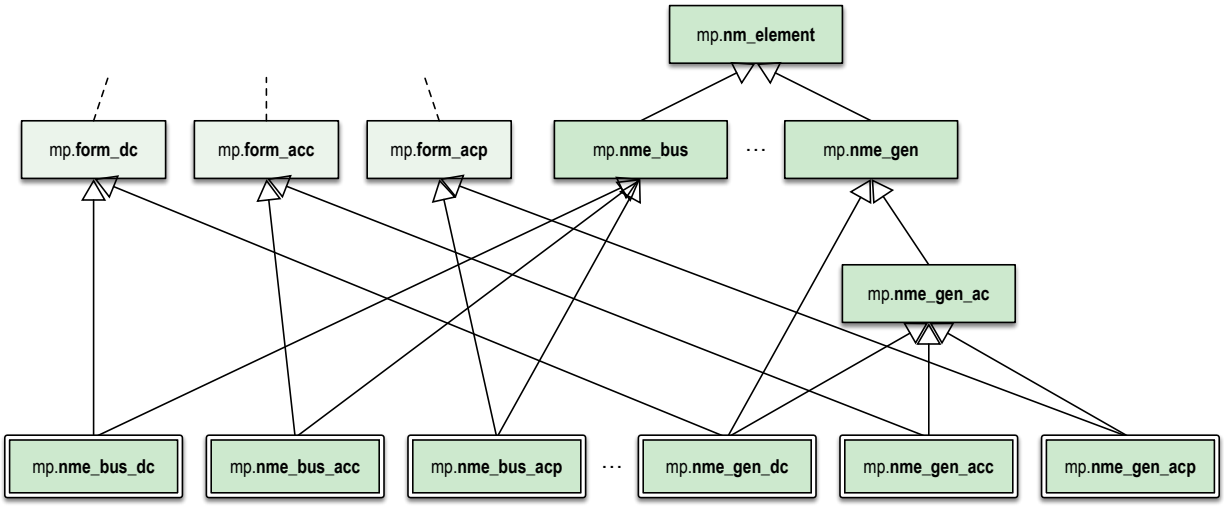

Figure 7.4: Network Model Element Classes

7.3.1 Example Elements

Here are brief descriptions of the network models for a few simple element types. There are other elements, and the point is that new elements are relatively simple to implement, simply by specifying the nodes, ports and states they add, and the parameters that define the relationships between the states and the port injections.

Bus

A **bus** element inherits from `mp.nme_bus` and defines a single node per in-service bus, with no ports or non-voltage states. So it has no model parameters.

Generator

A **gen** element is a 1-port element that inherits from `mp.nme_gen` and defines a single non-voltage state per in-service generator to represent the power injection. It connects to the node corresponding to a particular bus. The only non-zero parameters are $\underline{\boldsymbol{K}}$ (DC) or $\underline{\mathbf{N}}$ (AC), which are negative identity matrices, since the power injections (into the element) are the negative of the generated power.

Branch

A **branch** element is a 2-port element that inherits from `mp.nme_branch` with no nodes or non-voltage states. It connects to nodes corresponding to two particular buses. The only non-zero parameters are $\underline{\boldsymbol{B}}$ and $\underline{\boldsymbol{p}}$ (DC), or $\underline{\mathbf{Y}}$ (AC).

Load

A **load** element is a 1-port element that inherits from `mp.nme_load` with no ports or states. It connects to the node corresponding to a particular bus. For a simple constant power load, the only non-zero parameters are $\underline{\boldsymbol{p}}$ (DC) or $\underline{\mathbf{s}}$ (AC), equal to the power consumed by the load.

7.3.2 Building Element Parameters

Typically, a network model element builds parameters only for its in-service elements, stacking the corresponding parameters into vectors and matrices, with one row per element of that type. For the DC formulation, these are the three parameters $\underline{\boldsymbol{B}}$, $\underline{\boldsymbol{K}}$ and $\underline{\boldsymbol{p}}$ from Section 7.1.1. For the AC formulations they are the six parameters, $\underline{\mathbf{Y}}$, $\underline{\mathbf{L}}$, $\underline{\mathbf{M}}$, $\underline{\mathbf{N}}$, $\underline{\mathbf{i}}$, and $\underline{\mathbf{s}}$ from Section 7.1.2.

