User Manual
PGMcpp : PRIMED Grid Modelling Code (in C++) - v2.1

November 6, 2023

Drafted using LaTeX

License

Contents

List of Tables

1

Introduction

The intent of PGMcpp is to provide a general purpose code base for the modelling and simulation of microgrids, with the particular goal of assessing the economic and environmental impacts of integrating renewable energy generation and energy storage assets over some project life. It is designed to be open and extensible, so that the researcher can modify it to suit their individual needs.

This user manual will go over the features of PGMcpp, including the various classes that make up the code base, and will conclude with a treatment of dispatch control and the presentation of an example project.

2

The `Model` Class

<u>Header</u>: `header/Model.h`
<u>Source</u>: `source/Model.cpp`

The central class of PGMcpp is the `Model` class, which is designed to act as a container class for the electrical demand data, renewable resource data, various production and storage assets, and dispatch control strategies.

2.1 Attributes

Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Model` class constructor. This structure is defined as follows

```
struct structModel {
    bool print_flag = false;
    bool test_flag = false;

    DispatchMode dispatch_mode = LOAD_FOLLOWING_IN_ORDER;

    double nominal_inflation_rate_annual = 0.02;
    double nominal_discount_rate_annual = 0.04;

    std::string path_2_load_data = "";
};
```

Each attribute of `structModel` is described below in Table 2.1.

Table 2.1: `structModel` attribute descriptions

Attribute	**Type**	**Default Value**	**Description**
`print_flag`	boolean	false	Controls whether or not running prints are done.
`test_flag`	boolean	false	Toggles testing printouts.
`dispatch_mode`	DispatchMode	LOAD_FOLLOWING_IN_ORDER	Defines which dispatch control mode to use during a `Model` run. More on this later.
`nominal_inflation_rate`	double	0.02	The nominal inflation rate (annual) to be applied to the entire model.
`nominal_discount_rate`	double	0.04	The nominal discount rate (annual) to be applied to the entire model.
`path_2_load_data`	string	empty	The (relative) path to the file containing the intended electrical load data (time series).

Note that these structured attributes are the only `Model` attributes which are intended to be modified by the user; all other attributes of the class (while nonetheless public) are not intended to be modified by the user *under normal execution*. That said, all attributes are, of course, intended to be read by the user.

The complete set of `Model` attributes is described below in Table 2.2.

Table 2.2: `Model` attribute descriptions

Attribute	Type	Default Value	Description
`struct_model`	structModel	as laid out in Table 2.1	The structure of inputs required by the `Model` class constructor.
`n_timesteps`	integer	8760	The number of time steps over which the model will run (inferred during construction).
`project_life_yrs`	double	0	The project life, in years, to be modelled (inferred during construction).
`total_load_served_kWh`	double	0	The total load (energy) served during a model run, expressed in kiloWatt-hours.
`total_fuel_consumed_L`	double	0	The total volume of fuel consumed during a model run, expressed in litres.
`total_CO2_emitted_kg`	double	0	The total mass of carbon dioxide emitted during a model run, expressed in kilograms.
`total_CO_emitted_kg`	double	0	The total mass of carbon monoxide emitted during a model run, expressed in kilograms.
`total_NOx_emitted_kg`	double	0	The total mass of nitrogen oxides emitted during a model run, expressed in kilograms.
`total_SOx_emitted_kg`	double	0	The total mass of sulfur oxides emitted during a model run, expressed in kilograms.
`total_CH4_emitted_kg`	double	0	The total mass of methane emitted during a model run, expressed in kilograms.
`total_PM_emitted_kg`	double	0	The total mass of particulate matter emitted during a model run, expressed in kilograms.
`real_discount_rate_annual`	double	0	The real discount rate (annual) to be applied to the entire model (computed from the given nominal inflation and discount rates).
`net_present_cost`	double	0	The net present cost of the system, as computed following a model run. The units of currency are left undefined.
`levellized_cost_of_energy_per_kWh`	double	0	The levellized cost of energy (per kiloWatt-hour served), as computed following a model run. The units of currency are left undefined.
`dt_vec_hr`	vector of doubles	empty	A sequence of time deltas, expressed in hours (inferred during construction).

Attribute	Type	Default Value	Description
load_vec_kW	vector of doubles	empty	A sequence of load values, expressed in kiloWatt-hours. This is read from the given load data.
net_load_vec_kW	vector of doubles	empty	A sequence of net load values, expressed in kiloWatt-hours. The net load over a given time step is the load minus all renewable production, and so this attribute is computed as part of a model run.
remaining_load_vec_kW	vector of doubles	empty	A sequence of remaining load values, expressed in kiloWatt-hours, which are computed are part of a model. If any load is left unsatisfied by the grid design, it is recorded here.
time_vec_hr	vector of doubles	empty	A sequence of time values (from time = 0, taken to be the start of the project), expressed in hours. This is read in from the given load data.
resource_map_1D	map of (integer, vector of doubles) pairs	empty	A map containing the various one-dimensional renewable resources in play. This map is populated from data provided by the user.
resource_path_map_1D	map of (integer, string) pairs	empty	A map containing the paths to the one-dimensional resource data provided by the user.
resource_map_2D	map of (integer, vector of vectors of doubles) pairs	empty	A map containing the various two-dimensional renewable resources in play. This map is populated from data provided by the user.
resource_path_map_2D	map of (integer, string) pairs	empty	A map containing the paths to the two-dimensional resource data provided by the user.
nondisp_ptr_vec	vector of pointers to Nondispatchable	empty	A vector containing pointers to the various Nondispatchable assets added to the model.
combustion_ptr_vec	vector of pointers to Combustion	empty	A vector containing pointers to the various Combustion (Dispatchable) assets added to the model.
noncombustion_ptr_vec	vector of pointers to Dispatchable	empty	A vector containing pointers to the various non-Combustion (Dispatchable) assets added to the model.
storage_ptr_vec	vector of pointers to Storage	empty	A vector containing pointers to the various Storage assets added to the model.

It is important to understand one of the key modelling constraints built into PGMcpp. That is, *all time series are taken to be commensurate with the given load data*. That is, every time series is required to be associated with the same `time_vec_hr`, so all input data must be prepared accordingly (and this is checked for and enforced by the model!). However, having satisfied that constraint, one can make the time series data as short or as long as desired. Furthermore, the time series need not be uniform (that is, the time deltas can vary throughout the time series).

2.2 Methods

The `Model` class has a number of helper methods (denoted by the leading _ in the method name) which are not intended to be called directly (that is, they do not make up part of the intended user interface). You can learn more about these methods by reviewing the header and source files for `Model`. In this section, the methods that make up the intended user interface are presented, as these methods are the only ones the user should need to work with *under normal execution*.

2.2.1 `Model( )` (Constructor)

```
Model :: Model(structModel struct_model)
```

This is the constructor for the `Model` class, and it expects a `structModel` instance as its sole argument. A minimal working example of invoking the class constructor is as follows

```
structModel model_inputs;
model_inputs.path_2_load_data =
    "data/input/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

Model model(model_inputs);
```

The user must provide a valid (relative) path to electrical load data at construction, and the load data must have the expected format. For an example of the expected format, see

```
data/input/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv
```

2.2.2 `add1dRenewableResource( )`

```
void Model :: add1dRenewableResource(
    std::string type_str,
    std::string path_2_resource_data,
    int map_key
)
```

This method loads a one-dimensional renewable resource into an existing `Model` instance, where "one-dimensional renewable resource" means a time series of scalar resource values (i.e., one value per point in time). It expects three arguments

1. `std::string type_str`, a string denoting the resource type (i.e., solar, wind, tidal, etc.). This argument is not case sensitive, but it is sensitive to spelling.

2. `std::string path_2_resource_data`, a string denoting the (relative) path to the appropriate one-dimensional resource time series.

