PGMcpp: PRIMED Grid Modelling (in C++)

Generated by Doxygen 1.9.1

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

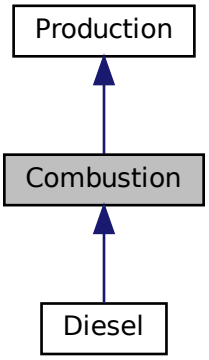
Class Documentation

4.1 Combustion Class Reference

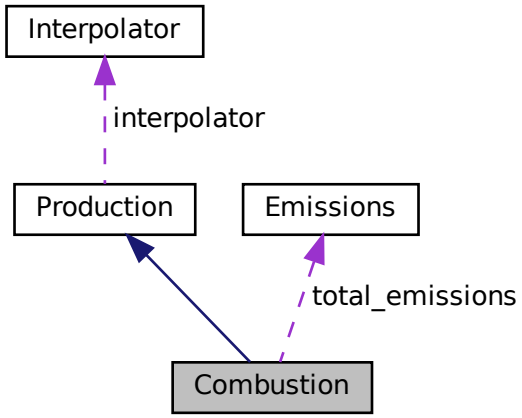
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

`#include <Combustion.h>`

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

- Combustion (void)

  Constructor (dummy) for the Combustion class.
- Combustion (int, double, CombustionInputs, std::vector< double > *)

  Constructor (intended) for the Combustion class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeFuelAndEmissions (void)

  Helper method to compute the total fuel consumption and emissions over the Model run.
- void computeEconomics (std::vector< double > *)

  Helper method to compute key economic metrics for the Model run.
- virtual double requestProductionkW (int, double, double)
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- double getFuelConsumptionL (double, double)

  Method which takes in production and returns volume of fuel burned over the given interval of time.
- Emissions getEmissionskg (double)

  Method which takes in volume of fuel consumed and returns mass spectrum of resulting emissions.
- void writeResults (std::string, std::vector< double > *, int, int=-1)

  Method which writes Combustion results to an output directory.
- virtual ~Combustion (void)

  Destructor for the Combustion class.

Public Attributes

- CombustionType type

  The type (CombustionType) of the asset.
- FuelMode fuel_mode

  The fuel mode to use in modelling fuel consumption.
- Emissions total_emissions

  An Emissions structure for holding total emissions [kg].
- double fuel_cost_L

  The cost of fuel [1/L] (undefined currency).
- double nominal_fuel_escalation_annual

  The nominal, annual fuel escalation rate to use in computing model economics.
- double real_fuel_escalation_annual

  The real, annual fuel escalation rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double linear_fuel_slope_LkWh

  The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double linear_fuel_intercept_LkWh

  The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double CO2_emissions_intensity_kgL

  Carbon dioxide (CO2) emissions intensity [kg/L].
- double CO_emissions_intensity_kgL

  Carbon monoxide (CO) emissions intensity [kg/L].
- double NOx_emissions_intensity_kgL

  Nitrogen oxide (NOx) emissions intensity [kg/L].
- double SOx_emissions_intensity_kgL

  Sulfur oxide (SOx) emissions intensity [kg/L].
- double CH4_emissions_intensity_kgL

  Methane (CH4) emissions intensity [kg/L].
- double PM_emissions_intensity_kgL

  Particulate Matter (PM) emissions intensity [kg/L].
- double total_fuel_consumed_L

  The total fuel consumed [L] over a model run.
- std::string fuel_mode_str

  A string describing the fuel mode of the asset.
- std::vector< double > fuel_consumption_vec_L

  A vector of fuel consumed [L] over each modelling time step.
- std::vector< double > fuel_cost_vec

  A vector of fuel costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > CO2_emissions_vec_kg

  A vector of carbon dioxide (CO2) emitted [kg] over each modelling time step.
- std::vector< double > CO_emissions_vec_kg

  A vector of carbon monoxide (CO) emitted [kg] over each modelling time step.
- std::vector< double > NOx_emissions_vec_kg

  A vector of nitrogen oxide (NOx) emitted [kg] over each modelling time step.
- std::vector< double > SOx_emissions_vec_kg

  A vector of sulfur oxide (SOx) emitted [kg] over each modelling time step.
- std::vector< double > CH4_emissions_vec_kg

  A vector of methane (CH4) emitted [kg] over each modelling time step.
- std::vector< double > PM_emissions_vec_kg

  A vector of particulate matter (PM) emitted [kg] over each modelling time step.

Private Member Functions

- void __checkInputs (CombustionInputs)

  Helper method to check inputs to the Combustion constructor.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

4.1.1 Detailed Description

The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Combustion() [1/2]

```
Combustion::Combustion (
            void  )
```

Constructor (dummy) for the Combustion class.

```
77 {
78     return;
79 }   /* Combustion() */
```

4.1.2.2 Combustion() [2/2]

```
Combustion::Combustion (
            int n_points,
            double n_years,
            CombustionInputs combustion_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Combustion class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
combustion_inputs	A structure of Combustion constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
111   :
112 Production(
113     n_points,
114     n_years,
115     combustion_inputs.production_inputs,
116     time_vec_hrs_ptr
117 )
```

```
118 {
119     //  1. check inputs
120     this->__checkInputs(combustion_inputs);
121 
122     //  2. set attributes
123     this->fuel_mode = combustion_inputs.fuel_mode;
124 
125     switch (this->fuel_mode) {
126         case (FuelMode :: FUEL_MODE_LINEAR): {
127             this->fuel_mode_str = "FUEL_MODE_LINEAR";
128 
129             break;
130         }
131 
132         case (FuelMode :: FUEL_MODE_LOOKUP): {
133             this->fuel_mode_str = "FUEL_MODE_LOOKUP";
134 
135             this->interpolator.addData1D(
136                 0,
137                 combustion_inputs.path_2_fuel_interp_data
138             );
139 
140             break;
141         }
142 
143         default: {
144             std::string error_str = "ERROR:  Combustion():  ";
145             error_str += "fuel mode ";
146             error_str += std::to_string(this->fuel_mode);
147             error_str += " not recognized";
148 
149             #ifdef _WIN32
150                 std::cout « error_str « std::endl;
151             #endif
152 
153             throw std::runtime_error(error_str);
154 
155             break;
156         }
157     }
158 
159     this->fuel_cost_L = 0;
160     this->nominal_fuel_escalation_annual =
161         combustion_inputs.nominal_fuel_escalation_annual;
162 
163     this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
164         combustion_inputs.nominal_fuel_escalation_annual,
165         combustion_inputs.production_inputs.nominal_discount_annual
166     );
167 
168     this->linear_fuel_slope_LkWh = 0;
169     this->linear_fuel_intercept_LkWh = 0;
170 
171     this->CO2_emissions_intensity_kgL = 0;
172     this->CO_emissions_intensity_kgL = 0;
173     this->NOx_emissions_intensity_kgL = 0;
174     this->SOx_emissions_intensity_kgL = 0;
175     this->CH4_emissions_intensity_kgL = 0;
176     this->PM_emissions_intensity_kgL = 0;
177 
178     this->total_fuel_consumed_L = 0;
179 
180     this->fuel_consumption_vec_L.resize(this->n_points, 0);
181     this->fuel_cost_vec.resize(this->n_points, 0);
182 
183     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
184     this->CO_emissions_vec_kg.resize(this->n_points, 0);
185     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
186     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
187     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
188     this->PM_emissions_vec_kg.resize(this->n_points, 0);
189 
190     //  3. construction print
191     if (this->print_flag) {
192         std::cout « "Combustion object constructed at " « this « std::endl;
193     }
194 
195     return;
196 }   /* Combustion() */
```

4.1.2.3 ~Combustion()

```
Combustion::~Combustion (
            void  )  [virtual]
```

Destructor for the Combustion class.

```
534 {
535     //  1. destruction print
536     if (this->print_flag) {
537         std::cout « "Combustion object at " « this « " destroyed" « std::endl;
538     }
539
540     return;
541 }   /* ~Combustion() */
```

4.1.3 Member Function Documentation

4.1.3.1 __checkInputs()

```
void Combustion::__checkInputs (
            CombustionInputs combustion_inputs )  [private]
```

Helper method to check inputs to the Combustion constructor.

Parameters

combustion_inputs	A structure of Combustion constructor inputs.

```
40 {
41     // 1. if FUEL_MODE_LOOKUP, check that path is given
42     if (
43         combustion_inputs.fuel_mode == FuelMode :: FUEL_MODE_LOOKUP and
44         combustion_inputs.path_2_fuel_interp_data.empty()
45     ) {
46         std::string error_str = "ERROR:  Combustion()  fuel mode was set to ";
47         error_str += "FuelMode::FUEL_MODE_LOOKUP, but no path to fuel interpolation ";
48         error_str += "data was given";
49
50         #ifdef _WIN32
51             std::cout « error_str « std::endl;
52         #endif
53
54         throw std::invalid_argument(error_str);
55     }
56
57     return;
58 }   /* __checkInputs() */
```

4.1.3.2 __writeSummary()

```
virtual void Combustion::__writeSummary (
            std::string  )  [inline], [private], [virtual]
```

Reimplemented in Diesel.

```
105 {return;}
```

4.1.3.3 __writeTimeSeries()

```
virtual void Combustion::__writeTimeSeries (
            std::string ,
            std::vector< double > * ,
            int  = -1 )  [inline], [private], [virtual]
```

Reimplemented in Diesel.

```
110          {return;}
```

4.1.3.4 commit()

```
double Combustion::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Production.

Reimplemented in Diesel.

```
326 {
327     //  1. invoke base class method
328     load_kW = Production :: commit(
329         timestep,
330         dt_hrs,
331         production_kW,
332         load_kW
333     );
334 
335 
336     if (this->is_running) {
337         //  2. compute and record fuel consumption
338         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
339         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
340 
341         //  3. compute and record emissions
342         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
343         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
344         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
345         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
346         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
347         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
348         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
349 
350         //  4. incur fuel costs
```

```
351         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
352     }
353
354     return load_kW;
355 }   /* commit() */
```

4.1.3.5 computeEconomics()

```
void Combustion::computeEconomics (
            std::vector< double > * time_vec_hrs_ptr )  [virtual]
```

Helper method to compute key economic metrics for the Model run.

Ref: HOMER [2023b]

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
270 {
271     //  1. account for fuel costs in net present cost
272     double t_hrs = 0;
273     double real_fuel_escalation_scalar = 0;
274
275     for (int i = 0; i < this->n_points; i++) {
276         t_hrs = time_vec_hrs_ptr->at(i);
277
278         real_fuel_escalation_scalar = 1.0 / pow(
279             1 + this->real_fuel_escalation_annual,
280             t_hrs / 8760
281         );
282
283         this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
284     }
285
286     //  2. invoke base class method
287     Production :: computeEconomics(time_vec_hrs_ptr);
288
289     return;
290 }   /* computeEconomics() */
```

4.1.3.6 computeFuelAndEmissions()

```
void Combustion::computeFuelAndEmissions (
            void  )
```

Helper method to compute the total fuel consumption and emissions over the Model run.

```
238 {
239     for (int i = 0; i < n_points; i++) {
240         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
241
242         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
243         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
244         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
245         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
246         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
247         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
248     }
249
250     return;
251 }   /* computeFuelAndEmissions() */
```

4.1.3.7 getEmissionskg()

```
Emissions Combustion::getEmissionskg (
            double fuel_consumed_L )
```

Method which takes in volume of fuel consumed and returns mass spectrum of resulting emissions.

Parameters

fuel_consumed_L	The volume of fuel consumed [L].