Take, for example, an AC model with two-port transmission lines modeled by a simple series admittance, where the two ports are labeled with f and t. For line i with series admittance y_s^i, we have

$$\begin{bmatrix} \mathrm{i}_f^i \\ \mathrm{i}_t^i \end{bmatrix} = \begin{bmatrix} \mathrm{y}_s^i & -\mathrm{y}_s^i \\ -\mathrm{y}_s^i & \mathrm{y}_s^i \end{bmatrix} \begin{bmatrix} \mathrm{v}_f^i \\ \mathrm{v}_t^i \end{bmatrix}. \tag{7.9}$$

The individual admittance parameters for the n_k individual lines are then stacked as follows,

$$\mathbf{Y}_s = \begin{bmatrix} \mathrm{y}_s^1 & & & \\ & \mathrm{y}_s^2 & & \\ & & \ddots & \\ & & & \mathrm{y}_s^{n_k} \end{bmatrix}, \tag{7.10}$$

to form the admittance matrix parameter $\underline{\mathbf{Y}}$ that we see in (7.4) for the corresponding element object.

$$\underline{\mathbf{Y}} = \begin{bmatrix} \mathbf{Y}_s & -\mathbf{Y}_s \\ -\mathbf{Y}_s & \mathbf{Y}_s \end{bmatrix} \tag{7.11}$$

Stacking the individual port current and voltage variables,

$$\mathbf{i}_f = \begin{bmatrix} \mathrm{i}_f^1 \\ \mathrm{i}_f^2 \\ \vdots \\ \mathrm{i}_f^{n_k} \end{bmatrix}, \ \mathbf{i}_t = \begin{bmatrix} \mathrm{i}_t^1 \\ \mathrm{i}_t^2 \\ \vdots \\ \mathrm{i}_t^{n_k} \end{bmatrix}, \ \mathbf{v}_f = \begin{bmatrix} \mathrm{v}_f^1 \\ \mathrm{v}_f^2 \\ \vdots \\ \mathrm{v}_f^{n_k} \end{bmatrix}, \ \mathbf{v}_t = \begin{bmatrix} \mathrm{v}_t^1 \\ \mathrm{v}_t^2 \\ \vdots \\ \mathrm{v}_t^{n_k} \end{bmatrix}, \tag{7.12}$$

results in the port injection currents from (7.8) for this aggregate element taking the form

$$\mathbf{g}^I(\mathbf{x}) = \mathbf{i}^{lin}(\mathbf{x}) = \begin{bmatrix} \mathbf{i}_f \\ \mathbf{i}_t \end{bmatrix} = \underline{\mathbf{Y}} \begin{bmatrix} \mathbf{v}_f \\ \mathbf{v}_t \end{bmatrix} = \underline{\mathbf{Y}}\mathbf{v}. \tag{7.13}$$

When building its parameters, each network model element object also defines an element-node incidence matrix C for each of its ports and an element-variable incidence matrix D for each non-voltage states. For example, a transmission line element would define two C matrices, one mapping branches to their corresponding *from* bus and the other to their corresponding *to* bus.

7.3.3 Aggregation

Since the model parameters are consistent across all network model elements for a given formulation, and the connectivity of the elements is captured in the C and D incidence matrices for each element type, the network model object can assemble the parameters from all elements into a single aggregate network model characterized by parameters of the same form. This aggregate model can then be used to compute port or node injections from the aggregate system state, as well as any needed derivatives of these injection functions.

For more details on how the aggregation is done, see [TN5].

8 Mathematical Model Object

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 and the formulation.

Power Flow

The *power flow* problem involves solving a system of nonlinear equations for the vector $\boldsymbol{x}$.