3. `int map_key`, an integer that forms the key part of a (key, value) pair. This is the key used when inserting the given data and path into `resource_map_1D` and `resource_path_map_1D`, respectively.

A minimal working example of invoking this method is as follows (e.g., adding a time series of solar resource)

```
int solar_resource_key = 1;

model.add1dRenewableResource(
    "solar",
    "data/input/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv",
    solar_resource_key
);
```

The user must provide a valid (relative) path to resource data, and the data must have the expected format. For examples of the expected format, see the example resource files provided in

```
data/input/test/
```

Finally, if the user re-uses the same map key argument in successive calls to this method, then the corresponding contents of the one-dimensional resource maps are overwritten on each call (this will generate a warning).

2.2.3 add2dRenewableResource( )

```
void Model :: add2dRenewableResource(
    std::string type_str,
    std::string path_2_resource_data,
    int map_key
)
```

This method loads a two-dimensional renewable resource into an existing `Model` instance, where "two-dimensional renewable resource" means a time series of vector resource values (i.e., two values per point in time). It expects three arguments

1. `std::string type_str`, a string denoting the resource type (i.e., wave). This argument is not case sensitive, but it is sensitive to spelling.

2. `std::string path_2_resource_data`, a string denoting the (relative) path to the appropriate two-dimensional resource time series.

3. `int map_key`, an integer that forms the key part of a (key, value) pair. This is the key used when inserting the given data and path into `resource_map_2D` and `resource_path_map_2D`, respectively.

A minimal working example of invoking this method is as follows (e.g., adding a time series of wave resource)

```
int wave_resource_key = 4;

model.add2dRenewableResource(
    "wave",
    "data/input/test/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv",
    wave_resource_key
);
```

The user must provide a valid (relative) path to resource data, and the data must have the expected format. For examples of the expected format, see the example resource files provided in

```
data/input/test/
```

Finally, if the user re-uses the same map key argument in successive calls to this method, then the corresponding contents of the two-dimensional resource maps are overwritten on each call (this will generate a warning).

2.2.4 addSolar( )

```
void Model :: addSolar(
    structNondispatchable struct_nondisp,
    structSolar struct_solar
)
```

This method adds an instance of the `Solar` class to an existing `Model` instance. For more info on the `Solar` class, including the relevant input structures, refer to the chapter on the `Nondispatchable` class hierarchy. A minimal working example of invoking this method is as follows

```
structNondispatchable nondisp_inputs;

structSolar solar_inputs;
solar_inputs.resource_key = solar_resource_key;

model.addSolar(nondisp_inputs, solar_inputs);
```

Upon invocation, a pointer to the constructed `Solar` instanced is pushed onto the back of the `Model` `nondisp_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.5 addTidal( )

```
void Model :: addTidal(
    structNondispatchable struct_nondisp,
    structTidal struct_tidal
)
```

This method adds an instance of the `Tidal` class to an existing `Model` instance. For more info on the `Tidal` class, including the relevant input structures, refer to the chapter on the `Nondispatchable` class hierarchy. A minimal working example of invoking this method is as follows

```
structNondispatchable nondisp_inputs;

structTidal tidal_inputs;
tidal_inputs.resource_key = tidal_resource_key;

model.addTidal(nondisp_inputs, tidal_inputs);
```

Upon invocation, a pointer to the constructed `Tidal` instanced is pushed onto the back of the `Model` `nondisp_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.6 addWave( )

```
void Model :: addWave(
    structNondispatchable struct_nondisp,
    structWave struct_wave
)
```

This method adds an instance of the `Wave` class to an existing `Model` instance. For more info on the `Wave` class, including the relevant input structures, refer to the chapter on the `Nondispatchable` class hierarchy. A minimal working example of invoking this method is as follows

```
structNondispatchable nondisp_inputs;

structWave wave_inputs;
wave_inputs.resource_key = wave_resource_key;

model.addWave(nondisp_inputs, wave_inputs);
```

Upon invocation, a pointer to the constructed `Wave` instanced is pushed onto the back of the `Model` `nondisp_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.7 addWind( )

```
void Model :: addWind(
    structNondispatchable struct_nondisp,
    structWind struct_wind
)
```

This method adds an instance of the `Wind` class to an existing `Model` instance. For more info on the `Wind` class, including the relevant input structures, refer to the chapter on the `Nondispatchable` class hierarchy. A minimal working example of invoking this method is as follows

```
structNondispatchable nondisp_inputs;

structWind wind_inputs;
wind_inputs.resource_key = wind_resource_key;

model.addWind(nondisp_inputs, wind_inputs);
```

Upon invocation, a pointer to the constructed `Wind` instanced is pushed onto the back of the `Model` `nondisp_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.8 addDiesel( )

```
void Model :: addDiesel(
    structDispatchable struct_disp,
    structCombustion struct_combustion,
    structDiesel struct_diesel
)
```

This method adds an instance of the `Diesel` class to an existing `Model` instance. For more info on the `Diesel` class, including the relevant input structures, refer to the chapter on the `Dispatchable` class hierarchy. A minimal working example of invoking this method is as follows

```
structDispatchable disp_inputs;

structCombustion combustion_inputs;

structDiesel diesel_inputs;

model.addDiesel(disp_inputs, combustion_inputs, diesel_inputs);
```

Upon invocation, a pointer to the constructed `Diesel` instanced is pushed onto the back of the `Model` `combustion_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.9 addLiIon( )

```
void Model :: addLiIon(
    structStorage struct_storage,
    structBatteryStorage struct_battery_storage,
    structLiIon struct_liion
)
```

This method adds an instance of the `LiIon` class to an existing `Model` instance. For more info on the `LiIon` class, including the relevant input structures, refer to the chapter on the `Storage` class hierarchy. A minimal working example of invoking this method is as follows

```
structStorage storage_inputs;

structBatteryStorage batt_storage_inputs;

structLiIon liion_inputs;

model.addLiIon(storage_inputs, batt_storage_inputs, liion_inputs);
```

Upon invocation, a pointer to the constructed `LiIon` instanced is pushed onto the back of the `Model` `storage_ptr_vec` attribute. So, the order of elements in this attribute is entirely defined by the order in which methods of this type are invoked.

2.2.10 run( )

```
void Model :: run()
```

Once the user is finished setting up a `Model` instance, invoking this method does exactly what one might expect; it runs the energy modelling for this instance. Once the `run()` method has been invoked, all attributes of the `Model` instance are populated and ready for use. A minimal working example of invoking this method is as follows

```
model.run();
```

2.2.11 writeResults( )

```
void Model :: writeResults(std::string project_name)
```

This method writes the results of a model run to the disk, and it expects a project name as its sole argument. A minimal working example of invoking this method is as follows

```
model.writeResults("example_project");
```

Upon invocation, the attributes of the `Model` instance, as well as the attributes of any contained `Nondispatchable`, `Dispatchable`, and `Storage` instances, are written to

```
data/output/<project_name>/
```

If this directory does not exist at invocation time, then it is created. If this directory already exists at invocation time, then it is overwritten (as in deleted and rebuilt); this will generate a warning.

2.2.12 clearAssets( )

```
void Model :: clearAssets()
```

This method clears all `Nondispatchable`, `Dispatchable`, and `Storage` instances from the `Model` instance. That is to say, invocation clears and resets the following (and only the following) attributes of the `Model` instance

1. `nondisp_ptr_vec`
2. `combustion_ptr_vec`
3. `noncombustion_ptr_vec`
4. `storage_ptr_vec`

This method is called automatically by the `Model` class destructor.