Returns

A structure containing the mass spectrum of resulting emissions.

```
434                                                         {
435     Emissions emissions;
436
437     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
438     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
439     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
440     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
441     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
442     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
443
444     return emissions;
445 }   /* getEmissionskg() */
```

4.1.3.8 getFuelConsumptionL()

```
double Combustion::getFuelConsumptionL (
            double dt_hrs,
            double production_kW )
```

Method which takes in production and returns volume of fuel burned over the given interval of time.

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.

Returns

The volume of fuel consumed [L].

```
377 {
378     double fuel_consumed_L = 0;
379
380     switch (this->fuel_mode) {
381         case (FuelMode :: FUEL_MODE_LINEAR): {
382             fuel_consumed_L = (
383                 this->linear_fuel_slope_LkWh * production_kW +
384                 this->linear_fuel_intercept_LkWh * this->capacity_kW
385             ) * dt_hrs;
386
387             break;
388         }
389
390         case (FuelMode :: FUEL_MODE_LOOKUP): {
```

```
391             double load_ratio = production_kW / this->capacity_kW;
392
393             fuel_consumed_L = this->interpolator.interp1D(0, load_ratio) * dt_hrs;
394
395             break;
396         }
397
398         default: {
399             std::string error_str = "ERROR:  Combustion::getFuelConsumptionL():  ";
400             error_str += "fuel mode ";
401             error_str += std::to_string(this->fuel_mode);
402             error_str += " not recognized";
403
404             #ifdef _WIN32
405                 std::cout << error_str << std::endl;
406             #endif
407
408             throw std::runtime_error(error_str);
409
410             break;
411         }
412     }
413
414     return fuel_consumed_L;
415 } /* getFuelConsumptionL() */
```

4.1.3.9 handleReplacement()

```
void Combustion::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Diesel.

```
214 {
215     //  1. reset attributes
216     //...
217
218     //  2. invoke base class method
219     Production :: handleReplacement(timestep);
220
221     return;
222 }   /* __handleReplacement() */
```

4.1.3.10 requestProductionkW()

```
virtual double Combustion::requestProductionkW (
            int ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Diesel.

```
156 {return 0;}
```

4.1.3.11 writeResults()

```
void Combustion::writeResults (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            int combustion_index,
            int max_lines = -1 )
```

Method which writes Combustion results to an output directory.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
combustion_index	An integer which corresponds to the index of the Combustion asset in the Model.
max_lines	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```
481 {
482     //  1. handle sentinel
483     if (max_lines < 0) {
484         max_lines = this->n_points;
485     }
486 
487     //  2. create subdirectories
488     write_path += "Production/";
489     if (not std::filesystem::is_directory(write_path)) {
490         std::filesystem::create_directory(write_path);
491     }
492 
493     write_path += "Combustion/";
494     if (not std::filesystem::is_directory(write_path)) {
495         std::filesystem::create_directory(write_path);
496     }
497 
498     write_path += this->type_str;
499     write_path += "_";
500     write_path += std::to_string(int(ceil(this->capacity_kW)));
501     write_path += "kW_idx";
502     write_path += std::to_string(combustion_index);
503     write_path += "/";
504     std::filesystem::create_directory(write_path);
505 
506     //  3. write summary
507     this->__writeSummary(write_path);
508 
509     //  4. write time series
510     if (max_lines > this->n_points) {
511         max_lines = this->n_points;
512     }
513 
514     if (max_lines > 0) {
515         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
516     }
517 
518     return;
519 }   /* writeResults() */
```

4.1.4 Member Data Documentation

4.1.4.1 CH4_emissions_intensity_kgL

```
double Combustion::CH4_emissions_intensity_kgL
```

Methane (CH4) emissions intensity [kg/L].

4.1.4.2 CH4_emissions_vec_kg

```
std::vector<double> Combustion::CH4_emissions_vec_kg
```

A vector of methane (CH4) emitted [kg] over each modelling time step.

4.1.4.3 CO2_emissions_intensity_kgL

```
double Combustion::CO2_emissions_intensity_kgL
```

Carbon dioxide (CO2) emissions intensity [kg/L].

4.1.4.4 CO2_emissions_vec_kg

```
std::vector<double> Combustion::CO2_emissions_vec_kg
```

A vector of carbon dioxide (CO2) emitted [kg] over each modelling time step.

4.1.4.5 CO_emissions_intensity_kgL

```
double Combustion::CO_emissions_intensity_kgL
```

Carbon monoxide (CO) emissions intensity [kg/L].

4.1.4.6 CO_emissions_vec_kg

```
std::vector<double> Combustion::CO_emissions_vec_kg
```

A vector of carbon monoxide (CO) emitted [kg] over each modelling time step.

4.1.4.7 fuel_consumption_vec_L

```
std::vector<double> Combustion::fuel_consumption_vec_L
```

A vector of fuel consumed [L] over each modelling time step.

4.1.4.8 fuel_cost_L

```
double Combustion::fuel_cost_L
```

The cost of fuel [1/L] (undefined currency).

4.1.4.9 fuel_cost_vec

```
std::vector<double> Combustion::fuel_cost_vec
```

A vector of fuel costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

4.1.4.10 fuel_mode

```
FuelMode Combustion::fuel_mode
```

The fuel mode to use in modelling fuel consumption.

4.1.4.11 fuel_mode_str

```
std::string Combustion::fuel_mode_str
```

A string describing the fuel mode of the asset.

4.1.4.12 linear_fuel_intercept_LkWh

```
double Combustion::linear_fuel_intercept_LkWh
```

The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.

4.1.4.13 linear_fuel_slope_LkWh

```
double Combustion::linear_fuel_slope_LkWh
```

The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.

4.1.4.14 nominal_fuel_escalation_annual

```
double Combustion::nominal_fuel_escalation_annual
```

The nominal, annual fuel escalation rate to use in computing model economics.

4.1.4.15 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

Nitrogen oxide (NOx) emissions intensity [kg/L].

4.1.4.16 NOx_emissions_vec_kg

```
std::vector<double> Combustion::NOx_emissions_vec_kg
```

A vector of nitrogen oxide (NOx) emitted [kg] over each modelling time step.

4.1.4.17 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

Particulate Matter (PM) emissions intensity [kg/L].

4.1.4.18 PM_emissions_vec_kg

```
std::vector<double> Combustion::PM_emissions_vec_kg
```

A vector of particulate matter (PM) emitted [kg] over each modelling time step.

4.1.4.19 real_fuel_escalation_annual

```
double Combustion::real_fuel_escalation_annual
```

The real, annual fuel escalation rate used in computing model economics. Is computed from the given nominal inflation and discount rates.

4.1.4.20 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

Sulfur oxide (SOx) emissions intensity [kg/L].

4.1.4.21 SOx_emissions_vec_kg

```
std::vector<double> Combustion::SOx_emissions_vec_kg
```

A vector of sulfur oxide (SOx) emitted [kg] over each modelling time step.

4.1.4.22 total_emissions

```
Emissions Combustion::total_emissions
```

An Emissions structure for holding total emissions [kg].

4.1.4.23 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

The total fuel consumed [L] over a model run.

4.1.4.24 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

The documentation for this class was generated from the following files:

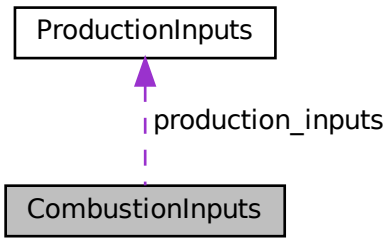
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

4.2 CombustionInputs Struct Reference

A structure which bundles the necessary inputs for the Combustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

`#include <Combustion.h>`

Collaboration diagram for CombustionInputs:



Public Attributes

- ProductionInputs production_inputs

  An encapsulated ProductionInputs instance.
- FuelMode fuel_mode = FuelMode :: FUEL_MODE_LINEAR

  The fuel mode to use in modelling fuel consumption.
- double nominal_fuel_escalation_annual = 0.05

  The nominal, annual fuel escalation rate to use in computing model economics.
- std::string path_2_fuel_interp_data = ""

  A path (either relative or absolute) to a set of fuel consumption data.

4.2.1 Detailed Description

A structure which bundles the necessary inputs for the Combustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

4.2.2 Member Data Documentation

4.2.2.1 fuel_mode

```
FuelMode CombustionInputs::fuel_mode = FuelMode ::  FUEL_MODE_LINEAR
```

The fuel mode to use in modelling fuel consumption.

4.2.2.2 nominal_fuel_escalation_annual

```
double CombustionInputs::nominal_fuel_escalation_annual = 0.05
```

The nominal, annual fuel escalation rate to use in computing model economics.

4.2.2.3 path_2_fuel_interp_data

```
std::string CombustionInputs::path_2_fuel_interp_data = ""
```

A path (either relative or absolute) to a set of fuel consumption data.

4.2.2.4 production_inputs

```
ProductionInputs CombustionInputs::production_inputs
```

An encapsulated ProductionInputs instance.

The documentation for this struct was generated from the following file:

- header/Production/Combustion/Combustion.h

4.3 Controller Class Reference

A class which contains a various dispatch control logic. Intended to serve as a component class of Model.

```
#include <Controller.h>
```

Public Member Functions

- Controller (void)

  Constructor for the Controller class.
- void setControlMode (ControlMode)
- void init (ElectricalLoad *, std::vector< Renewable * > *, Resources *, std::vector< Combustion * > *)

  Method to initialize the Controller component of the Model.
- void applyDispatchControl (ElectricalLoad *, Resources *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *, std::vector< Storage * > *)

  Method to apply dispatch control at every point in the modelling time series.
- void clear (void)

  Method to clear all attributes of the Controller object.
- ~Controller (void)

  Destructor for the Controller class.

Public Attributes

- ControlMode control_mode

  The ControlMode that is active in the Model.
- std::string control_string

  A string describing the active ControlMode.
- std::vector< double > net_load_vec_kW

  A vector of net load values [kW] at each point in the modelling time series. Net load is defined as load minus all available Renewable production.
- std::vector< double > missed_load_vec_kW

  A vector of missed load values [kW] at each point in the modelling time series.
- std::map< double, std::vector< bool > > combustion_map

  A map of all possible combustion states, for use in determining optimal dispatch.

Private Member Functions

- void __computeNetLoad (ElectricalLoad *, std::vector< Renewable * > *, Resources *)

  Helper method to compute and populate the net load vector.
- void __constructCombustionMap (std::vector< Combustion * > *)

  Helper method to construct a Combustion map, for use in determining.
- void __applyLoadFollowingControl_CHARGING (int, ElectricalLoad *, Resources *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *, std::vector< Storage * > *)

  Helper method to apply load following control action for given timestep of the Model run when net load <= 0;.
- void __applyLoadFollowingControl_DISCHARGING (int, ElectricalLoad *, Resources *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *, std::vector< Storage * > *)

  Helper method to apply load following control action for given timestep of the Model run when net load > 0;.
- void __applyCycleChargingControl_CHARGING (int, ElectricalLoad *, Resources *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *, std::vector< Storage * > *)

  Helper method to apply cycle charging control action for given timestep of the Model run when net load <= 0. Simply defaults to load following control.
- void __applyCycleChargingControl_DISCHARGING (int, ElectricalLoad *, Resources *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *, std::vector< Storage * > *)

  Helper method to apply cycle charging control action for given timestep of the Model run when net load > 0. Defaults to load following control if no depleted storage assets.
- void __handleStorageCharging (int, double, std::list< Storage * >, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *)

  Helper method to handle the charging of the given Storage assets.
- void __handleStorageCharging (int, double, std::vector< Storage * > *, std::vector< Combustion * > *, std::vector< Noncombustion * > *, std::vector< Renewable * > *)

  Helper method to handle the charging of the given Storage assets.
- double __getRenewableProduction (int, double, Renewable *, Resources *)

  Helper method to compute the production from the given Renewable asset at the given point in time.
- double __handleCombustionDispatch (int, double, double, std::vector< Combustion * > *, bool)

  bool is_cycle_charging)
- double __handleNoncombustionDispatch (int, double, double, std::vector< Noncombustion * > *, Resources *)
- double __handleStorageDischarging (int, double, double, std::list< Storage * >)

  Helper method to handle the discharging of the given Storage assets.

4.3.1 Detailed Description

A class which contains a various dispatch control logic. Intended to serve as a component class of Model.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Controller()

```
Controller::Controller (
            void  )
```

Constructor for the Controller class.