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \tag{8.1}$$

For the DC version, the function $\boldsymbol{f}(\boldsymbol{x})$ is linear, so the problem takes the more specific form,

$$\underline{\boldsymbol{A}}\boldsymbol{x} - \underline{\boldsymbol{b}} = \boldsymbol{0}. \tag{8.2}$$

Continuation Power Flow

The *continuation power flow* problem involves tracing the solution curve for a parameterized system of equations, as the parameter λ is varied.

$$\boldsymbol{f}(\boldsymbol{x}, \lambda) = \boldsymbol{0}, \tag{8.3}$$

Optimal Power Flow

The *optimal power flow* problem, on the other hand, is a constrained optimization problem of the form,

$$\begin{aligned} &\min_{\boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}) \\ \text{such that} \quad &\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0} \\ &\boldsymbol{h}(\boldsymbol{x}) \leq \boldsymbol{0} \\ &\underline{\boldsymbol{x}}_{\min} \leq \boldsymbol{x} \leq \underline{\boldsymbol{x}}_{\max}. \end{aligned} \tag{8.4}$$

This reduces to a simple quadratic program (QP) for the DC OPF case,

$$\begin{aligned} &\min_{\boldsymbol{x}} \boldsymbol{x}^{\mathsf{T}}\underline{\boldsymbol{Q}}\boldsymbol{x} + \underline{\boldsymbol{c}}^{\mathsf{T}}\boldsymbol{x} + \underline{k} \\ \text{such that} \quad &\underline{\boldsymbol{l}} \leq \underline{\boldsymbol{A}}\boldsymbol{x} \leq \underline{\boldsymbol{u}} \\ &\underline{\boldsymbol{x}}_{\min} \leq \boldsymbol{x} \leq \underline{\boldsymbol{x}}_{\max}. \end{aligned} \tag{8.5}$$

8.1 Mathematical Models

A math model object is a container for math model element objects and it is also an MP-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 as illustrated in Figure 8.1, and sometimes inherit from abstract mix-in classes that are shared across tasks or formulations. These shared classes are described further in Section 8.3.

Figure 8.1: Math Model Classes

By convention, math model variables are named `mm` and math model class names begin with `mp.math_model`.

8.1.1 Building a Mathematical Model

A math model object is created in two steps. The first is to call the constructor of the desired math model class, without arguments. This initializes the `element_classes` property with a list of math model element classes. This list can be modified before the second step, which is to call the `build()` method, passing in the network and data model objects and a MATPOWER options struct.

```
mm = mp.math_model_opf_acps();
mm.build(nm, dm, mpopt);
```

The `build()` method proceeds through the following stages sequentially, looping through each element for the last 3 stages.

1. **Create** – Instantiate each element object.

2. **Count and add** - For each element object, determine the number of online elements from the corresponding data model element and, if nonzero, add the object to the `elements` property of the `mm`.
3. **Add auxiliary data** – Add auxiliary data, e.g. network node types, for use by the model.
4. **Add variables** – Add variables and allow each element to add their own variables to the model.
5. **Add constraints** – Add constraints and allow each element to add their own constraints to the model.
6. **Add costs** – Add costs and allow each element to add their own costs to the model.

The adding of variables, constraints and costs to the model is done by the math model and model model element objects using the interfaces provided by MP-Opt-Model.

8.1.2 Solving a Math Model

Once the math model build is complete and it contains the full set of variables, constraints and costs for the model, the solver options are initialized by calling the `solve_opts()` method and then passed to the `solve()` method.

```
opt = mm.solve_opts(nm, dm, mpopt);
mm.solve(opt);
```

The `solve()` method, also inherited from MP-Opt-Model, invokes the appropriate solver based on the characteristics of the model and the options provided.

8.1.3 Updating Network and Data Models

The solved math model can then be used to update the solved state of the network and data models by calling the `network_model_x_soln()` and `data_model_update()` methods, respectively.

```
nm = mm.network_model_x_soln(nm);
dm = mm.data_model_update(nm, dm, mpopt);
```

The math model's `data_model_update()` method cycles through the math model element objects, calling the `data_model_update()` for each element.

8.2 Mathematical Model Elements

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`. Figure 8.2 shows the inheritance relationships between a few example math model element classes. Here the `mp.nme_bus_pf_acp` and `mp.nme_bus_opf_acp` classes are used for PF and OPF problems, respectively, with an AC polar formulation. AC cartesian and DC formulations use their own respective task-specific subclasses. And each element type, has a similar set of task and formulation-specific subclasses, such as those for `mp.mme_gen`.