2.2.13 reset( )

```
void Model :: reset()
```

This method invokes `clearAssets( )`, and then resets the following (and only the following) attributes of the `Model` instance

1. `total_load_served_kWh`
2. `total_fuel_consumed_L`
3. `total_CO2_emitted_kg`
4. `total_CO_emitted_kg`
5. `total_NOx_emitted_kg`
6. `total_SOx_emitted_kg`
7. `total_CH4_emitted_kg`
8. `total_PM_emitted_kg`
9. `net_present_cost`
10. `levellized_cost_of_energy_per_kWh`
11. `net_load_vec_kW`
12. `remaining_load_vec_kW`

The intent of this method is to reset the `Model` instance for a different set up and `run()` using the same load and resource data. That way, it avoids having to reconstruct a new `Model` instance (and hence reload all the load and resource data) every time the user wants to model a new grid design.

3

The `Nondispatchable` Class Hierarchy

The `Nondispatchable` class hierarchy is where the modelling of nondispatchable (as in noncontrollable) production assets, namely renewable production assets, is implemented. The class hierarchy is organized as follows

```
Nondispatchable
    <-- Solar
    <-- Tidal
    <-- Wave
    <-- Wind
```

That is, the `Nondispatchable` class is parent to each of the `Solar`, `Tidal`, `Wave`, and `Wind` classes. This hierarchy will be expanded and updated as new `Nondispatchable` assets are added to PGMcpp.

All attributes of the `Nondispatchable` objects that make up a `Model` instance will be written to the disk upon invoking `Model :: writeResults( )`. However, should you need access to attribute values within your program, you can always get a pointer to the n^{th} `Nondispatchable` that was added to the `Model` by way of

```
Nondispatchable* nondisp_ptr = model.nondisp_ptr_vec[n];
```

3.1 The `Nondispatchable` Class

Header: `header/assets/nondispatchable/Nondispatchable.h`
Source: `source/assets/nondispatchable/Nondispatchable.cpp`

The `Nondispatchable` class is the root of its class hierarchy. As such, it is intended to define the attributes and methods common to all members of the hierarchy. Interacting directly with its methods is not intended *under normal execution*; please refer to the header and source files for more information on these methods. That said, this class is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Nondispatchable` class constructor. This structure is defined as follows

```
struct structNondispatchable {
    bool is_sunk = false;
    bool print_flag = false;
    bool test_flag = false;

    double cap_kW = 100;
    double replace_running_hrs = 90000;

    double nominal_inflation_rate_annual = 0.02;
    double nominal_discount_rate_annual = 0.04;

    double capital_cost = -1;
    double op_maint_cost_per_kWh = -1;
};
```

Each attribute of `structNondispatchable` is described below in Table 3.1.

Table 3.1: `structNondispatchable` attribute descriptions

Attribute	**Type**	**Default Value**	**Description**
`is_sunk`	boolean	false	Defines whether or not the object is modelled as a sunk cost.
`print_flag`	boolean	false	Controls whether or not running prints are done.
`test_flag`	boolean	false	Toggles testing printouts.
`cap_kW`	double	100	Defines the rated power production capacity, expressed in kiloWatts, of the object.
`replace_running_hours`	double	90000	Defines the number of running hours at which a replacement of the object is triggered.
`nominal_inflation_rate_annual`	double	0.02	Defines the nominal inflation rate (annual), used in computing economic values for the object.
`nominal_discount_rate_annual`	double	0.04	Defines the nominal discount rate (annual), used in computing economic values for the object.
`capital_cost`	double	-1 (sentinel)	Defines the capital cost of the object. The default value of -1 is a sentinel value, which triggers a generic capital cost model within the constructor.
`op_maint_cost_per_kWh`	double	-1 (sentinel)	Defines the operation and maintenance costs, per kiloWatt-hour produced, of the object. The default value of -1 is a sentinel value, which triggers a generic operation and maintenance cost model within the constructor.

Additionally, the following enumeration of the `Nondispatchable` class hierary is defined

```
enum NondispatchableType {
    SOLAR,
    TIDAL,
    WAVE,
    WIND
};
```

The complete set of `Nondispatchable` attributes is described below in Table 3.2.

Table 3.2: `Nondispatchable` attribute descriptions

Attribute	Type	Default Value	Description
`nondisp_type`	NondispatchableType	SOLAR	Defines the `Nondispathcable` type of the object.
`struct_nondisp`	structNondispatchable	as laid out in Table 3.1	The structure of inputs required by the `Nondispatchable` class constructor.
`is_running`	boolean	false	Tracks whether the object is running (i.e., in operation) or not.
`n_timesteps`	integer	0	The number of time steps over which the model will run (same as the corresponding `Model` attribute).
`n_replacements`	integer	0	Tracks the number of times the object has been replaced.
`project_life_yrs`	double	0	The project life, in years, to be modelled.
`running_hrs`	double	0	Tracks the running hours of the object.
`total_dispatch_kWh`	double	0	Tracks the total energy, expressed in kiloWatt-hours, that has been dispatched by the object.
`real_discount_rate_annual`	double	0	The real discount rate (annual) to be applied to the object (computed from the given nominal inflation and discount rates).
`net_present_cost`	double	0	The net present cost of the object, as computed following a model run. The units of currency are left undefined.
`levellized_cost_of_energy_per_kWh`	double	0	The levellized cost of energy (per kiloWatt-hour dispatched), as computed following a model run. The units of currency are left undefined.
`nondisp_type_str`	string	empty	A string corresponding to the `nondisp_type` attribute.
`is_running_vec`	vector of booleans	empty	A vector which records which time steps the object was running (i.e., in operation) for.
`replaced_vec`	vector of booleans	empty	A vector which records replacements of the object.

Attribute	Type	Default Value	Description
production_vec_kW	vector of doubles	empty	A vector which records the production, in kiloWatts, of the object over each time step.
dispatch_vec_kW	vector of doubles	empty	A vector which records the dispatch, in kiloWatts, of the object over each time step.
curtailment_vec_kW	vector of doubles	empty	A vector which records how much excess production (if any), in kiloWatts, is curtailed over each time step.
storage_vec_kW	vector of doubles	empty	A vector which records how much excess production (if any), in kiloWatts, is stored over each time step.
real_capital_cost_vec	vector of doubles	empty	A vector which records the real capital costs incurred over each time step. The units of currency are left undefined.
real_opt_maint_cost_vec	vector of doubles	empty	A vector which records the real operation and maintenance costs incurred over each time step. The units of currency are left undefined.
ptr_2_dt_vec_hr	pointer to vector of doubles	NULL	A pointer to the dt_vec_hr attribute of the Model.
ptr_2_time_vec_hr	pointer to vector of doubles	NULL	A pointer to the time_vec_hr attribute of the Model.

3.2 The Solar Class

Header: header/assets/nondispatchable/Solar.h
Source: source/assets/nondispatchable/Solar.cpp

The Solar class implements the modelling of a solar photovoltaic (PV) array. Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the Solar class constructor. This structure is defined as follows

```
struct structSolar {
    int resource_key = 0;

    double derating = 0.8;
    double capital_cost_per_kW = 3000;
};
```

Each attribute of structSolar is described below in Table 3.3.

Table 3.3: `structSolar` attribute descriptions

Attribute	Type	Default Value	Description
`resource_key`	integer	0	The key to the corresponding solar (one-dimensional) resource. This is the key used to index into the `resource_map_1D` attribute of the `Model`.
`derating`	double	0.8	This is the derating factor applied to the modelling of production under a given solar resource value.
`capital_cost_per_kW`	double	3000	This is the model capital cost, per kiloWatt installed capacity, used to compute the capital cost of the object. This value is only used if the generic capital cost model within the `Nondispatchable` class constructor is triggered.

The sole attribute of the `Solar` class is the provided `structSolar` instance. `Solar` production is modelled using a simple, derated linear model. See the source file for more details.