```
1248 {
1249     return;
1250 }   /* Controller() */
```

4.3.2.2 ~Controller()

```
Controller::~Controller (
            void  )
```

Destructor for the Controller class.

```
1494 {
1495     this->clear();
1496
1497     return;
1498 }   /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

```
void Controller::__applyCycleChargingControl_CHARGING (
            int timestep,
            ElectricalLoad * electrical_load_ptr,
            Resources * resources_ptr,
            std::vector< Combustion * > * combustion_ptr_vec_ptr,
            std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr,
            std::vector< Storage * > * storage_ptr_vec_ptr )  [private]
```

Helper method to apply cycle charging control action for given timestep of the Model run when net load <= 0. Simply defaults to load following control.

Parameters

timestep	The current time step of the Model run.
electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
storage_ptr_vec_ptr	A pointer to the Storage pointer vector of the Model.

```
450 {
451     //  1. default to load following
452     this->__applyLoadFollowingControl_CHARGING(
453         timestep,
454         electrical_load_ptr,
455         resources_ptr,
456         combustion_ptr_vec_ptr,
457         noncombustion_ptr_vec_ptr,
458         renewable_ptr_vec_ptr,
459         storage_ptr_vec_ptr
460     );
461
462     return;
463 }   /* __applyCycleChargingControl_CHARGING() */
```

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

```
void Controller::__applyCycleChargingControl_DISCHARGING (
              int timestep,
              ElectricalLoad * electrical_load_ptr,
              Resources * resources_ptr,
              std::vector< Combustion * > * combustion_ptr_vec_ptr,
              std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
              std::vector< Renewable * > * renewable_ptr_vec_ptr,
              std::vector< Storage * > * storage_ptr_vec_ptr )  [private]
```

Helper method to apply cycle charging control action for given timestep of the Model run when net load $>$ 0. Defaults to load following control if no depleted storage assets.

Parameters

timestep	The current time step of the Model run.
electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
storage_ptr_vec_ptr	A pointer to the Storage pointer vector of the Model.

curtailment

```
511 {
512     //  1. get dt_hrs, net load
513     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
514     double net_load_kW = this->net_load_vec_kW[timestep];
515
516     //  2. partition Storage assets into depleted and non-depleted
517     std::list<Storage*> depleted_storage_ptr_list;
```

```
518     std::list<Storage*> nondepleted_storage_ptr_list;
519
520     Storage* storage_ptr;
521     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
522         storage_ptr = storage_ptr_vec_ptr->at(i);
523
524         if (storage_ptr->is_depleted) {
525             depleted_storage_ptr_list.push_back(storage_ptr);
526         }
527
528         else {
529             nondepleted_storage_ptr_list.push_back(storage_ptr);
530         }
531     }
532
533     //  3. discharge non-depleted storage assets
534     net_load_kW = this->__handleStorageDischarging(
535         timestep,
536         dt_hrs,
537         net_load_kW,
538         nondepleted_storage_ptr_list
539     );
540
541     //  4. request optimal production from all Noncombustion assets
542     net_load_kW = this->__handleNoncombustionDispatch(
543         timestep,
544         dt_hrs,
545         net_load_kW,
546         noncombustion_ptr_vec_ptr,
547         resources_ptr
548     );
549
550     //  5. request optimal production from all Combustion assets
551     //     default to load following if no depleted storage
552     if (depleted_storage_ptr_list.empty()) {
553         net_load_kW = this->__handleCombustionDispatch(
554             timestep,
555             dt_hrs,
556             net_load_kW,
557             combustion_ptr_vec_ptr,
558             false   // is_cycle_charging
559         );
560     }
561
562     else {
563         net_load_kW = this->__handleCombustionDispatch(
564             timestep,
565             dt_hrs,
566             net_load_kW,
567             combustion_ptr_vec_ptr,
568             true    // is_cycle_charging
569         );
570     }
571
572     //  6. attempt to charge depleted Storage assets using any and all available
574     //     charge priority is Combustion, then Renewable
575     this->__handleStorageCharging(
576         timestep,
577         dt_hrs,
578         depleted_storage_ptr_list,
579         combustion_ptr_vec_ptr,
580         noncombustion_ptr_vec_ptr,
581         renewable_ptr_vec_ptr
582     );
583
584     //  7. record any missed load
585     if (net_load_kW > 1e-6) {
586         this->missed_load_vec_kW[timestep] = net_load_kW;
587     }
588
589     return;
590 }   /* __applyCycleChargingControl_DISCHARGING() */
```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```
void Controller::__applyLoadFollowingControl_CHARGING (
            int timestep,
            ElectricalLoad * electrical_load_ptr,
```

```
            Resources * resources_ptr,
            std::vector< Combustion * > * combustion_ptr_vec_ptr,
            std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr,
            std::vector< Storage * > * storage_ptr_vec_ptr )  [private]
```

Helper method to apply load following control action for given timestep of the Model run when net load <= 0;.

Parameters

timestep	The current time step of the Model run.
electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
storage_ptr_vec_ptr	A pointer to the Storage pointer vector of the Model.

```
255 {
256     //  1. get dt_hrs, set net load
257     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
258     double net_load_kW = 0;
259 
260     //  2. request zero production from all Combustion assets
261     this->__handleCombustionDispatch(
262         timestep,
263         dt_hrs,
264         net_load_kW,
265         combustion_ptr_vec_ptr,
266         false   // is_cycle_charging
267     );
268 
269     //  3. request zero production from all Noncombustion assets
270     this->__handleNoncombustionDispatch(
271         timestep,
272         dt_hrs,
273         net_load_kW,
274         noncombustion_ptr_vec_ptr,
275         resources_ptr
276     );
277 
278     //  4. attempt to charge all Storage assets using any and all available curtailment
279     //     charge priority is Combustion, then Renewable
280     this->__handleStorageCharging(
281         timestep,
282         dt_hrs,
283         storage_ptr_vec_ptr,
284         combustion_ptr_vec_ptr,
285         noncombustion_ptr_vec_ptr,
286         renewable_ptr_vec_ptr
287     );
288 
289     return;
290 }   /* __applyLoadFollowingControl_CHARGING() */
```

4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

```
void Controller::__applyLoadFollowingControl_DISCHARGING (
            int timestep,
            ElectricalLoad * electrical_load_ptr,
            Resources * resources_ptr,
            std::vector< Combustion * > * combustion_ptr_vec_ptr,
            std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr,
            std::vector< Storage * > * storage_ptr_vec_ptr )  [private]
```

Helper method to apply load following control action for given timestep of the Model run when net load > 0;.

Parameters

timestep	The current time step of the Model run.
electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
storage_ptr_vec_ptr	A pointer to the Storage pointer vector of the Model.

curtailment

```
337 {
338     //  1. get dt_hrs, net load
339     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
340     double net_load_kW = this->net_load_vec_kW[timestep];
341 
342     //  2. partition Storage assets into depleted and non-depleted
343     std::list<Storage*> depleted_storage_ptr_list;
344     std::list<Storage*> nondepleted_storage_ptr_list;
345 
346     Storage* storage_ptr;
347     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
348         storage_ptr = storage_ptr_vec_ptr->at(i);
349 
350         if (storage_ptr->is_depleted) {
351             depleted_storage_ptr_list.push_back(storage_ptr);
352         }
353 
354         else {
355             nondepleted_storage_ptr_list.push_back(storage_ptr);
356         }
357     }
358 
359     //  3. discharge non-depleted storage assets
360     net_load_kW = this->__handleStorageDischarging(
361         timestep,
362         dt_hrs,
363         net_load_kW,
364         nondepleted_storage_ptr_list
365     );
366 
367     //  4. request optimal production from all Noncombustion assets
368     net_load_kW = this->__handleNoncombustionDispatch(
369         timestep,
370         dt_hrs,
371         net_load_kW,
372         noncombustion_ptr_vec_ptr,
373         resources_ptr
374     );
375 
376     //  5. request optimal production from all Combustion assets
377     net_load_kW = this->__handleCombustionDispatch(
378         timestep,
379         dt_hrs,
380         net_load_kW,
381         combustion_ptr_vec_ptr,
382         false   // is_cycle_charging
383     );
384 
385     //  6. attempt to charge depleted Storage assets using any and all available
387     //     charge priority is Combustion, then Renewable
388     this->__handleStorageCharging(
389         timestep,
390         dt_hrs,
391         depleted_storage_ptr_list,
392         combustion_ptr_vec_ptr,
393         noncombustion_ptr_vec_ptr,
394         renewable_ptr_vec_ptr
395     );
396 
397     //  7. record any missed load
398     if (net_load_kW > 1e-6) {
399         this->missed_load_vec_kW[timestep] = net_load_kW;
400     }
401 
402     return;
403 }   /* __applyLoadFollowingControl_DISCHARGING() */
```

4.3.3.5 __computeNetLoad()

```
void Controller::__computeNetLoad (
            ElectricalLoad * electrical_load_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr,
            Resources * resources_ptr )  [private]
```

Helper method to compute and populate the net load vector.

The net load at a given point in time is defined as the load at that point in time, minus the sum of all Renewable production at that point in time. Therefore, a negative net load indicates a surplus of Renewable production, and a positive net load indicates a deficit of Renewable production.

Parameters

electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
resources_ptr	A pointer to the Resources component of the Model.

```
57 {
58     //  1. init
59     this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
60     this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
61 
62     //  2. populate net load vector
63     double dt_hrs = 0;
64     double load_kW = 0;
65     double net_load_kW = 0;
66     double production_kW = 0;
67 
68     Renewable* renewable_ptr;
69 
70     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
71         dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
72         load_kW = electrical_load_ptr->load_vec_kW[i];
73         net_load_kW = load_kW;
74 
75         for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {
76             renewable_ptr = renewable_ptr_vec_ptr->at(j);
77 
78             production_kW = this->__getRenewableProduction(
79                 i,
80                 dt_hrs,
81                 renewable_ptr,
82                 resources_ptr
83             );
84 
85             load_kW = renewable_ptr->commit(
86                 i,
87                 dt_hrs,
88                 production_kW,
89                 load_kW
90             );
91 
92             net_load_kW -= production_kW;
93         }
94 
95         this->net_load_vec_kW[i] = net_load_kW;
96     }
97 
98     return;
99 }   /* __computeNetLoad() */
```

4.3.3.6 __constructCombustionMap()

```
void Controller::__constructCombustionMap (
            std::vector< Combustion * > * combustion_ptr_vec_ptr )  [private]
```

Helper method to construct a Combustion map, for use in determining.

Parameters

combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.