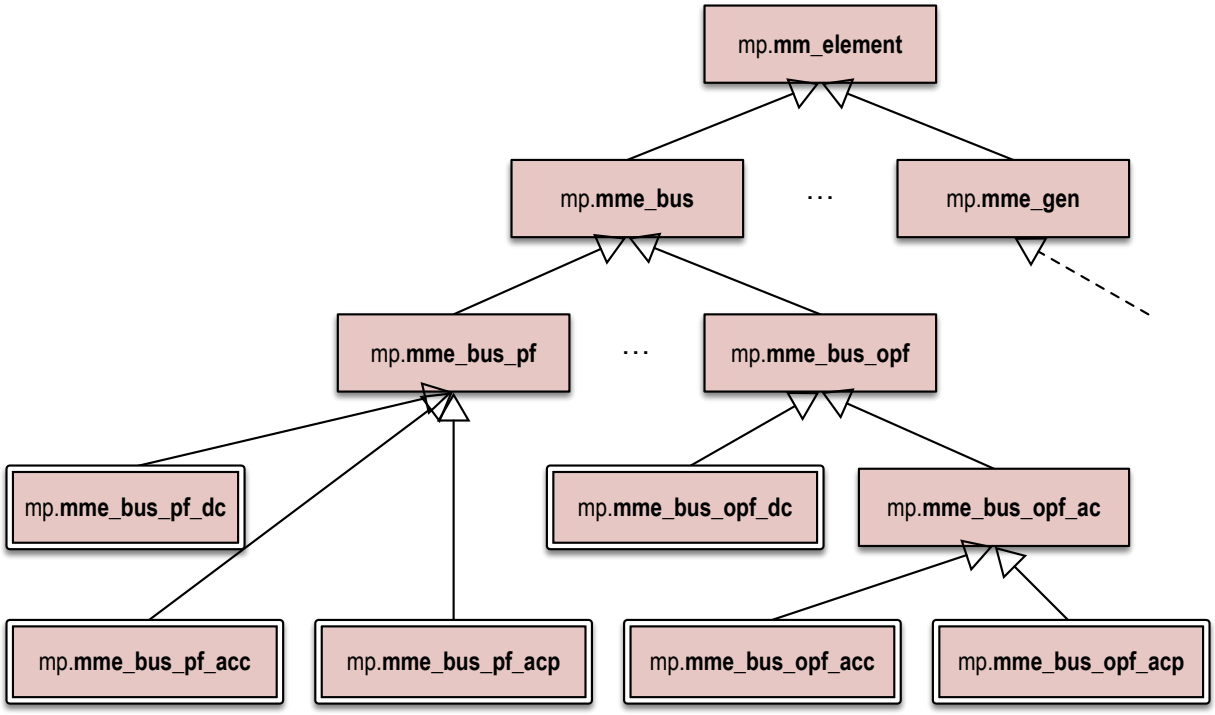

Figure 8.2: Math Model Element Classes

8.2.1 Adding Variables, Constraints, and Costs

Both the `mm` container object and the `mme` element objects can add their own variables, costs and constraints to the model.

For a standard optimal power flow, for example, the optimization variables are added by the container object, since they are determined directly from state variables of the *(container)* network model object. Similarly, the nodal power or current balance constraints are added by the container since they are built directly from the port injection functions of the aggregate network model.

However, generator cost functions and any variables and constraints associated with piecewise linear generator costs are added by the appropriate subclass of `mp.mme_gen`, since they relate only to generator model parameters. Similarly, branch flow and branch angle difference constraints are added by the appropriate subclass of `mp.mme_branch`, since they are specific to branches and are completely independent of other element types.

8.2.2 Updating Data Model Elements

The data in the data model is stored primarily in its individual element objects, so it makes sense that the individual math model element objects would be responsible for extracting the math model solution data relevant to a given element and updating the corresponding data model element. This updating is performed by the `data_model_update()` method.