3.3 The `Tidal` Class

Header: `header/assets/nondispatchable/Tidal.h`
Source: `source/assets/nondispatchable/Tidal.cpp`

The `Tidal` class implements the modelling of a tidal turbine, or a tidal energy converter (TEC). Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Tidal` class constructor. This structure is defined as follows

```
struct structTidal {
    int resource_key = 0;

    TidalPowerCurve power_curve = CUBIC;

    double design_speed_ms = 2;
};
```

Each attribute of `structTidal` is described below in Table 3.4.

Table 3.4: `structTidal` attribute descriptions

Attribute	Type	Default Value	Description
`resource_key`	integer	0	The key to the corresponding tidal (one-dimensional) resource. This is the key used to index into the `resource_map_1D` attribute of the `Model`.
`power_curve`	TidalPowerCurve	CUBIC	Defines which generic power curve to use when modelling production under a given tidal resource value.
`design_speed_ms`	double	2	Defines the design speed, in metres per second, of the turbine. This is used to calibrate the selected generic power curve model.

The sole attribute of the `Tidal` class is the provided `structTidal` instance.

The modelling of `Tidal` production is handled in one of two ways, as enumerated by

```
enum TidalPowerCurve {
    CUBIC,
    EXPONENTIAL
};
```

If `structTidal :: power_curve` is CUBIC, then a generic cubic power curve model is employed. If `structTidal :: power_curve` is EXPONENTIAL, then a generic exponential power curve model is employed. See the source file for more details.

3.4 The `Wave` Class

Header: `header/assets/nondispatchable/Wave.h`
Source: `source/assets/nondispatchable/Wave.cpp`

The `Wave` class implements the modelling of a wave energy converter (WEC). Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Wave` class constructor. This structure is defined as follows

```
struct structWave {
    int resource_key = 0;

    WavePowerMode power_mode = PARABOLOID;

    double design_significant_wave_height_m = 2;
```

```
    double design_energy_period_s = 10;

    std::string path_2_normalized_performance_matrix = "";
};
```

Each attribute of `structWave` is described below in Table 3.5.

Table 3.5: `structWave` attribute descriptions

Attribute	**Type**	**Default Value**	**Description**
`resource_key`	integer	0	The key to the corresponding wave (two-dimensional) resource. This is the key used to index into the `resource_map_2D` attribute of the `Model`.
`power_mode`	WavePowerMode	PARABOLOID	Defines which approach to use when modelling production under a given wave resource.
`design_significant_wave_height_m`	double	2	The design significant wave height, expressed in metres, of the WEC. This is only used if `power_mode` = GAUSSIAN.
`design_energy_period_s`	double	10	The design energy period, expressed in seconds, of the WEC. This is only used if `power_mode` = GAUSSIAN.
`path_2_normalized_performance_matrix`	string	empty	The (relative) path to the file containing a normalized performance matrix for the WEC. This is only used if `power_mode` = NORMALIZED_PERF...

The complete set of `Wave` attributes is described below in Table 3.6.

Table 3.6: `Wave` attribute descriptions

Attribute	Type	Default Value	Description
`struct_wave`	structWave	as laid out in Table 3.5	The structure of inputs required by the `Wave` class constructor.
`min_interp_sig_wave_height_m`	double	0	The minimum significant wave height value, in metres, for the purpose of interpolating production. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`max_interp_sig_wave_height_m`	double	0	The maximum significant wave height value, in metres, for the purpose of interpolating production. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`min_interp_energy_period_s`	double	0	The minimum energy period value, in seconds, for the purpose of interpolating production. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`max_interp_energy_period_s`	double	0	The maximum energy period value, in seconds, for the purpose of interpolating production. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`interp_sig_wave_height_vec_m`	vector of doubles	empty	A vector of significant wave height values, in metres, for the purpose of interpolating production. Corresponds to the significant wave height values in the given normalized performance matrix. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`interp_energy_period_vec_s`	vector of doubles	empty	A vector of energy period values, in seconds, for the purpose of interpolating production. Corresponds to the energy period values in the given normalized performance matrix. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...
`interp_normalized_performance_matrix`	vector of vectors of doubles	empty	The normalized performance values from the given normalized performance matrix, for the purpose of interpolating production. This is only used if `structWave :: power_mode` = NORMALIZED_PERF...

The modelling of `Wave` production is handled in one of three ways, as enumerated by

```
enum WavePowerMode {
    GAUSSIAN,
    NORMALIZED_PERFORMANCE_MATRIX,
    PARABOLOID
};
```

If `structWave :: power_mode` is GAUSSIAN, then a generic Gaussian (i.e. bell curve) production model is employed. If `structWave :: power_curve` is NORMALIZED_PERFORMANCE_MATRIX, then production is interpolated (linearly) from the given normalized performance matrix. If `structWave :: power_mode` is PARABOLOID, then a generic paraboloid (i.e. quadratic form) production model is employed. See the source file for more details.

Finally, for the NORMALIZED_PERFORMANCE_MATRIX case, note that the normalized performance matrix is just that; a matrix of *normalized* performance values for the WEC, with values varying continuously from 0 (no production) to 1 (full, or rated, production). The normalized performance matrix must adhere to the expected format in order for PGMcpp to be able to read it. For an example of the expected format, see

```
data/input/test/normalized_performance_matrix.csv
```

3.5 The `Wind` Class

Header: `header/assets/nondispatchable/Wind.h`
Source: `source/assets/nondispatchable/Wind.cpp`

The `Wind` class implements the modelling of a wind turbine. Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Wind` class constructor. This structure is defined as follows

```
struct structWind {
    int resource_key = 0;

    double design_speed_ms = 8;
};
```

Each attribute of `structWind` is described below in Table 3.7.

Table 3.7: `structWind` attribute descriptions

Attribute	Type	Default Value	Description
`resource_key`	integer	0	The key to the corresponding wind (one-dimensional) resource. This is the key used to index into the `resource_map_1D` attribute of the `Model`.
`design_speed_ms`	double	8	Defines the design speed, in metres per second, of the turbine. This is used to calibrate the generic power curve model.

The sole attribute of the `Wind` class is the provided `structWind` instance. `Wind` production is modelled using a generic exponential power curve. See the source file for more details.

4

The Dispatchable Class Hierarchy

The `Dispatchable` class hierarchy is where the modelling of dispatchable (as in controllable) production assets is implemented. The class hierarchy is organized as follows

```
Dispatchable
    <-- Combustion
        <-- Diesel
```

That is, the `Dispatchable` class is parent to the `Combustion` class, which in turn is parent to the `Diesel` class. This hierarchy will be expanded and updated as new `Dispatchable` assets are added to PGMcpp.

All attributes of the `Dispatchable` objects that make up a `Model` instance will be written to the disk upon invoking `Model :: writeResults()`. However, should you need access to attribute values within your program, you can always get a pointer to the n^{th} `Dispatchable` that was added to the `Model` by way of (for example)

```
Dispatchable* disp_ptr = model.combustion_ptr_vec[n];
```

4.1 The Dispatchable Class

Header: `header/assets/dispatchable/Dispatchable.h`
Source: `source/assets/dispatchable/Dispatchable.cpp`

The `Dispatchable` class is the root of its class hierarchy. As such, it is intended to define the attributes and methods common to all members of the hierarchy. Interacting directly with its methods is not intended *under normal execution*; please refer to the header and source files for more information on these methods. That said, this class is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Dispatchable` class constructor. This structure is defined as follows

```
struct structDispatchable {
    bool is_sunk = false;
    bool print_flag = false;
```

```
        bool test_flag = false;

        double cap_kW = 100;
        double replace_running_hrs = 30000;

        double nominal_inflation_rate_annual = 0.02;
        double nominal_discount_rate_annual = 0.04;

        double capital_cost = -1;
        double op_maint_cost_per_kWh = -1;
    };
```

Note that this structure is exactly the same as `structNondispatchable`, so the attribute descriptions laid out in Table 3.1 also apply here. Additionally, the following enumeration of the `Dispatchable` class hierarchy is defined

```
    enum DispatchableType {
        DIESEL
    };
```

The complete set of `Dispatchable` attributes is nearly identical to the complete set of `Nondispatchable` attributes as laid out in Table 3.2. Where the `Dispatchable` attributes differ is described below in Table 4.1.