```
121 {
122     //  1. get state table dimensions
123     int n_cols = combustion_ptr_vec_ptr->size();
124     int n_rows = pow(2, n_cols);
125 
126     //  2. init state table (all possible on/off combinations)
127     std::vector<std::vector<bool» state_table;
128     state_table.resize(n_rows, {});
129 
130     int x = 0;
131     for (int i = 0; i < n_rows; i++) {
132         state_table[i].resize(n_cols, false);
133 
134         x = i;
135         for (int j = 0; j < n_cols; j++) {
136             if (x % 2 == 0) {
137                 state_table[i][j] = true;
138             }
139             x /= 2;
140         }
141     }
142 
143     //  3. construct combustion map (handle duplicates by keeping rows with minimum
144     //     trues)
145     double total_capacity_kW = 0;
146     int truth_count = 0;
147     int current_truth_count = 0;
148 
149     for (int i = 0; i < n_rows; i++) {
150         total_capacity_kW = 0;
151         truth_count = 0;
152         current_truth_count = 0;
153 
154         for (int j = 0; j < n_cols; j++) {
155             if (state_table[i][j]) {
156                 total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
157                 truth_count++;
158             }
159         }
160 
161         if (this->combustion_map.count(total_capacity_kW) > 0) {
162             for (int j = 0; j < n_cols; j++) {
163                 if (this->combustion_map[total_capacity_kW][j]) {
164                     current_truth_count++;
165                 }
166             }
167 
168             if (truth_count < current_truth_count) {
169                 this->combustion_map.erase(total_capacity_kW);
170             }
171         }
172 
173         this->combustion_map.insert(
174             std::pair<double, std::vector<bool» (
175                 total_capacity_kW,
176                 state_table[i]
177             )
178         );
179     }
180 
181     /*
182     // ==== TEST PRINT ==== //
183     std::cout « std::endl;
184 
185     std::cout « "\t\t";
186     for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
187         std::cout « combustion_ptr_vec_ptr->at(i)->capacity_kW « "\t";
188     }
189     std::cout « std::endl;
190 
191     std::map<double, std::vector<bool»::iterator iter;
192     for (
193         iter = this->combustion_map.begin();
194         iter != this->combustion_map.end();
195         iter++
196     ) {
197         std::cout « iter->first « ":\t{\t";
198 
199         for (size_t i = 0; i < iter->second.size(); i++) {
200             std::cout « iter->second[i] « "\t";
201         }
202         std::cout « "}" « std::endl;
```

```
203     }
204     // ==== END TEST PRINT ==== //
205     //*/
206
207     return;
208 }   /* __constructCombustionTable() */
```

4.3.3.7 __getRenewableProduction()

```
double Controller::__getRenewableProduction (
            int timestep,
            double dt_hrs,
            Renewable * renewable_ptr,
            Resources * resources_ptr )  [private]
```

Helper method to compute the production from the given Renewable asset at the given point in time.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
renewable_ptr	A pointer to the Renewable asset.
resources_ptr	A pointer to the Resources component of the Model.

Returns

The production [kW] of the Renewable asset.

```
879 {
880     double production_kW = 0;
881
882     switch (renewable_ptr->type) {
883         case (RenewableType :: SOLAR): {
884             double resource_value = 0;
885
886             if (not renewable_ptr->normalized_production_series_given) {
887                 resource_value =
888                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
889             }
890
891             production_kW = renewable_ptr->computeProductionkW(
892                 timestep,
893                 dt_hrs,
894                 resource_value
895             );
896
897             break;
898         }
899
900         case (RenewableType :: TIDAL): {
901             double resource_value = 0;
902
903             if (not renewable_ptr->normalized_production_series_given) {
904                 resource_value =
905                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
906             }
907
908             production_kW = renewable_ptr->computeProductionkW(
909                 timestep,
910                 dt_hrs,
911                 resource_value
912             );
913
914             break;
915         }
916
917         case (RenewableType :: WAVE): {
918             double significant_wave_height_m = 0;
```

```
919             double energy_period_s = 0;
920
921             if (not renewable_ptr->normalized_production_series_given) {
922                 significant_wave_height_m =
923                     resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0];
924
925                 energy_period_s =
926                     resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1];
927             }
928
929             production_kW = renewable_ptr->computeProductionkW(
930                 timestep,
931                 dt_hrs,
932                 significant_wave_height_m,
933                 energy_period_s
934             );
935
936             break;
937         }
938
939         case (RenewableType :: WIND): {
940             double resource_value = 0;
941
942             if (not renewable_ptr->normalized_production_series_given) {
943                 resource_value =
944                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
945             }
946
947             production_kW = renewable_ptr->computeProductionkW(
948                 timestep,
949                 dt_hrs,
950                 resource_value
951             );
952
953             break;
954         }
955
956         default: {
957             std::string error_str = "ERROR:  Controller::__getRenewableProduction():  ";
958             error_str += "renewable type ";
959             error_str += std::to_string(renewable_ptr->type);
960             error_str += " not recognized";
961
962             #ifdef _WIN32
963                 std::cout « error_str « std::endl;
964             #endif
965
966             throw std::runtime_error(error_str);
967
968             break;
969         }
970     }
971
972     return production_kW;
973 }   /* __getRenewableProduction() */
```

4.3.3.8 __handleCombustionDispatch()

```
double Controller::__handleCombustionDispatch (
            int timestep,
            double dt_hrs,
            double net_load_kW,
            std::vector< Combustion * > * combustion_ptr_vec_ptr,
            bool is_cycle_charging )  [private]
```

bool is_cycle_charging)

Helper method to handle the optimal dispatch of Combustion assets. Dispatches for 1.2x the received net load, so as to ensure a "20% spinning reserve". Dispatches a minimum number of Combustion assets, which then share the load proportional to their rated capacities.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
net_load_kW	The net load [kW] before the dispatch is deducted from it.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
is_cycle_charging	A boolean which defines whether to apply cycle charging logic or not.

Returns

The net load [kW] remaining after the dispatch is deducted from it.

```
1016 {
1017     //  1. get minimal Combustion dispatch
1018     double target_production_kW = 1.2 * net_load_kW;
1019     double total_capacity_kW = 0;
1020 
1021     std::map<double, std::vector<bool»::iterator iter = this->combustion_map.begin();
1022     while (iter != std::prev(this->combustion_map.end(), 1)) {
1023         if (target_production_kW <= total_capacity_kW) {
1024             break;
1025         }
1026 
1027         iter++;
1028         total_capacity_kW = iter->first;
1029     }
1030 
1031     //  2. share load proportionally (by rated capacity) over active diesels
1032     Combustion* combustion_ptr;
1033     double production_kW = 0;
1034     double request_kW = 0;
1035     double _net_load_kW = net_load_kW;
1036 
1037     for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
1038         combustion_ptr = combustion_ptr_vec_ptr->at(i);
1039 
1040         if (total_capacity_kW > 0) {
1041             request_kW =
1042                 int(this->combustion_map[total_capacity_kW][i]) *
1043                 net_load_kW *
1044                 (combustion_ptr->capacity_kW / total_capacity_kW);
1045         }
1046 
1047         else {
1048             request_kW = 0;
1049         }
1050 
1051         if (is_cycle_charging and request_kW > 0) {
1052             if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
1053                 request_kW = 0.85 * combustion_ptr->capacity_kW;
1054             }
1055         }
1056 
1057         production_kW = combustion_ptr->requestProductionkW(
1058             timestep,
1059             dt_hrs,
1060             request_kW
1061         );
1062 
1063         _net_load_kW = combustion_ptr->commit(
1064             timestep,
1065             dt_hrs,
1066             production_kW,
1067             _net_load_kW
1068         );
1069     }
1070 
1071     return _net_load_kW;
1072 }   /* __handleCombustionDispatch() */
```

4.3.3.9 __handleNoncombustionDispatch()

```
double Controller::__handleNoncombustionDispatch (
            int timestep,
```

```
            double dt_hrs,
            double net_load_kW,
            std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
            Resources * resources_ptr )  [private]
{
    Noncombustion* noncombustion_ptr;
    double production_kW = 0;

    for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
        noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);

        switch (noncombustion_ptr->type) {
            case (NoncombustionType :: HYDRO): {
                double resource_value = 0;

                if (not noncombustion_ptr->normalized_production_series_given) {
                    resource_value =
                        resources_ptr->resource_map_1D[noncombustion_ptr->resource_key][timestep];
                }

                production_kW = noncombustion_ptr->requestProductionkW(
                    timestep,
                    dt_hrs,
                    net_load_kW,
                    resource_value
                );

                net_load_kW = noncombustion_ptr->commit(
                    timestep,
                    dt_hrs,
                    production_kW,
                    net_load_kW,
                    resource_value
                );

                break;
            }

            default: {
                production_kW = noncombustion_ptr->requestProductionkW(
                    timestep,
                    dt_hrs,
                    net_load_kW
                );

                net_load_kW = noncombustion_ptr->commit(
                    timestep,
                    dt_hrs,
                    production_kW,
                    net_load_kW
                );

                break;
            }
        }
    }

    return net_load_kW;
}   /* __handleNoncombustionDispatch() */
```

4.3.3.10 __handleStorageCharging() [1/2]

```
void Controller::__handleStorageCharging (
            int timestep,
            double dt_hrs,
            std::list< Storage * > storage_ptr_list,
            std::vector< Combustion * > * combustion_ptr_vec_ptr,
            std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr )  [private]
```

Helper method to handle the charging of the given Storage assets.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
storage_ptr_list	A list of pointers to the Storage assets that are to be charged.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.

```
633 {
634     double acceptable_kW = 0;
635     double curtailment_kW = 0;
636 
637     Storage* storage_ptr;
638     Combustion* combustion_ptr;
639     Noncombustion* noncombustion_ptr;
640     Renewable* renewable_ptr;
641 
642     std::list<Storage*>::iterator iter;
643     for (
644         iter = storage_ptr_list.begin();
645         iter != storage_ptr_list.end();
646         iter++
647     ){
648         storage_ptr = (*iter);
649 
650         //  1. attempt to charge from Combustion curtailment first
651         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
652             combustion_ptr = combustion_ptr_vec_ptr->at(i);
653             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
654 
655             if (curtailment_kW <= 0) {
656                 continue;
657             }
658 
659             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
660 
661             if (acceptable_kW > curtailment_kW) {
662                 acceptable_kW = curtailment_kW;
663             }
664 
665             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
666             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
667             storage_ptr->power_kW += acceptable_kW;
668         }
669 
670         //  2. attempt to charge from Noncombustion curtailment second
671         for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
672             noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
673             curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
674 
675             if (curtailment_kW <= 0) {
676                 continue;
677             }
678 
679             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
680 
681             if (acceptable_kW > curtailment_kW) {
682                 acceptable_kW = curtailment_kW;
683             }
684 
685             noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
686             noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
687             storage_ptr->power_kW += acceptable_kW;
688         }
689 
690         //  3. attempt to charge from Renewable curtailment third
691         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
692             renewable_ptr = renewable_ptr_vec_ptr->at(i);
693             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
694 
695             if (curtailment_kW <= 0) {
696                 continue;
697             }
698 
699             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
700 
701             if (acceptable_kW > curtailment_kW) {
702                 acceptable_kW = curtailment_kW;
703             }
704 
```

```
705             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
706             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
707             storage_ptr->power_kW += acceptable_kW;
708         }
709
710         //  4. commit charge
711         storage_ptr->commitCharge(
712             timestep,
713             dt_hrs,
714             storage_ptr->power_kW
715         );
716     }
717
718     return;
719 }   /* __handleStorageCharging() */
```

4.3.3.11 __handleStorageCharging() [2/2]

```
void Controller::__handleStorageCharging (
              int timestep,
              double dt_hrs,
              std::vector< Storage * > * storage_ptr_vec_ptr,
              std::vector< Combustion * > * combustion_ptr_vec_ptr,
              std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
              std::vector< Renewable * > * renewable_ptr_vec_ptr )  [private]
```

Helper method to handle the charging of the given Storage assets.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
storage_ptr_vec_ptr	A pointer to a vector of pointers to the Storage assets that are to be charged.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.