The updating of each data model element is done in two steps. First `data_model_update()` calls `data_model_update_off()` to handle any offline units (e.g. to zero out any solution values), then `data_model_update_on()` to handle the online units.

For example, updating the branch power flows and shadow prices on the flow and angle difference limits in the branch data model element is done by `data_model_update_on()` in the appropriate subclass of `mp.mme_branch`.

8.3 Shared Classes

In some cases, there is code shared between math model classes across differnt tasks, e.g. PF and CPF. In order to avoid code duplication, another hierarchy of abstract mix-in classes is used to implement methods for this shared functionality. By convention, the names of these classes begin with `mp.mm_shared_`.

For example, a method to evaluate the node balance equations and corresponding Jacobian are used by both the PF and CPF. Putting this method in a shared class, allows its functionality to be inherited by concrete math model classes for both PF and CPF.

9 Customizing MATPOWER

With the *object-oriented MATPOWER core architecture* and its explicit three layer modeling outlined in Section 3, the flexibility and customizability of MATPOWER has increased dramatically.

New functionality can be added or existing functionality modified simply by adding new classes and/or subclassing existing ones. This approach can be used to add or modify elements, problem formulations, and tasks.

9.1 Default Class Determination

In order to customize the behavior it is important to understand how MATPOWER selects which classes to instantiate when running a particular task. There are default specifications for each of the various types of objects, as well as several ways to override those defaults. The default, described below, is illustrated in Figure 9.1.

9.1.1 Task Class

First of all, at the top level, the **task class** is specified directly by the user through the function used to invoke the run. In fact, `run_pf()`, `run_cpf()`, and `run_opf()` are simple one-line wrappers around the `run_mp()` function. The only difference between the three is the value of the `task_class` argument, a handle to the corresponding task constructor, passed into `run_mp()`.

This means that a new task class can be used simply by invoking `run_mp()`, either directly or via a new wrapper, with the task constructor as the `task_class` argument.

9.1.2 Model and Data Converter Classes

The task class has methods that determine the classes used to create the data model converter and the three model objects. For each, there are two methods involved in determining the specific class to use, a *main* method and a *default* method. The *main* method calls the *default* method to get a handle to the constructor for the task's default class, but then allows that value to be overridden by MATPOWER extensions or MATPOWER options.