Table 4.1: `Dispatchable` attribute descriptions (where different from, or in addition to, the `Nondispatchable` attributes)

Attribute	Type	Default Value	Description
`disp_type`	DispatchableType	DIESEL	Defines the `Dispathcable` type of the object.
`struct_disp`	structDispatchable	as laid out in Table 3.1	The structure of inputs required by the `Dispatchable` class constructor.
`n_starts`	integer	0	Tracks the number of times the object was started.
`disp_type_str`	string	empty	A string corresponding to the `disp_type` attribute.

4.2 The `Combustion` Class

Header: `header/assets/dispatchable/combustion/Combustion.h`
Source: `source/assets/dispatchable/combustion/Combustion.cpp`

The `Combustion` class is the root of its branch of the class hierarchy. As such, it is intended to define the attributes and methods common to all members of the branch. Interacting directly with its methods is not intended *under normal execution*; please refer to the header and source files for more information on these methods. That said, this class is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Combustion` class constructor. This structure is defined as follows

```
struct structCombustion {
    FuelMode fuel_mode = LINEAR;

    double cycle_charging_load_ratio = 0.85;

    double ramp_rate_constraint_kWperhr = -1;   // sentinel

    double fuel_cost_L = 1.50;
    double nominal_fuel_escalation_rate_annual = 0.05;

    double linear_fuel_intercept_LkWh = -1; // sentinel
    double linear_fuel_slope_LkWh = -1;     // sentinel

    std::string path_2_fuel_consumption_data = "";

    double diesel_CO2_kgL = 2.7;
    double diesel_CO_kgL = 0.0178;
    double diesel_NOx_kgL = 0.0014;
    double diesel_SOx_kgL = 0.0042;
    double diesel_CH4_kgL = 0.0007;
    double diesel_PM_kgL = 0.0001;
};
```

Each attribute of `structCombustion` is described below in Table 4.2.

Table 4.2: `structCombustion` attribute descriptions

Attribute	Type	Default Value	Description
`fuel_mode`	FuelMode	LINEAR	Defines the approach to be used in modelling fuel consumption.
`cycle_charging_load_ratio`	double	0.85	Defines the proportion of `cap_kW` at which the object produces when running in cycle charging mode.
`ramp_rate_constraint_kWperhr`	double	-1 (sentinel)	Defines the ramp rate constraint. The default value of -1 is a sentinel value, which triggers a default ramp rate constraint. The default constraint is taken to be that rate which ramps production from zero up to rated capacity in 30 seconds.
`fuel_cost_L`	double	1.50	Defines the price of fuel per unit volume (expressed in litres). The units of currency are left undefined.
`nominal_fuel_escalation_rate`	double	0.05	Defines the nominal escalation rate in fuel cost (annual), used in computing real fuel costs. This rate is used instead of the given nominal inflation rate.
`linear_fuel_intercept_LkWh`	double	-1 (sentinel)	Defines the linear fuel intercept, expressed in litres per kiloWatt-hour produced, to be used in modelling fuel consumption. The default value of -1 is a sentinel value, which triggers a generic fuel intercept model within the constructor. This is only used if `fuel_mode` = LINEAR.
`linear_fuel_slope_LkWh`	double	-1 (sentinel)	Defines the linear fuel slope, expressed in litres per kiloWat-hour produced, to be used in modelling fuel consumption. The default value of -1 is a sentinel value, which triggers a generic fuel slope model within the constructor. This is only used if `fuel_mode` = LINEAR.
`path_2_fuel_consumption_data`	string	empty	Defines the (relative) path to the file containing fuel consumption data. This is only used if `fuel_mode` = LOOKUP.
`diesel_*_kgL`	double	varies; see structure definition	Defines the emissions rates for a variety of matter, in kilograms per litre consumed, for diesel fuel.

The complete set of `Combustion` attributes is described below in Table 4.3.

Table 4.3: `Combustion` attribute descriptions

Attribute	**Type**	**Default Value**	**Description**
`fuel_type`	FuelType	FUEL_DIESEL	Defines the type of fuel consumed by the object.
`struct_combustion`	structCombustion	as laid out in Table 4.2	The structure of inputs required by the `Combustion` class constructor.
`real_fuel_discount_rate_annual`	double	0	The real fuel discount rate (annual) to be applied to the object in computing real fuel costs (computed from the given nominal fuel escalation and discount rates).
`total_fuel_consumed_L`	double	0	Tracks the total fuel consumed, expressed in litres, of the object.
`total_CO2_emitted_kg`	double	0	Total mass of carbon dioxide, expressed in kilograms, emitted by the object.
`total_CO_emitted_kg`	double	0	Total mass of carbon monoxide, expressed in kilograms, emitted by the object.
`total_NOx_emitted_kg`	double	0	Total mass of nitrogen oxides, expressed in kilograms, emitted by the object.
`total_SOx_emitted_kg`	double	0	Total mass of sulfur oxides, expressed in kilograms, emitted by the object.
`total_CH4_emitted_kg`	double	0	Total mass of methane, expressed in kilograms, emitted by the object.
`total_PM_emitted_kg`	double	0	Total mass of particulate matter, expressed in kilograms, emitted by the object.
`fuel_interp_load_ratio_vec`	vector of doubles	empty	A vector of load ratios for the purpose of interpolating fuel consumption. This is only used if `fuel_mode` = LOOKUP.
`fuel_interp_consumption_vec_Lhr`	vector of doubles	empty	A vector of fuel consumption rates, expressed in litres per horus, for the purpose of interpolating fuel consumption. This is only used if `fuel_mode` = LOOKUP.
`fuel_vec_L`	vector of doubles	empty	A vector which records the fuel consumed, in litres, by the object over each time step.
`real_fuel_cost_vec`	vector of doubles	empty	A vector which records the real fuel costs incurred over each time step. The units of currency are left undefined.

Attribute	Type	Default Value	Description
CO2_vec_kg	vector of doubles	empty	A vector which records the mass of carbon dioxide emitted, in kilograms, by the object over each time step.
CO_vec_kg	vector of doubles	empty	A vector which records the mass of carbon monoxide emitted, in kilograms, by the object over each time step.
NOx_vec_kg	vector of doubles	empty	A vector which records the mass of nitrogen oxides emitted, in kilograms, by the object over each time step.
SOx_vec_kg	vector of doubles	empty	A vector which records the mass of sulfur oxides emitted, in kilograms, by the object over each time step.
CH4_vec_kg	vector of doubles	empty	A vector which records the mass of methane emitted, in kilograms, by the object over each time step.
PM_vec_kg	vector of doubles	empty	A vector which records the mass of particulate matter emitted, in kilograms, by the object over each time step.

The modelling of fuel consumption is handled in one of two ways, as enumerated by

```
enum FuelMode {
    LINEAR,
    LOOKUP
};
```

If `structCombustion :: fuel_mode` = LINEAR, then a generic linear fuel consumption model is employed. If `structCombustion :: fuel_mode` = LOOKUP, then fuel consumption is interpolated (linearly) from the given fuel consumption data. See the source file for more details.

Finally, for the LOOKUP case, note that the fuel consumption data must adhere to the units (i.e. litres per hour) and format expected by PGMcpp in order for it to be readable. For an example of the expected format, see

```
data/input/test/diesel_fuel_curve.csv
```

4.3 The `Diesel` Class

Header: `header/assets/dispatchable/combustion/Diesel.h`
Source: `source/assets/dispatchable/combustion/Diesel.cpp`

The `Diesel` class implements the modelling of a diesel generator. Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Diesel` class constructor. This structure is defined as follows

```
struct structDiesel {
    double minimum_load_ratio = 0.2;
    double minimum_runtime_hrs = 5;
};
```

Each attribute of `structDiesel` is described below in Table 4.4.