```
762 {
763     double acceptable_kW = 0;
764     double curtailment_kW = 0;
765
766     Storage* storage_ptr;
767     Combustion* combustion_ptr;
768     Noncombustion* noncombustion_ptr;
769     Renewable* renewable_ptr;
770
771     for (size_t j = 0; j < storage_ptr_vec_ptr->size(); j++) {
772         storage_ptr = storage_ptr_vec_ptr->at(j);
773
774         //  1. attempt to charge from Combustion curtailment first
775         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
776             combustion_ptr = combustion_ptr_vec_ptr->at(i);
777             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
778
779             if (curtailment_kW <= 0) {
780                 continue;
781             }
782
783             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
784
785             if (acceptable_kW > curtailment_kW) {
786                 acceptable_kW = curtailment_kW;
787             }
788
789             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
790             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
791             storage_ptr->power_kW += acceptable_kW;
792         }
```

```
793
794         //  2. attempt to charge from Noncombustion curtailment second
795         for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
796             noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
797             curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
798
799             if (curtailment_kW <= 0) {
800                 continue;
801             }
802
803             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
804
805             if (acceptable_kW > curtailment_kW) {
806                 acceptable_kW = curtailment_kW;
807             }
808
809             noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
810             noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
811             storage_ptr->power_kW += acceptable_kW;
812         }
813
814         //  3. attempt to charge from Renewable curtailment third
815         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
816             renewable_ptr = renewable_ptr_vec_ptr->at(i);
817             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
818
819             if (curtailment_kW <= 0) {
820                 continue;
821             }
822
823             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
824
825             if (acceptable_kW > curtailment_kW) {
826                 acceptable_kW = curtailment_kW;
827             }
828
829             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
830             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
831             storage_ptr->power_kW += acceptable_kW;
832         }
833
834         //  4. commit charge
835         storage_ptr->commitCharge(
836             timestep,
837             dt_hrs,
838             storage_ptr->power_kW
839         );
840     }
841
842     return;
843 }   /* __handleStorageCharging() */
```

4.3.3.12 __handleStorageDischarging()

```
double Controller::__handleStorageDischarging (
            int timestep,
            double dt_hrs,
            double net_load_kW,
            std::list< Storage * > storage_ptr_list )  [private]
```

Helper method to handle the discharging of the given Storage assets.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
storage_ptr_list	A list of pointers to the Storage assets that are to be discharged.

Returns

The net load [kW] remaining after the discharge is deducted from it.

```
1201 {
1202     double discharging_kW = 0;
1203 
1204     Storage* storage_ptr;
1205 
1206     std::list<Storage*>::iterator iter;
1207     for (
1208         iter = storage_ptr_list.begin();
1209         iter != storage_ptr_list.end();
1210         iter++
1211     ){
1212         storage_ptr = (*iter);
1213 
1214         discharging_kW = storage_ptr->getAvailablekW(dt_hrs);
1215 
1216         if (discharging_kW > net_load_kW) {
1217             discharging_kW = net_load_kW;
1218         }
1219 
1220         net_load_kW = storage_ptr->commitDischarge(
1221             timestep,
1222             dt_hrs,
1223             discharging_kW,
1224             net_load_kW
1225         );
1226     }
1227 
1228     return net_load_kW;
1229 }   /* __handleStorageDischarging() */
```

4.3.3.13 applyDispatchControl()

```
void Controller::applyDispatchControl (
              ElectricalLoad * electrical_load_ptr,
              Resources * resources_ptr,
              std::vector< Combustion * > * combustion_ptr_vec_ptr,
              std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
              std::vector< Renewable * > * renewable_ptr_vec_ptr,
              std::vector< Storage * > * storage_ptr_vec_ptr )
```

Method to apply dispatch control at every point in the modelling time series.

Parameters

electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
noncombustion_ptr_vec_ptr	A pointer to the Noncombustion pointer vector of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
storage_ptr_vec_ptr	A pointer to the Storage pointer vector of the Model.

```
1381 {
1382     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
1383         switch (this->control_mode) {
1384             case (ControlMode :: LOAD_FOLLOWING): {
1385                 if (this->net_load_vec_kW[i] <= 0) {
1386                     this->__applyLoadFollowingControl_CHARGING(
1387                         i,
1388                         electrical_load_ptr,
1389                         resources_ptr,
1390                         combustion_ptr_vec_ptr,
1391                         noncombustion_ptr_vec_ptr,
1392                         renewable_ptr_vec_ptr,
1393                         storage_ptr_vec_ptr
```

```
1394                     );
1395                 }
1396 
1397                 else {
1398                     this->__applyLoadFollowingControl_DISCHARGING(
1399                         i,
1400                         electrical_load_ptr,
1401                         resources_ptr,
1402                         combustion_ptr_vec_ptr,
1403                         noncombustion_ptr_vec_ptr,
1404                         renewable_ptr_vec_ptr,
1405                         storage_ptr_vec_ptr
1406                     );
1407                 }
1408 
1409                 break;
1410             }
1411 
1412             case (ControlMode :: CYCLE_CHARGING): {
1413                 if (this->net_load_vec_kW[i] <= 0) {
1414                     this->__applyCycleChargingControl_CHARGING(
1415                         i,
1416                         electrical_load_ptr,
1417                         resources_ptr,
1418                         combustion_ptr_vec_ptr,
1419                         noncombustion_ptr_vec_ptr,
1420                         renewable_ptr_vec_ptr,
1421                         storage_ptr_vec_ptr
1422                     );
1423                 }
1424 
1425                 else {
1426                     this->__applyCycleChargingControl_DISCHARGING(
1427                         i,
1428                         electrical_load_ptr,
1429                         resources_ptr,
1430                         combustion_ptr_vec_ptr,
1431                         noncombustion_ptr_vec_ptr,
1432                         renewable_ptr_vec_ptr,
1433                         storage_ptr_vec_ptr
1434                     );
1435                 }
1436 
1437                 break;
1438             }
1439 
1440             default: {
1441                 std::string error_str = "ERROR:  Controller :: applyDispatchControl():  ";
1442                 error_str += "control mode ";
1443                 error_str += std::to_string(this->control_mode);
1444                 error_str += " not recognized";
1445 
1446                 #ifdef _WIN32
1447                     std::cout « error_str « std::endl;
1448                 #endif
1449 
1450                 throw std::runtime_error(error_str);
1451 
1452                 break;
1453             }
1454         }
1455     }
1456 
1457     return;
1458 }   /* applyDispatchControl() */
```

4.3.3.14 clear()

```
void Controller::clear (
            void  )
```

Method to clear all attributes of the Controller object.

```
1473 {
1474     this->net_load_vec_kW.clear();
1475     this->missed_load_vec_kW.clear();
1476     this->combustion_map.clear();
1477 
1478     return;
1479 }   /* clear() */
```

4.3.3.15 init()

```
void Controller::init (
            ElectricalLoad * electrical_load_ptr,
            std::vector< Renewable * > * renewable_ptr_vec_ptr,
            Resources * resources_ptr,
            std::vector< Combustion * > * combustion_ptr_vec_ptr )
```

Method to initialize the Controller component of the Model.

Parameters

electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
resources_ptr	A pointer to the Resources component of the Model.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.

```
1331 {
1332     //  1. compute net load
1333     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);
1334 
1335     //  2. construct Combustion table
1336     this->__constructCombustionMap(combustion_ptr_vec_ptr);
1337 
1338     return;
1339 }   /* init() */
```

4.3.3.16 setControlMode()

```
void Controller::setControlMode (
            ControlMode control_mode )
```

Parameters

control_mode	The ControlMode which is to be active in the Controller.

```
1265 {
1266     this->control_mode = control_mode;
1267 
1268     switch(control_mode) {
1269         case (ControlMode :: LOAD_FOLLOWING): {
1270             this->control_string = "LOAD_FOLLOWING";
1271 
1272             break;
1273         }
1274 
1275         case (ControlMode :: CYCLE_CHARGING): {
1276             this->control_string = "CYCLE_CHARGING";
1277 
1278             break;
1279         }
1280 
1281         default: {
1282             std::string error_str = "ERROR:  Controller :: setControlMode():  ";
1283                 error_str += "control mode ";
1284                 error_str += std::to_string(control_mode);
1285                 error_str += " not recognized";
1286 
1287                 #ifdef _WIN32
1288                     std::cout « error_str « std::endl;
1289                 #endif
1290 
1291                 throw std::runtime_error(error_str);
1292 
1293             break;
```

```
1294         }
1295     }
1296
1297     return;
1298 }   /* setControlMode() */
```

4.3.4 Member Data Documentation

4.3.4.1 combustion_map

`std::map<double, std::vector<bool> > Controller::combustion_map`

A map of all possible combustion states, for use in determining optimal dispatch.

4.3.4.2 control_mode

`ControlMode Controller::control_mode`

The ControlMode that is active in the Model.

4.3.4.3 control_string

`std::string Controller::control_string`

A string describing the active ControlMode.

4.3.4.4 missed_load_vec_kW

`std::vector<double> Controller::missed_load_vec_kW`

A vector of missed load values [kW] at each point in the modelling time series.

4.3.4.5 net_load_vec_kW

`std::vector<double> Controller::net_load_vec_kW`

A vector of net load values [kW] at each point in the modelling time series. Net load is defined as load minus all available Renewable production.

The documentation for this class was generated from the following files:

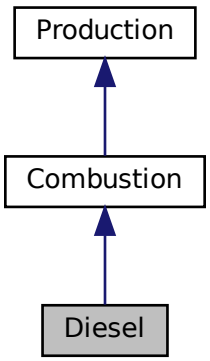
- header/Controller.h
- source/Controller.cpp

4.4 Diesel Class Reference

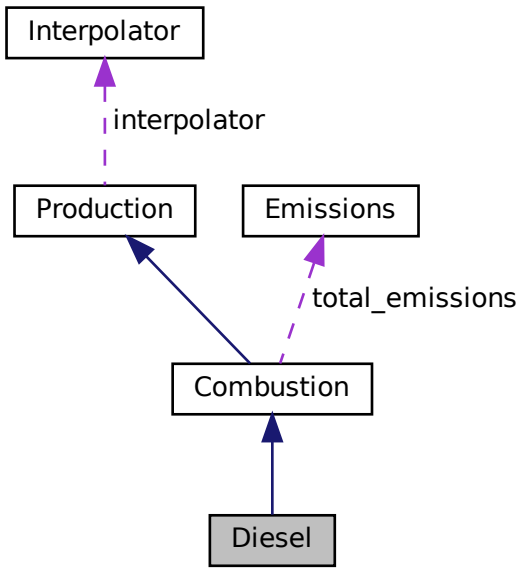
A derived class of the Combustion branch of Production which models production using a diesel generator.

`#include <Diesel.h>`

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

- Diesel (void)

  Constructor (dummy) for the Diesel class.
- Diesel (int, double, DieselInputs, std::vector< double > *)

  Constructor (intended) for the Diesel class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double requestProductionkW (int, double, double)

  Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Diesel (void)

  Destructor for the Diesel class.

Public Attributes

- double minimum_load_ratio

  The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.
- double minimum_runtime_hrs

  The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.
- double time_since_last_start_hrs

  The time that has elapsed [hrs] since the last start of the asset.

Private Member Functions

- void __checkInputs (DieselInputs)

  Helper method to check inputs to the Diesel constructor.
- void __handleStartStop (int, double, double)

  Helper method (private) to handle the starting/stopping of the diesel generator. The minimum runtime constraint is enforced in this method.
- double __getGenericFuelSlope (void)

  Helper method to generate a generic, linearized fuel consumption slope for a diesel generator.
- double __getGenericFuelIntercept (void)

  Helper method to generate a generic, linearized fuel consumption intercept for a diesel generator.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic diesel generator capital cost.
- double __getGenericOpMaintCost (void)

  Helper method (private) to generate a generic diesel generator operation and maintenance cost. This is a cost incurred per unit energy produced.
- void __writeSummary (std::string)

  Helper method to write summary results for Diesel.
- void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

  Helper method to write time series results for Diesel.

4.4.1 Detailed Description

A derived class of the Combustion branch of Production which models production using a diesel generator.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Diesel() [1/2]

```
Diesel::Diesel (
            void  )
```

Constructor (dummy) for the Diesel class.