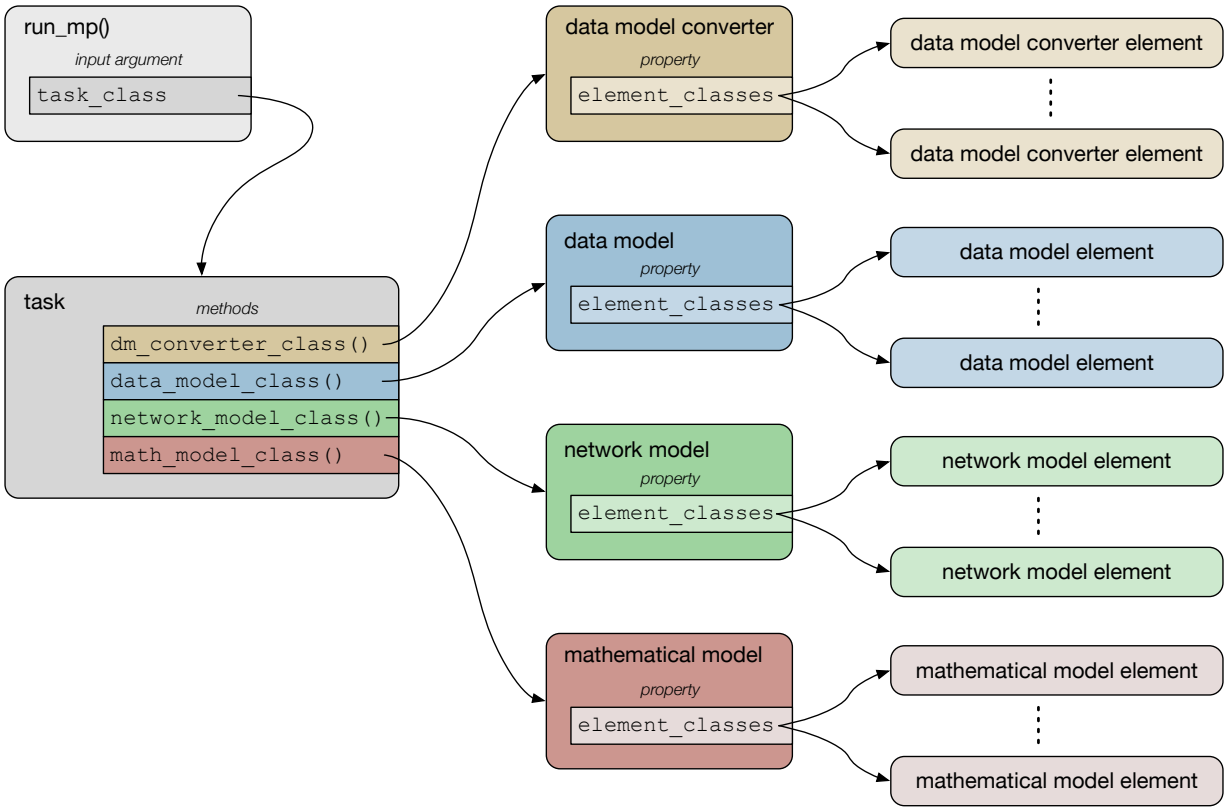

Figure 9.1: Determination of Default Classes

Table 9.1: Class Specification Methods of a Task

Method	Description
`dm_converter_class()`	Returns the final class for the data model converter, after any overrides of the default.
`data_model_class()`	Returns the final class for the data model, after any overrides of the default.
`network_model_class()`	Returns the final class for the network model, after any overrides of the default.
`math_model_class()`	Returns the final class for the math model, after any overrides of the default.
`dm_converter_class_mpc2_default()`	Returns the *default* class for the data model converter for this task. Note that this is specific to the data format. Each data format would have it's own "default" method.
`data_model_class_default()`	Returns the *default* class for the data model for this task.
`network_model_class_default()`	Returns the *default* class for the network model for this task.
`math_model_class_default()`	Returns the *default* class for the math model for this task.

Table 9.1 shows the methods that determine the classes for each of the 4 objects. Each method returns a handle to a class constructor. In general, the *main* methods (the first 4 in the table) are inherited from `mp.task` and only the *default* methods (the last 4) would be overridden to customize a task with new model or data model converter classes.

9.1.3 Element Classes

Each of the element container objects, that is the data model converter and the 3 model objects, contains a set of *elements*. The classes used to construct these elements are determined by the container class. Each container class inherits from `mp.element_container`, and as such it has an `element_classes` property, which is a cell array populated by the constructor with handles to constructors for the elements. This means that a container subclass can, by overriding its constructor, modify the list of element classes provided by its parent.

The elements are instantiated by a call to the container object's `build()` method, so the resulting set can be customized at runtime by modifying the list in `element_classes` after the container object is created and before its `build()` method is called.

This is done using **element class modifiers**, specified either by MATPOWER extensions or MATPOWER options. There are 3 types of element class modifiers, for adding, deleting or replacing an entry in an `element_classes` property. The 3 types are described in Table 9.2.

Table 9.2: Element Class Modifiers

Action	Value	Description
add	`@new_class`	Appends `@new_class` to the end of the list.
delete	`'old_class'`	For each element `E` in the list, if `isa(E(), 'old_class')` is true, element `E` is deleted from the list.
replace	`{@new_class, 'old_class'}`	For each element `E` in the list, if `isa(E(), 'old_class')` is true, element `E` is replaced with `@new_class`.

Typically, multiple element class modifiers can be provided in a cell array and they are processed sequentially to modify the existing list by the `modify_element_classes()` from `mp.element_container`.

9.2 Customization via MATPOWER Options

In addition to the MATPOWER options previously available that affect the formulation of the problem (e.g. polar vs. cartesian voltage representation, or current vs. power balance), there are several experimental options that can be used to directly modify the classes coming from the default class selection process outlined above. These options, summarized in Table 9.3, are specified by assigning them directly to an existing MATPOWER options struct `mpopt` as optional fields in `mpopt.exp`. They must be assigned directly, since `mpoption()` does not recognize them.