Table 4.4: `structDiesel` attribute descriptions

Attribute	Type	Default Value	Description
`minimum_load_ratio`	double	0.2	Defines the minimum load ratio (operating constraint) for the object.
`minimum_runtime_hrs`	double	5	Defines the minimum run time (operating constraint), expressed in hours, for the object.

The complete set of `Diesel` attributes is described below in Table 4.5.

Table 4.5: `Diesel` attribute descriptions

Attribute	Type	Default Value	Description
`struct_diesel`	structDiesel	as laid out in Table 4.4	The structure of inputs required by the `Diesel` class constructor.
`time_since_last_start_hrs`	double	0	Tracks the time elapsed, expressed in hours, since the object was last started.

5

The Storage Class Hierarchy

The `Storage` class hierarchy is where the modelling of energy storage assets is implemented. The class hierarchy is organized as follows

```
Storage
    <-- BatteryStorage
        <-- LiIon
```

That is, the `Storage` class is parent to the `BatteryStorage` class, which in turn is parent to the `LiIon` class. This hierarchy will be expanded and updated as new `Storage` assets are added to PGMcpp.

All attributes of the `Storage` objects that make up a `Model` instance will be written to the disk upon invoking `Model :: writeResults( )`. However, should you need access to attribute values within your program, you can always get a pointer to the n^{th} `Storage` that was added to the `Model` by way of

```
Storage* storage_ptr = model.storage_ptr_vec[n];
```

5.1 The Storage Class

Header: `header/assets/storage/Storage.h`
Source: `source/assets/storage/Storage.cpp`

The `Storage` class is the root of its class hierarchy. As such, it is intended to define the attributes and methods common to all members of the hierarchy. Interacting directly with its methods is not intended *under normal execution*; please refer to the header and source files for more information on these methods. That said, this class is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `Storage` class constructor. This structure is defined as follows

```
struct structStorage {
    bool is_sunk = false;
    bool print_flag = false;
```

```
    bool test_flag = false;

    double cap_kW = 100;
    double cap_kWh = 1000;  // this is "nominal capacity", and is static

    double nominal_inflation_rate_annual = 0.02;
    double nominal_discount_rate_annual = 0.04;

    double capital_cost = -1;
    double op_maint_cost_per_kWh = -1;
};
```

Each attribute of `structStorage` is described below in Table 5.1 (largely the same as `structNondispatchable`)

Table 5.1: `structStorage` attribute descriptions

Attribute	**Type**	**Default Value**	**Description**
`is_sunk`	boolean	false	Defines whether or not the object is modelled as a sunk cost.
`print_flag`	boolean	false	Controls whether or not running prints are done.
`test_flag`	boolean	false	Toggles testing printouts.
`cap_kW`	double	100	Defines the rated power capacity, expressed in kiloWatts, of the object.
`cap_kWh`	double	1000	Defines the rated energy capacity, expressed in kiloWatt-hours, of the object. This attribute is static, and does not vary with object degradation.
`nominal_inflation_rate_annual`	double	0.02	Defines the nominal inflation rate (annual), used in computing economic values for the object.
`nominal_discount_rate_annual`	double	0.04	Defines the nominal discount rate (annual), used in computing economic values for the object.
`capital_cost`	double	-1 (sentinel)	Defines the capital cost of the object. The default value of -1 is a sentinel value, which triggers a generic capital cost model within the constructor.
`op_maint_cost_per_kWh`	double	-1 (sentinel)	Defines the operation and maintenance costs, per kiloWatt-hour produced, of the object. The default value of -1 is a sentinel value, which triggers a generic operation and maintenance cost model within the constructor.

Additionally, the following enumeration of the `Storage` class hierarchy is defined

```
enum StorageType {
    LIION
};
```

The complete set of `Storage` attributes is described below in Table 5.2.

Table 5.2: `Storage` attribute descriptions

Attribute	Type	Default Value	Description
`storage_type`	StorageType	LIION	Defines the type `Storage` type of the object.
`struct_storage`	structStorage	as laid out in Table 5.1	The structure of inputs required by the `Storage` class constructor.
`n_timesteps`	integer	0	The number of time steps over which the model will run (same as the corresponding `Model` attribute).
`n_replacements`	integer	0	Tracks the number of times the object has been replaced.
`cap_kWh`	double	0	Defines the energy capacity, expressed in kiloWatt-hours, of the object. This attribute is dynamic, and does vary with object degradation.
`charge_kWh`	double	0	Tracks the charge (i.e., energy content), expressed in kiloWatt-hours, of the object.
`min_charge_kWh`	double	0	Defines the minimum charge (operating constraint), expressed in kiloWatt-hours, of the object.
`max_charge_kWh`	double	1000	Defines the maximum charge (operating constraint), expressed in kiloWatt-hours, of the object.
`project_life_yrs`	double	0	The project life, in years, to be modelled.
`total_throughput_kWh`	double	0	Tracks the total energy, expressed in kiloWatt-hours, that has been transported through the object.
`acceptable_kW`	double	0	Holds the charging power, expressed in kiloWatts, that can be accepted by the object at a particular point in time. Is used for memoization.
`charging_kW`	double	0	Holds the charging power, expressed in kiloWatts, that is being accepted by the object at a particular point in time. Is used for memoization.
`real_discount_rate_annual`	double	0	The real discount rate (annual) to be applied to the object (computed from the given nominal inflation and discount rates).
`net_present_cost`	double	0	The net present cost of the object, as computed following a model run. The units of currency are left undefined.

Attribute	Type	Default Value	Description
`levellized_cost_of_energy_per_kWh`	double	0	The levellized cost of energy (per kiloWatt-hour throughput), as computed following a model run. The units of currency are left undefined.
`storage_type_str`	string	empty	A string corresponding to the `storage_type` attribute.
`replaced_vec`	vector of booleans	empty	A vector which records replacements of the object.
`charge_vec_kWh`	vector of doubles	empty	A vector which records the charge (i.e., energy content), expressed in kiloWatt-hours, of the object at each point in time.
`charging_vec_kW`	vector of doubles	empty	A vector which records the charging power (if any), expressed in kiloWatts, that is accepted by the object at each point in time.
`discharging_vec_kW`	vector of doubles	empty	A vector which records the discharging power (if any), expressed in kiloWatts, that is provided by the object at each point in time.
`real_capital_cost_vec`	vector of doubles	empty	A vector which records the real capital costs incurred over each time step. The units of currency are left undefined.
`real_opt_maint_cost_vec`	vector of doubles	empty	A vector which records the real operation and maintenance costs incurred over each time step. The units of currency are left undefined.
`ptr_2_dt_vec_hr`	pointer to vector of doubles	NULL	A pointer to the `dt_vec_hr` attribute of the `Model`.
`ptr_2_time_vec_hr`	pointer to vector of doubles	NULL	A pointer to the `time_vec_hr` attribute of the `Model`.

5.2 The `BatteryStorage` Class

Header: `header/assets/storage/batterystorage/BatteryStorage.h`
Source: `source/assets/storage/batterystorage/BatteryStorage.cpp`

The `BatteryStorage` class is the root of its branch of the class hierarchy. As such, it is intended to define the attributes and methods common to all members of the branch. Interacting directly with its methods is not intended *under normal execution*; please refer to the header and source files for more information on these methods. That said, this class is paired with an input structure which provides the user with the complete bundle of input parameters expected by the `BatteryStorage` class constructor. This structure is defined as follows

```
struct structBatteryStorage {
    double init_SOC = 0.5;  // SOC = state of charge
    double min_SOC = 0.4;
    double max_SOC = 0.9;

    double hysteresis_SOC = 0.8;
    double reserve_SOC = 0.2;

    double charge_efficiency = 0.9;
    double discharge_efficiency = 0.9;
};
```

Each attribute of `structBatteryStorage` is described below in Table 5.3.