```
604 {
605     return;
606 }   /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

```
Diesel::Diesel (
            int n_points,
            double n_years,
            DieselInputs diesel_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Diesel class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
diesel_inputs	A structure of Diesel constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
638   :
639 Combustion(
640     n_points,
641     n_years,
642     diesel_inputs.combustion_inputs,
643     time_vec_hrs_ptr
644 )
645 {
646     //  1. check inputs
647     this->__checkInputs(diesel_inputs);
648
649     //  2. set attributes
650     this->type = CombustionType :: DIESEL;
651     this->type_str = "DIESEL";
652
653     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
654
655     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
656
657     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
658     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
659     this->time_since_last_start_hrs = 0;
```

```
660
661     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
662     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
663     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
664     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
665     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
666     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
667
668     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
669         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
670     }
671     else {
672         this->linear_fuel_slope_LkWh = diesel_inputs.linear_fuel_slope_LkWh;
673     }
674
675     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
676         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
677     }
678     else {
679         this->linear_fuel_intercept_LkWh = diesel_inputs.linear_fuel_intercept_LkWh;
680     }
681
682     if (diesel_inputs.capital_cost < 0) {
683         this->capital_cost = this->__getGenericCapitalCost();
684     }
685     else {
686         this->capital_cost = diesel_inputs.capital_cost;
687     }
688
689     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
690         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
691     }
692     else {
693         this->operation_maintenance_cost_kWh =
694             diesel_inputs.operation_maintenance_cost_kWh;
695     }
696
697     if (not this->is_sunk) {
698         this->capital_cost_vec[0] = this->capital_cost;
699     }
700
701     //  3. construction print
702     if (this->print_flag) {
703         std::cout << "Diesel object constructed at " << this << std::endl;
704     }
705
706     return;
707 }   /* Diesel() */
```

4.4.2.3 ~Diesel()

```
Diesel::~Diesel (
            void  )
```

Destructor for the Diesel class.

```
869 {
870     //  1. destruction print
871     if (this->print_flag) {
872         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
873     }
874
875     return;
876 }   /* ~Diesel() */
```

4.4.3 Member Function Documentation

4.4.3.1 __checkInputs()

```
void Diesel::__checkInputs (
            DieselInputs diesel_inputs )  [private]
```

Helper method to check inputs to the Diesel constructor.

Parameters

diesel_inputs	A structure of Diesel constructor inputs.

```
39 {
40     //  1. check fuel_cost_L
41     if (diesel_inputs.fuel_cost_L < 0) {
42         std::string error_str = "ERROR:  Diesel():  ";
43         error_str += "DieselInputs::fuel_cost_L must be >= 0";
44 
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48 
49         throw std::invalid_argument(error_str);
50     }
51 
52     //  2. check CO2_emissions_intensity_kgL
53     if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
54         std::string error_str = "ERROR:  Diesel():  ";
55         error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";
56 
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60 
61         throw std::invalid_argument(error_str);
62     }
63 
64     //  3. check CO_emissions_intensity_kgL
65         if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
66         std::string error_str = "ERROR:  Diesel():  ";
67         error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";
68 
69         #ifdef _WIN32
70             std::cout << error_str << std::endl;
71         #endif
72 
73         throw std::invalid_argument(error_str);
74     }
75 
76     //  4. check NOx_emissions_intensity_kgL
77     if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
78         std::string error_str = "ERROR:  Diesel():  ";
79         error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";
80 
81         #ifdef _WIN32
82             std::cout << error_str << std::endl;
83         #endif
84 
85         throw std::invalid_argument(error_str);
86     }
87 
88     //  5. check SOx_emissions_intensity_kgL
89     if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
90         std::string error_str = "ERROR:  Diesel():  ";
91         error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";
92 
93         #ifdef _WIN32
94             std::cout << error_str << std::endl;
95         #endif
96 
97         throw std::invalid_argument(error_str);
98     }
99 
100     //  6. check CH4_emissions_intensity_kgL
101     if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
102         std::string error_str = "ERROR:  Diesel():  ";
103         error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";
104 
105         #ifdef _WIN32
106             std::cout << error_str << std::endl;
107         #endif
108 
109         throw std::invalid_argument(error_str);
110     }
111 
112     //  7. check PM_emissions_intensity_kgL
113     if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
114         std::string error_str = "ERROR:  Diesel():  ";
115         error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";
116 
117         #ifdef _WIN32
118             std::cout << error_str << std::endl;
119         #endif
120 
```

```
121         throw std::invalid_argument(error_str);
122     }
123 
124     //  8. check minimum_load_ratio
125     if (diesel_inputs.minimum_load_ratio < 0) {
126         std::string error_str = "ERROR:  Diesel():  ";
127         error_str += "DieselInputs::minimum_load_ratio must be >= 0";
128 
129         #ifdef _WIN32
130             std::cout << error_str << std::endl;
131         #endif
132 
133         throw std::invalid_argument(error_str);
134     }
135 
136     //  9. check minimum_runtime_hrs
137     if (diesel_inputs.minimum_runtime_hrs < 0) {
138         std::string error_str = "ERROR:  Diesel():  ";
139         error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";
140 
141         #ifdef _WIN32
142             std::cout << error_str << std::endl;
143         #endif
144 
145         throw std::invalid_argument(error_str);
146     }
147 
148     //  10. check replace_running_hrs
149     if (diesel_inputs.replace_running_hrs <= 0) {
150         std::string error_str = "ERROR:  Diesel():  ";
151         error_str += "DieselInputs::replace_running_hrs must be > 0";
152 
153         #ifdef _WIN32
154             std::cout << error_str << std::endl;
155         #endif
156 
157         throw std::invalid_argument(error_str);
158     }
159 
160     return;
161 }   /* __checkInputs() */
```

4.4.3.2 __getGenericCapitalCost()

```
double Diesel::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic diesel generator capital cost.

This model was obtained by way of surveying an assortment of published diesel generator costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD].

Returns

A generic capital cost for the diesel generator [CAD].

```
238 {
239     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.425) + 800;
240 
241     return capital_cost_per_kW * this->capacity_kW;
242 }   /* __getGenericCapitalCost() */
```

4.4.3.3 __getGenericFuelIntercept()

```
double Diesel::__getGenericFuelIntercept (
            void  )  [private]
```

Helper method to generate a generic, linearized fuel consumption intercept for a diesel generator.

This model was obtained by way of surveying an assortment of published diesel generator fuel consumption data, and then constructing a best fit model.

Ref: HOMER [2023c]
Ref: HOMER [2023d]

Returns

A generic fuel intercept coefficient for the diesel generator [L/kWh].

```
213 {
214     double linear_fuel_intercept_LkWh = 0.0940 * pow(this->capacity_kW, -0.2735);
215
216     return linear_fuel_intercept_LkWh;
217 }   /* __getGenericFuelIntercept() */
```

4.4.3.4 __getGenericFuelSlope()

```
double Diesel::__getGenericFuelSlope (
            void  )  [private]
```

Helper method to generate a generic, linearized fuel consumption slope for a diesel generator.

This model was obtained by way of surveying an assortment of published diesel generator fuel consumption data, and then constructing a best fit model.

Ref: HOMER [2023c]
Ref: HOMER [2023e]

Returns

A generic fuel slope for the diesel generator [L/kWh].

```
185 {
186     double linear_fuel_slope_LkWh = 0.4234 * pow(this->capacity_kW, -0.1012);
187
188     return linear_fuel_slope_LkWh;
189 }   /* __getGenericFuelSlope() */
```

4.4.3.5 __getGenericOpMaintCost()

```
double Diesel::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method (private) to generate a generic diesel generator operation and maintenance cost. This is a cost incurred per unit energy produced.

This model was obtained by way of surveying an assortment of published diesel generator costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars per kiloWatt-hour production [CAD/kWh].

Returns

A generic operation and maintenance cost, per unit energy produced, for the diesel generator [CAD/kWh].

```
266 {
267     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
268 
269     return operation_maintenance_cost_kWh;
270 }   /* __getGenericOpMaintCost() */
```

4.4.3.6 __handleStartStop()

```
void Diesel::__handleStartStop (
            int timestep,
            double dt_hrs,
            double production_kW )  [private]
```

Helper method (private) to handle the starting/stopping of the diesel generator. The minimum runtime constraint is enforced in this method.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
production_kW	The current rate of production [kW] of the generator.

```
300 {
301     /*
302      *  Helper method (private) to handle the starting/stopping of the diesel
303      *  generator. The minimum runtime constraint is enforced in this method.
304      */
305 
306     if (this->is_running) {
307         // handle stopping
308         if (
309             production_kW <= 0 and
310             this->time_since_last_start_hrs >= this->minimum_runtime_hrs
311         ) {
312             this->is_running = false;
313         }
314     }
315 
316     else {
317         // handle starting
318         if (production_kW > 0) {
319             this->is_running = true;
320             this->n_starts++;
321             this->time_since_last_start_hrs = 0;
322         }
323     }
324 
```

```
325     return;
326 }   /* __handleStartStop() */
```

4.4.3.7 __writeSummary()

```
void Diesel::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Diesel.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Combustion.

```
345 {
346     //  1. create filestream
347     write_path += "summary_results.md";
348     std::ofstream ofs;
349     ofs.open(write_path, std::ofstream::out);
350 
351     //  2. write to summary results (markdown)
352     ofs « "# ";
353     ofs « std::to_string(int(ceil(this->capacity_kW)));
354     ofs « " kW DIESEL Summary Results\n";
355     ofs « "\n--------\n\n";
356 
357     //  2.1. Production attributes
358     ofs « "## Production Attributes\n";
359     ofs « "\n";
360 
361     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
362     ofs « "\n";
363 
364     ofs « "Production Override: (N = 0 / Y = 1): "
365         « this->normalized_production_series_given « "  \n";
366     if (this->normalized_production_series_given) {
367         ofs « "Path to Normalized Production Time Series: "
368             « this->path_2_normalized_production_time_series « "  \n";
369     }
370     ofs « "\n";
371 
372     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
373     ofs « "Capital Cost: " « this->capital_cost « "  \n";
374     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
375         « " per kWh produced  \n";
376     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
377         « "  \n";
378     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
379         « "  \n";
380     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
381     ofs « "\n";
382 
383     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
384     ofs « "\n--------\n\n";
385 
386     //  2.2. Combustion attributes
387     ofs « "## Combustion Attributes\n";
388     ofs « "\n";
389 
390     ofs « "Fuel Cost: " « this->fuel_cost_L « " per L  \n";
391     ofs « "Nominal Fuel Escalation Rate (annual): "
392         « this->nominal_fuel_escalation_annual « "  \n";
393     ofs « "Real Fuel Escalation Rate (annual): "
394         « this->real_fuel_escalation_annual « "  \n";
395     ofs « "\n";
396 
397     ofs « "Fuel Mode: " « this->fuel_mode_str « "  \n";
398     switch (this->fuel_mode) {
399         case (FuelMode :: FUEL_MODE_LINEAR): {
400             ofs « "Linear Fuel Slope: " « this->linear_fuel_slope_LkWh
```