Table 9.3: Class Customization Options

Option	Description
`dm_converter_class`	function handle for data model converter constructor
`data_model_class`	function handle for data model constructor
`network_model_class`	function handle for network model constructor
`math_model_class`	function handle for math model constructor
`dmc_element_classes`	element class modifier(s)[1] for data model converter elements
`dm_element_classes`	element class modifier(s)[Page 38, 1] for data model elements
`nm_element_classes`	element class modifier(s)[1] for network model elements
`mm_element_classes`	element class modifier(s)[1] for math model elements
`exclude_elements`	cell array of names of elements to exclude from all four container objects, i.e. char arrays that match the `name` property of the element(s) to be excluded

9.3 MATPOWER Extensions

The *flexible MATPOWER framework* summarized in Section 3.3 introduces a **MATPOWER extension** API as a way to bundle a set of class additions and modifications together into a single named package.

For example, the `mp.xt_reserves` class and those it references, adds co-optimization of fixed zonal reserves to the standard OPF problem, as previously implemented by `toggle_reserves()` and `run_opf_w_res()` in MATPOWER 7.1 and earlier using its legacy OPF callback functions. To invoke an OPF with the `mp.xt_reserves` extension, you simply pass the extension object as an optional argument into the `run_opf()` function.

```
run_opf(mpc, mpopt, 'mpx', mp.xt_reserves);
```

A MATPOWER extension is a subclass of `mp.extension`, which implements a very simple interface consisting of nine methods. Five of them return a single class constructor handle, and the other four return a cell array of element class modifiers, described above in Table 9.2.

The methods are summarized in Table 9.4

Table 9.4: MATPOWER Extension Methods

Method	Description
`task_class()`	Returns a handle to the constructor for the task object.
`dm_converter_class()`	Returns a handle to the constructor for the data model converter.
`data_model_class()`	Returns a handle to the constructor for the data model.
`network_model_class()`	Returns a handle to the constructor for the network model.
`math_model_class()`	Returns a handle to the constructor for the math model.
`dmc_element_classes()`	Returns a cell array of element class modifiers for data model converter elements.

continues on next page

[1] Either a single element class modifier or a cell array of element class modifiers.

Table 9.4 – continued from previous page

Method	Description
dm_element_classes()	Returns a cell array of element class modifiers for data model elements.
nm_element_classes()	Returns a cell array of element class modifiers for network model elements.
mm_element_classes()	Returns a cell array of element class modifiers for math model elements.

Even something as complex as adding three-phase unbalanced buses, lines, loads and generators for multiple formulations of PF, CPF, and OPF problems can be implemented in terms of a single MATPOWER extension. Please see mp.xt_3p for an example.

10 Acknowledgments

The MATPOWER team would like to acknowledge the support of the numerous research grants and contracts that have contributed directly and indirectly to the development of MATPOWER over the years. This includes funding from the Power Systems Engineering Research Center (PSERC), the U.S. Department of Energy,[1] the National Science Foundation,[2] ARPA-E[3] and others.

[1] Supported in part by the Consortium for Electric Reliability Technology Solutions (CERTS) and the Office of Electricity Delivery and Energy Reliability, Transmission Reliability Program of the U.S. Department of Energy under the National Energy Technology Laboratory Cooperative Agreement No. DE-FC26-09NT43321.

[2] This material is based upon work supported in part by the National Science Foundation under Grant Nos. 0532744, 1642341 and 1931421. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[3] Supported in part by the "Synthetic Data for Power Grid R & D" project under the ARPA-E GRID DATA program.

11

References

Bibliography

[CTM] Carleton Coffrin, et. al., "The Common Electric Power Transmission System Model," *work in progress*. Available at: https://www.overleaf.com/project/5d94e3765cb3ba000129df3c.

[TN5] R. D. Zimmerman, "MP-Element: A Unified MATPOWER Element Model," *MATPOWER Technical Note 5*, October 2020. Available: https://matpower.org/docs/TN5-MP-Element.pdf