Table 5.3: `structBatteryStorage` attribute descriptions

Attribute	Type	Default Value	Description
`init_SOC`	double	0.5	Defines the initial state of charge (SOC) of the object. This is defined as `Storage :: charge_kWh` / `Storage :: structStorage :: cap_kWh`. This attribute will alter the initial value of `Storage :: charge_kWh`.
`min_SOC`	double	0.4	Defines the minimum SOC of the object. This attribute will alter the initial value of `Storage :: min_charge_kWh`.
`max_SOC`	double	0.9	Defines the maximum SOC of the object. This attribute will alter the initial value of `Storage :: max_charge_kWh`.
`hysteresis_SOC`	double	0.8	Defines the SOC that must be attained after reaching the minimum SOC before the object can be discharged again.
`reserve_SOC`	double	0.2	Defines a minimum SOC that comes into effect whenever there is otherwise insufficient production and stored energy to meet demand.
`charge_efficiency`	double	0.9	Defines how efficiently the object transforms a charging (i.e., input) power into an increase in charge (i.e., energy content).
`discharge_efficiency`	double	0.9	Defines how efficiently the object transforms a decrease in charge (i.e., energy content) into a discharging (i.e., output) power.

The sole attribute of the `BatteryStorage` class is the provided `structBatteryStroage` instance.

5.3 The LiIon Class

Header: header/assets/storage/batterystorage/LiIon.h
Source: source/assets/storage/batterystorage/LiIon.cpp

The LiIon class implements the modelling of a lithium ion (Li-ion) battery energy storage system. Like every class in PGMcpp, it is paired with an input structure which provides the user with the complete bundle of input parameters expected by the LiIon class constructor. This structure is defined as follows

```
struct structLiIon {
    double replace_SOH = 0.8;

    double degr_alpha = 10;              // [ ]
    double degr_beta = 1.1;              // [ ]
    double degr_B_hat_cal_0 = 5.222e6;   // [1/sqrt(hr)]
    double degr_r_cal = 0.350;           // [ ]
    double degr_Ea_cal_0 = 5.279e4;      // [J/mol]
    double degr_a_cal = 108.5;           // [J/mol]
    double degr_s_cal = 1.895;           // [ ]
    double gas_constant_JmolK = 8.31446;
    double temperature_K = 273 + 20;
};
```

The replace_SOH attribute defines the state of health (SOH) at which the object is considered "dead" (this is the trigger value for replacing the object within a Model run). The remaining attributes have to do with how the degradation of the object is modelled. For more details, see

docs/refs/battery_degradation.pdf

The complete set of LiIon attributes is described below in Table 5.4.

Table 5.4: LiIon attribute descriptions

Attribute	Type	Default Value	Description
struct_liion	structLiIon	as defined above	The structure of inputs required by the LiIon class constructor.
SOH	double	1	Tracks the state of health (SOH) of the object. This is defined as Storage :: cap_kWh / Storage :: structStorage :: cap_kWh.
SOH_vec	vector of doubles	empty	A vector which records the SOH of the object at each point in time.

6

Dispatch Control

The control strategy applied to the components of a microgrid can have a significant impact on the economic and environmental performance of the grid design. In light of this, PGMcpp is designed to allow for the modelling of various control strategies. Within the `Model` class, the following enumeration of dispatch (control) modes is defined

```
enum DispatchMode {
    LOAD_FOLLOWING_IN_ORDER,
    CYCLE_CHARGING_IN_ORDER
}
```

and this control enumeration will be expanded and updated as new dispatch control strategies are added to PGMcpp. Furthermore, given the importance of control, all associated code is stored in `source/control/`.

6.1 Net Load

Before summarizing the handling (i.e., control) of dispatchable assets, a word on the handling of nondispatchable assets (i.e., the renewable assets) is warranted. Whenever a call is made to `Model :: run()`, the following sequence of actions is taken

1. The net load over the entire modelled project life is computed. (This is where the nondispatchable assets are handled.)
2. The control of the dispatchable assets is then handled for each point in time. (This is where `LOAD_FOLLOWING_IN_ORDER` etc. is applied.)
3. Fuel consumption and emissions are then computed.
4. Model economics are then computed.

The net load, at any point in time, is here defined as the electrical load minus the sum of all renewable production. That is

$$\widehat{L}_i = L_i - \sum_{j=1}^{N} P_{i,\text{asset}j} \tag{6.1}$$

Where L_i is the i^{th} load value, $\widehat{L}_i$ is the corresponding net load value, and $P_{i,\text{asset}j}$ is the production from the j^{th} nondispatchable asset at the i^{th} point in time. Observe that, under this definition, a negative (or zero) net load indicates a surplus of nondispatchable production, and a positive net load indicates a deficit of nondispatchable production.

Finally, note that the production, dispatch, and curtailment of all nondispatchable assets is modelled and recorded, for every point in time, during the handling of action (1) of `Model :: run()`. As such, the nondispatchable assets are always used first at every point in time, regardless of which dispatch control strategy is applied in action (2) of `Model :: run()`.

6.2 Load Following, In Order

Source: `source/control/LOAD_FOLLOWING_IN_ORDER.cpp`

Load following, in order is perhaps the simplest control strategy, and the intent is to meet the load at every point in time using a minimum of production and dispatch. At each point in time, depending on the sign of the net load, dispatch control enters either a charging mode or a discharging mode.

When in charging mode (i.e., $\widehat{L}_i \leq 0$), the following sequence of actions are taken

1. Zero production is requested from all dispatchable assets (since they do not need to be producing anything in this case to meet the load). The assets may or may not be able to comply depending on their operating constraints.

2. All storage assets are charged using any available overproduction. In attempting to charge the storage assets, overproduction from combustion assets is considered first, then non-combustion assets, then nondispatchable assets.

When in discharging mode (i.e., $\widehat{L}_i > 0$), the following sequence of actions are taken

1. The set of all storage assets is partitioned into depleted and non-depleted.

2. All non-depleted storage assets are then used to satisfy as much of the load as possible.

3. If there is any load left unsatisfied, then all non-combustion assets are used to satisfy as much of the load as possible.

4. If there is any load left unsatisfied, then all combustion assets are used to satisfy as much of the load as possible.

5. All depleted storage assets are then charged using any available overproduction. In attempting to charge the storage assets, overproduction from combustion assets is considered first, then non-combustion assets, then nondispatchable assets.

Finally, the "in order" part of the control algorithm is important. As detailed above, different classes of asset take priority over others when either producing to meet load or sending overproduction to storage. In every case, sending overproduction to storage is always done in the order combustion, then non-combustion, then nondispatchable. Conversely, when in discharging mode, production/dispatch is always storage first, then non-combustion, then combustion. Furthermore, order within each class is also followed, with this order being defined simply by the order in which objects are added to the `Model`; understanding this offers the user control over asset priority within any given class.

6.3 Cycle Charging, In Order

Source: `source/control/CYCLE_CHARGING_IN_ORDER.cpp`

Cycle charging, in order control is largely similar to load following, in order control except for one key difference; whenever there are depleted storage assets, the combustion assets are run at at least some proportion of their capacity in order to charge storage assets more quickly while consuming fuel more efficiently. This "some proportion" is defined by the `cycle_charging_load_ratio` attribute of `structCombustion`. The simple logic here is, if the combustion asset would run at less than this proportion in load following mode, then it will run at this proportion in cycle charging mode (with the exception of zero; if the asset can shut down, then it will still do so). Conversely, if the combustion asset would run at greater than this proportion in load following mode, then it will also do so in cycle charging mode.