```
401                 « " L/kWh  \n";
402             ofs « "Linear Fuel Intercept Coefficient: "
403                 « this->linear_fuel_intercept_LkWh « " L/kWh  \n";
404             ofs « "\n";
405 
406             break;
407         }
408 
409         case (FuelMode :: FUEL_MODE_LOOKUP): {
410             ofs « "Fuel Consumption Data: " « this->interpolator.path_map_1D[0]
411                 « "  \n";
412 
413             break;
414         }
415 
416         default: {
417             // write nothing!
418 
419             break;
420         }
421     }
422 
423     ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
424         « this->CO2_emissions_intensity_kgL « " kg/L  \n";
425 
426     ofs « "Carbon Monoxide (CO) Emissions Intensity: "
427         « this->CO_emissions_intensity_kgL « " kg/L  \n";
428 
429     ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
430         « this->NOx_emissions_intensity_kgL « " kg/L  \n";
431 
432     ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
433         « this->SOx_emissions_intensity_kgL « " kg/L  \n";
434 
435     ofs « "Methane (CH4) Emissions Intensity: "
436         « this->CH4_emissions_intensity_kgL « " kg/L  \n";
437 
438     ofs « "Particulate Matter (PM) Emissions Intensity: "
439         « this->PM_emissions_intensity_kgL « " kg/L  \n";
440 
441     ofs « "\n--------\n\n";
442 
443     //  2.3. Diesel attributes
444     ofs « "## Diesel Attributes\n";
445     ofs « "\n";
446 
447     ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « "  \n";
448     ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs  \n";
449 
450     ofs « "\n--------\n\n";
451 
452     //  2.4. Diesel Results
453     ofs « "## Results\n";
454     ofs « "\n";
455 
456     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
457     ofs « "\n";
458 
459     ofs « "Total Dispatch: " « this->total_dispatch_kWh
460         « " kWh  \n";
461 
462     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
463         « " per kWh dispatched  \n";
464     ofs « "\n";
465 
466     ofs « "Running Hours: " « this->running_hours « "  \n";
467     ofs « "Starts: " « this->n_starts « "  \n";
468     ofs « "Replacements: " « this->n_replacements « "  \n";
469 
470     ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
471         « "(Annual Average: " « this->total_fuel_consumed_L / this->n_years
472         « " L/yr)  \n";
473     ofs « "\n";
474 
475     ofs « "Total Carbon Dioxide (CO2) Emissions: " «
476         this->total_emissions.CO2_kg « " kg "
477         « "(Annual Average: " «  this->total_emissions.CO2_kg / this->n_years
478         « " kg/yr)  \n";
479 
480     ofs « "Total Carbon Monoxide (CO) Emissions: " «
481         this->total_emissions.CO_kg « " kg "
482         « "(Annual Average: " «  this->total_emissions.CO_kg / this->n_years
483         « " kg/yr)  \n";
484 
485     ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
486         this->total_emissions.NOx_kg « " kg "
487         « "(Annual Average: " «  this->total_emissions.NOx_kg / this->n_years
```

```
488         « " kg/yr)  \n";
489
490     ofs « "Total Sulfur Oxides (SOx) Emissions: " «
491         this->total_emissions.SOx_kg « " kg "
492         « "(Annual Average: " «  this->total_emissions.SOx_kg / this->n_years
493         « " kg/yr)  \n";
494
495     ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
496         « "(Annual Average: " «  this->total_emissions.CH4_kg / this->n_years
497         « " kg/yr)  \n";
498
499     ofs « "Total Particulate Matter (PM) Emissions: " «
500         this->total_emissions.PM_kg « " kg "
501         « "(Annual Average: " «  this->total_emissions.PM_kg / this->n_years
502         « " kg/yr)  \n";
503
504     ofs « "\n--------\n\n";
505
506     ofs.close();
507     return;
508 }   /* __writeSummary() */
```

4.4.3.8 __writeTimeSeries()

```
void Diesel::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Diesel.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
max_lines	The maximum number of lines of output to write.

Reimplemented from Combustion.

```
538 {
539     //  1. create filestream
540     write_path += "time_series_results.csv";
541     std::ofstream ofs;
542     ofs.open(write_path, std::ofstream::out);
543
544     //  2. write time series results (comma separated value)
545     ofs « "Time (since start of data) [hrs],";
546     ofs « "Production [kW],";
547     ofs « "Dispatch [kW],";
548     ofs « "Storage [kW],";
549     ofs « "Curtailment [kW],";
550     ofs « "Is Running (N = 0 / Y = 1),";
551     ofs « "Fuel Consumption [L],";
552     ofs « "Fuel Cost (actual),";
553     ofs « "Carbon Dioxide (CO2) Emissions [kg],";
554     ofs « "Carbon Monoxide (CO) Emissions [kg],";
555     ofs « "Nitrogen Oxides (NOx) Emissions [kg],";
556     ofs « "Sulfur Oxides (SOx) Emissions [kg],";
557     ofs « "Methane (CH4) Emissions [kg],";
558     ofs « "Particulate Matter (PM) Emissions [kg],";
559     ofs « "Capital Cost (actual),";
560     ofs « "Operation and Maintenance Cost (actual),";
561     ofs « "\n";
562
563     for (int i = 0; i < max_lines; i++) {
564         ofs « time_vec_hrs_ptr->at(i) « ",";
565         ofs « this->production_vec_kW[i] « ",";
566         ofs « this->dispatch_vec_kW[i] « ",";
567         ofs « this->storage_vec_kW[i] « ",";
568         ofs « this->curtailment_vec_kW[i] « ",";
```

```
569         ofs « this->is_running_vec[i] « ",";
570         ofs « this->fuel_consumption_vec_L[i] « ",";
571         ofs « this->fuel_cost_vec[i] « ",";
572         ofs « this->CO2_emissions_vec_kg[i] « ",";
573         ofs « this->CO_emissions_vec_kg[i] « ",";
574         ofs « this->NOx_emissions_vec_kg[i] « ",";
575         ofs « this->SOx_emissions_vec_kg[i] « ",";
576         ofs « this->CH4_emissions_vec_kg[i] « ",";
577         ofs « this->PM_emissions_vec_kg[i] « ",";
578         ofs « this->capital_cost_vec[i] « ",";
579         ofs « this->operation_maintenance_cost_vec[i] « ",";
580         ofs « "\n";
581     }
582
583     ofs.close();
584     return;
585 }   /* __writeTimeSeries() */
```

4.4.3.9 commit()

```
double Diesel::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Combustion.

```
827 {
828     //  1. handle start/stop, enforce minimum runtime constraint
829     this->__handleStartStop(timestep, dt_hrs, production_kW);
830
831     //  2. invoke base class method
832     load_kW = Combustion :: commit(
833         timestep,
834         dt_hrs,
835         production_kW,
836         load_kW
837     );
838
839     if (this->is_running) {
840         //  3. log time since last start
841         this->time_since_last_start_hrs += dt_hrs;
842
843         //  4. correct operation and maintenance costs (should be non-zero if idling)
844         if (production_kW <= 0) {
845             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
846
847             double operation_maintenance_cost =
848                 this->operation_maintenance_cost_kWh * produced_kWh;
849             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
850         }
```

```
851     }
852
853     return load_kW;
854 }   /* commit() */
```

4.4.3.10 handleReplacement()

```
void Diesel::handleReplacement (
             int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Combustion.

```
725 {
726     //  1. reset attributes
727     this->time_since_last_start_hrs = 0;
728
729     //  2. invoke base class method
730     Combustion :: handleReplacement(timestep);
731
732     return;
733 }   /* __handleReplacement() */
```

4.4.3.11 requestProductionkW()

```
double Diesel::requestProductionkW (
             int timestep,
             double dt_hrs,
             double request_kW )  [virtual]
```

Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
request_kW	The requested production [kW].

Returns

The production [kW] delivered by the diesel generator.

Reimplemented from Combustion.

```
765 {
766     //  0. given production time series override
767     if (this->normalized_production_series_given) {
768         double production_kW = Production :: getProductionkW(timestep);
```

```
769
770         return production_kW;
771     }
772
773     //  1. return on request of zero
774     if (request_kW <= 0) {
775         return 0;
776     }
777
778     double deliver_kW = request_kW;
779
780     //  2. enforce capacity constraint
781     if (deliver_kW > this->capacity_kW) {
782         deliver_kW = this->capacity_kW;
783     }
784
785     //  3. enforce minimum load ratio
786     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
787         deliver_kW = this->minimum_load_ratio * this->capacity_kW;
788     }
789
790     return deliver_kW;
791 }   /* requestProductionkW() */
```

4.4.4 Member Data Documentation

4.4.4.1 minimum_load_ratio

`double Diesel::minimum_load_ratio`

The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.

4.4.4.2 minimum_runtime_hrs

`double Diesel::minimum_runtime_hrs`

The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.

4.4.4.3 time_since_last_start_hrs

`double Diesel::time_since_last_start_hrs`

The time that has elapsed [hrs] since the last start of the asset.

The documentation for this class was generated from the following files:

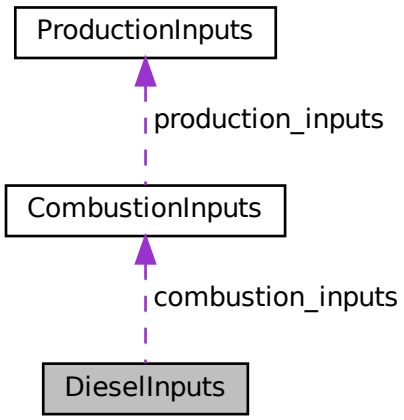
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

4.5 DieselInputs Struct Reference

A structure which bundles the necessary inputs for the Diesel constructor. Provides default values for every necessary input. Note that this structure encapsulates CombustionInputs.

`#include <Diesel.h>`

Collaboration diagram for DieselInputs:



Public Attributes

- CombustionInputs combustion_inputs

  An encapsulated CombustionInputs instance.
- double replace_running_hrs = 30000

  The number of running hours after which the asset must be replaced. Overwrites the ProductionInputs attribute.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double fuel_cost_L = 1.70

  The cost of fuel [1/L] (undefined currency).
- double minimum_load_ratio = 0.2

  The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.
- double minimum_runtime_hrs = 4

  The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.
- double linear_fuel_slope_LkWh = -1

The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced. -1 is a sentinel value, which triggers a generic fuel consumption model on construction (in fact, any negative value here will trigger).

- double linear_fuel_intercept_LkWh = -1

  The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced. -1 is a sentinel value, which triggers a generic fuel consumption model on construction (in fact, any negative value here will trigger).
- double CO2_emissions_intensity_kgL = 2.7

  Carbon dioxide (CO2) emissions intensity [kg/L].
- double CO_emissions_intensity_kgL = 0.0178

  Carbon monoxide (CO) emissions intensity [kg/L].
- double NOx_emissions_intensity_kgL = 0.0014

  Nitrogen oxide (NOx) emissions intensity [kg/L].
- double SOx_emissions_intensity_kgL = 0.0042

  Sulfur oxide (SOx) emissions intensity [kg/L].
- double CH4_emissions_intensity_kgL = 0.0007

  Methane (CH4) emissions intensity [kg/L].
- double PM_emissions_intensity_kgL = 0.0001

  Particulate Matter (PM) emissions intensity [kg/L].

4.5.1 Detailed Description

A structure which bundles the necessary inputs for the Diesel constructor. Provides default values for every necessary input. Note that this structure encapsulates CombustionInputs.