7

An Example PGMcpp Project

7.1 First Time Setup

7.1.1 Linux

Set up should be fairly straightforward on any Linux distribution.[1] Once you have downloaded the PGMcpp files, extract them into a directory of your choice and then issue the command

```
$  make all
```

This will build PGMcpp and then run the test suite (see `test/` for more details). Once you see

```
************************
**  ALL TESTS PASSED  **
************************
```

you are all set up and ready to go.

7.1.2 Windows

In order to use PGMcpp on Windows,[2] it is recommended that the user install MSYS2, which is a collection of tools and libraries that provide an easy-to-use environment for building, installing, and running native Windows software. For install instructions, see `https://www.msys2.org/` (be sure to follow *all* steps).

Once you have worked through the MSYS2 install instructions, there are two more package installations that need to be performed (from within an MSYS2 terminal) in order for PGMcpp to compile; namely

```
$  pacman -Syu
$  pacman -S base-devel gcc vim cmake
```

[1] PGMcpp was developed and tested in Linux Mint 20.2

[2] PGMcpp was tested in Windows 11 Home (10.0.22621 Build 22621).

Once the additional packages have been installed, extract the PGMcpp files to somewhere within `C:\msys64\home` (assuming you installed MSYS2 to the default location). Then, launch MSYS2 and navigate to the PGMcpp folder by way of (for example)

```
$  cd /home/PGM_cpp_v2-1_dev/
```

Finally, issue the command

```
$  make all
```

This will build PGMcpp and then run the test suite (see `test/` for more details). Once you see

```
************************
**  ALL TESTS PASSED  **
************************
```

you are all set up and ready to go.

7.2 Example Project Code

Source: `projects/example_project.cpp`

The example project considers a one year project life at a time resolution of one hour; this is defined by the given electrical load time series data. The project code begins with a minimal instantiation of a `Model` object.

```
structModel model_inputs;
model_inputs.path_2_load_data =
    "data/input/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

Model model(model_inputs);
```

Once the `Model` has been instantiated, renewable resource data for solar, wind, tidal, and wave are loaded.

```
//  1. solar
int solar_resource_key = 1;

model.add1dRenewableResource(
    "solar",
    "data/input/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv",
    solar_resource_key
);

//  2. wind
int wind_resource_key = 2;

model.add1dRenewableResource(
    "wind",
    "data/input/test/wind_speed_peak-25ms_1yr_dt-1hr.csv",
    wind_resource_key
);

//  3. tidal
int tidal_resource_key = 3;

model.add1dRenewableResource(
    "tidal",
    "data/input/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv",
    tidal_resource_key
);

//  4. wave
int wave_resource_key = 4;

model.add2dRenewableResource(
    "wave",
    "data/input/test/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv",
    wave_resource_key
);
```

Then, renewable production assets are added to the `Model`, one asset per corresponding resource. Again, minimal instantiations are used.

```
//  1. solar
structNondispatchable nondisp_inputs;
nondisp_inputs.cap_kW = 150;

structSolar solar_inputs;
solar_inputs.resource_key = solar_resource_key;

model.addSolar(nondisp_inputs, solar_inputs);

//  2. wind
nondisp_inputs.cap_kW = 100;

structWind wind_inputs;
wind_inputs.resource_key = wind_resource_key;

model.addWind(nondisp_inputs, wind_inputs);

//  3. tidal
nondisp_inputs.cap_kW = 50;

structTidal tidal_inputs;
tidal_inputs.resource_key = tidal_resource_key;

model.addTidal(nondisp_inputs, tidal_inputs);

//  4. wave
nondisp_inputs.cap_kW = 50;

structWave wave_inputs;
wave_inputs.resource_key = wave_resource_key;

model.addWave(nondisp_inputs, wave_inputs);
```

Then, a diesel generator is added to the `Model`. Again, minimal.

```
structDispatchable disp_inputs;
disp_inputs.cap_kW = 1.2 * 500;

structCombustion combustion_inputs;

structDiesel diesel_inputs;

model.addDiesel(disp_inputs, combustion_inputs, diesel_inputs);
```

Then, a lithium-ion battery energy storage system is added to the `Model`. Again, minimal.

```
structStorage storage_inputs;
storage_inputs.cap_kW = 150;
storage_inputs.cap_kWh = 1000;

structBatteryStorage batt_storage_inputs;

structLiIon liion_inputs;

model.addLiIon(storage_inputs, batt_storage_inputs, liion_inputs);
```

Finally, the `Model` is run and the results are written to the disk.

```
//  run model
model.run();


//  write modelling results to disk
model.writeResults("example_project");
```

The user is, of course, encouraged to use the provided example project as a template for all future projects.

7.3 Compiling and Executing

To compile and execute the example project (or in fact, any project), simply issue the command

```
$  make project
```

For compiling and executing some other project, say `my_project.cpp` for example, one need only make a single change to the provided makefile. Namely, the `PROJECT_NAME` variable need only be changed from `example_project` to `my_project`. Note that this works on the assumption that `my_project.cpp` exists and is located within `projects/`.

7.4 Expected Output

Upon successfully compiling and executing the example project, you should see

```
Results successfully written to data/output/example_project/
```

Within `data/output/example_project/`, you should find the following file structure

```
data/output/example_project/
    Combustion/
    Model/
    Nondispatchable/
    Storage/
```

7.4.1 Combustion/

The Combustion/ folder contains the results for each Combustion object that was added to the Model prior to being run. In this case, there is only one set of results; namely 600kW_DIESEL_0, the results for the 600 kW diesel generator. Within this results folder, you will find two files

1. 600kW_DIESEL_0_results.csv: a complete time series of the diesel generator dynamics over the course of the modelled project life; and,
2. 600kW_DIESEL_0_summary.txt: a summary of the aggregate model results for the diesel generator, including key economic and environmental metrics.

7.4.2 Model/

The Model/ folder contains the model-level results from the model run. Within this folder, you will find three files

1. Model_dispatch_results.csv: a complete time series of the load in parallel with the dispatch time series for every object added to the Model prior to being run;
2. Model_load_results.csv: a complete time series of the load, net load, and remaining load; and,
3. Model_summary.txt: a summary of the aggregate model-level results, including key economic and environmental metrics.

7.4.3 Nondispatchable/

The Nondispatchable/ folder contains the results for each Nondispatchable (i.e., renewable) object that was added to the Model prior to being run. In this case, there are four sets of results; namely

1. 50kW_TIDAL_2: the results for the 50 kW tidal turbine;
2. 50kW_WAVE_3: the results for the 50 kW wave energy converter;
3. 100kW_WIND_1: the results for the 100 kW wind turbine; and,
4. 150kW_SOLAR_0: the results for the 150 kW solar photovoltaic array.

Note that the trailing integer in the folder names is simply the index of the object within Model :: nondisp_ptr_vec; that is, it simply indicates the order in which the objects were added to the Model.

Within each object results folder, you will find two files

1. *_results.csv: a complete time series of the object dynamics over the course of the modelled project life; and,
2. *_summary.txt: a summary of the aggregate model results for the object, including key economic and environmental metrics.

7.4.4 Storage/

The Storage/ folder contains the results for each Storage object that was added to the Model prior to being run. In this case, there is only one set of results; namely 150kW_1000kWh_LIION_0, the results for the 150 kW, 1000 kWh lithium-ion battery energy storage system. Within this results folder, you will find two files

1. 150kW_1000kWh_LIION_0_results.csv: a complete time series of the lithium-ion battery energy storage system dynamics over the course of the modelled project life; and,

2. 150kW_1000kWh_LIION_0_summary.txt: a summary of aggregate model results for the lithium-ion battery energy storage system, including key economic and environmental metrics.

7.5 More Example Code

More examples of how to interact with a Model object can also be found in test/test_Model.cpp.