Ref: HOMER [2023c]
Ref: HOMER [2023d]
Ref: HOMER [2023e]
Ref: NRCan [2014]
Ref: CIMAC [2008]

4.5.2 Member Data Documentation

4.5.2.1 capital_cost

```
double DieselInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.5.2.2 CH4_emissions_intensity_kgL

```
double DieselInputs::CH4_emissions_intensity_kgL = 0.0007
```

Methane (CH4) emissions intensity [kg/L].

4.5.2.3 CO2_emissions_intensity_kgL

```
double DieselInputs::CO2_emissions_intensity_kgL = 2.7
```

Carbon dioxide (CO2) emissions intensity [kg/L].

4.5.2.4 CO_emissions_intensity_kgL

```
double DieselInputs::CO_emissions_intensity_kgL = 0.0178
```

Carbon monoxide (CO) emissions intensity [kg/L].

4.5.2.5 combustion_inputs

```
CombustionInputs DieselInputs::combustion_inputs
```

An encapsulated CombustionInputs instance.

4.5.2.6 fuel_cost_L

```
double DieselInputs::fuel_cost_L = 1.70
```

The cost of fuel [1/L] (undefined currency).

4.5.2.7 linear_fuel_intercept_LkWh

```
double DieselInputs::linear_fuel_intercept_LkWh = -1
```

The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced. -1 is a sentinel value, which triggers a generic fuel consumption model on construction (in fact, any negative value here will trigger).

4.5.2.8 linear_fuel_slope_LkWh

```
double DieselInputs::linear_fuel_slope_LkWh = -1
```

The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced. -1 is a sentinel value, which triggers a generic fuel consumption model on construction (in fact, any negative value here will trigger).

4.5.2.9 minimum_load_ratio

```
double DieselInputs::minimum_load_ratio = 0.2
```

The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.

4.5.2.10 minimum_runtime_hrs

```
double DieselInputs::minimum_runtime_hrs = 4
```

The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.

4.5.2.11 NOx_emissions_intensity_kgL

```
double DieselInputs::NOx_emissions_intensity_kgL = 0.0014
```

Nitrogen oxide (NOx) emissions intensity [kg/L].

4.5.2.12 operation_maintenance_cost_kWh

```
double DieselInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.5.2.13 PM_emissions_intensity_kgL

```
double DieselInputs::PM_emissions_intensity_kgL = 0.0001
```

Particulate Matter (PM) emissions intensity [kg/L].

4.5.2.14 replace_running_hrs

```
double DieselInputs::replace_running_hrs = 30000
```

The number of running hours after which the asset must be replaced. Overwrites the ProductionInputs attribute.

4.5.2.15 SOx_emissions_intensity_kgL

```
double DieselInputs::SOx_emissions_intensity_kgL = 0.0042
```

Sulfur oxide (SOx) emissions intensity [kg/L].

The documentation for this struct was generated from the following file:

- header/Production/Combustion/Diesel.h

4.6 ElectricalLoad Class Reference

A class which contains time and electrical load data. Intended to serve as a component class of Model.

```
#include <ElectricalLoad.h>
```

Public Member Functions

- ElectricalLoad (void)

  Constructor (dummy) for the ElectricalLoad class.
- ElectricalLoad (std::string)

  Constructor (intended) for the ElectricalLoad class.
- void readLoadData (std::string)

  Method to read electrical load data into an already existing ElectricalLoad object. Clears and overwrites any existing attribute values.
- void clear (void)

  Method to clear all attributes of the ElectricalLoad object.
- ~ElectricalLoad (void)

  Destructor for the ElectricalLoad class.

Public Attributes

- int n_points

  The number of points in the modelling time series.
- double n_years

  The number of years being modelled (inferred from time_vec_hrs).
- double min_load_kW

  The minimum [kW] of the given electrical load time series.
- double mean_load_kW

  The mean, or average, [kW] of the given electrical load time series.
- double max_load_kW

  The maximum [kW] of the given electrical load time series.
- std::string path_2_electrical_load_time_series

  A string defining the path (either relative or absolute) to the given electrical load time series.
- std::vector< double > time_vec_hrs

  A vector to hold a given sequence of model times [hrs]. This defines the modelling time series.
- std::vector< double > dt_vec_hrs

  A vector to hold a sequence of model time deltas [hrs].
- std::vector< double > load_vec_kW

  A vector to hold a given sequence of electrical load values [kW].

4.6.1 Detailed Description

A class which contains time and electrical load data. Intended to serve as a component class of Model.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 ElectricalLoad() [1/2]

```
ElectricalLoad::ElectricalLoad (
            void  )
```

Constructor (dummy) for the ElectricalLoad class.

```
37 {
38     return;
39 }   /* ElectricalLoad() */
```

4.6.2.2 ElectricalLoad() [2/2]

```
ElectricalLoad::ElectricalLoad (
            std::string path_2_electrical_load_time_series )
```

Constructor (intended) for the ElectricalLoad class.

Parameters

path_2_electrical_load_time_series	A string defining the path (either relative or absolute) to the given electrical load time series.

```
57 {
58     this->readLoadData(path_2_electrical_load_time_series);
59
60     return;
61 }   /* ElectricalLoad() */
```

4.6.2.3 ~ElectricalLoad()

```
ElectricalLoad::~ElectricalLoad (
            void  )
```

Destructor for the ElectricalLoad class.

```
184 {
185     this->clear();
186     return;
187 }   /* ~ElectricalLoad() */
```

4.6.3 Member Function Documentation

4.6.3.1 clear()

```
void ElectricalLoad::clear (
            void  )
```

Method to clear all attributes of the ElectricalLoad object.

```
157 {
158     this->n_points = 0;
159     this->n_years = 0;
160     this->min_load_kW = 0;
161     this->mean_load_kW = 0;
162     this->max_load_kW = 0;
163 
164     this->path_2_electrical_load_time_series.clear();
165     this->time_vec_hrs.clear();
166     this->dt_vec_hrs.clear();
167     this->load_vec_kW.clear();
168 
169     return;
170 }   /* clear() */
```

4.6.3.2 readLoadData()

```
void ElectricalLoad::readLoadData (
            std::string path_2_electrical_load_time_series )
```

Method to read electrical load data into an already existing ElectricalLoad object. Clears and overwrites any existing attribute values.

Parameters

path_2_electrical_load_time_series	A string defining the path (either relative or absolute) to the given electrical load time series.

```
79 {
80     //  1. clear
81     this->clear();
82 
83     //  2. init CSV reader, record path
84     io::CSVReader<2> CSV(path_2_electrical_load_time_series);
85 
86     CSV.read_header(
87         io::ignore_extra_column,
88         "Time (since start of data) [hrs]",
89         "Electrical Load [kW]"
90     );
91 
92     this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;
93 
94     //  3. read in time and load data, increment n_points, track min and max load
95     double time_hrs = 0;
96     double load_kW = 0;
97     double load_sum_kW = 0;
98 
99     this->n_points = 0;
100 
101     this->min_load_kW = std::numeric_limits<double>::infinity();
102     this->max_load_kW = -1 * std::numeric_limits<double>::infinity();
103 
104     while (CSV.read_row(time_hrs, load_kW)) {
105         this->time_vec_hrs.push_back(time_hrs);
106         this->load_vec_kW.push_back(load_kW);
107 
108         load_sum_kW += load_kW;
109 
110         this->n_points++;
111 
112         if (this->min_load_kW > load_kW) {
113             this->min_load_kW = load_kW;
114         }
115 
```

```
116         if (this->max_load_kW < load_kW) {
117             this->max_load_kW = load_kW;
118         }
119     }
120
121     //  4. compute mean load
122     this->mean_load_kW = load_sum_kW / this->n_points;
123
124     //  5. set number of years (assuming 8,760 hours per year)
125     this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;
126
127     //  6. populate dt_vec_hrs
128     this->dt_vec_hrs.resize(n_points, 0);
129
130     for (int i = 0; i < n_points; i++) {
131         if (i == n_points - 1) {
132             this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
133         }
134
135         else {
136             double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];
137
138             this->dt_vec_hrs[i] = dt_hrs;
139         }
140     }
141
142     return;
143 }   /* readLoadData() */
```

4.6.4 Member Data Documentation

4.6.4.1 dt_vec_hrs

`std::vector<double> ElectricalLoad::dt_vec_hrs`

A vector to hold a sequence of model time deltas [hrs].

4.6.4.2 load_vec_kW

`std::vector<double> ElectricalLoad::load_vec_kW`

A vector to hold a given sequence of electrical load values [kW].

4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

The maximum [kW] of the given electrical load time series.

4.6.4.4 mean_load_kW

```
double ElectricalLoad::mean_load_kW
```

The mean, or average, [kW] of the given electrical load time series.

4.6.4.5 min_load_kW

```
double ElectricalLoad::min_load_kW
```

The minimum [kW] of the given electrical load time series.

4.6.4.6 n_points

```
int ElectricalLoad::n_points
```

The number of points in the modelling time series.

4.6.4.7 n_years

```
double ElectricalLoad::n_years
```

The number of years being modelled (inferred from time_vec_hrs).

4.6.4.8 path_2_electrical_load_time_series

```
std::string ElectricalLoad::path_2_electrical_load_time_series
```

A string defining the path (either relative or absolute) to the given electrical load time series.

4.6.4.9 time_vec_hrs

```
std::vector<double> ElectricalLoad::time_vec_hrs
```

A vector to hold a given sequence of model times [hrs]. This defines the modelling time series.

The documentation for this class was generated from the following files:

- header/ElectricalLoad.h
- source/ElectricalLoad.cpp

4.7 Emissions Struct Reference

A structure which bundles the emitted masses of various emissions chemistries.

`#include <Combustion.h>`

Public Attributes

- double CO2_kg = 0
  The mass of carbon dioxide (CO2) emitted [kg].
- double CO_kg = 0
  The mass of carbon monoxide (CO) emitted [kg].
- double NOx_kg = 0
  The mass of nitrogen oxides (NOx) emitted [kg].
- double SOx_kg = 0
  The mass of sulfur oxides (SOx) emitted [kg].
- double CH4_kg = 0
  The mass of methane (CH4) emitted [kg].
- double PM_kg = 0
  The mass of particulate matter (PM) emitted [kg].

4.7.1 Detailed Description

A structure which bundles the emitted masses of various emissions chemistries.

4.7.2 Member Data Documentation

4.7.2.1 CH4_kg

`double Emissions::CH4_kg = 0`

The mass of methane (CH4) emitted [kg].

4.7.2.2 CO2_kg

`double Emissions::CO2_kg = 0`

The mass of carbon dioxide (CO2) emitted [kg].

4.7.2.3 CO_kg

```
double Emissions::CO_kg = 0
```

The mass of carbon monoxide (CO) emitted [kg].

4.7.2.4 NOx_kg

```
double Emissions::NOx_kg = 0
```

The mass of nitrogen oxides (NOx) emitted [kg].

4.7.2.5 PM_kg

```
double Emissions::PM_kg = 0
```

The mass of particulate matter (PM) emitted [kg].

4.7.2.6 SOx_kg

```
double Emissions::SOx_kg = 0
```

The mass of sulfur oxides (SOx) emitted [kg].

The documentation for this struct was generated from the following file:

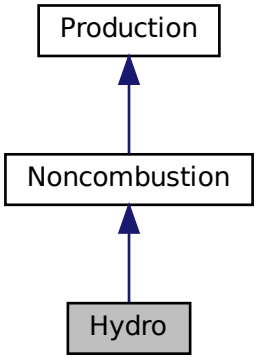
- header/Production/Combustion/Combustion.h

4.8 Hydro Class Reference

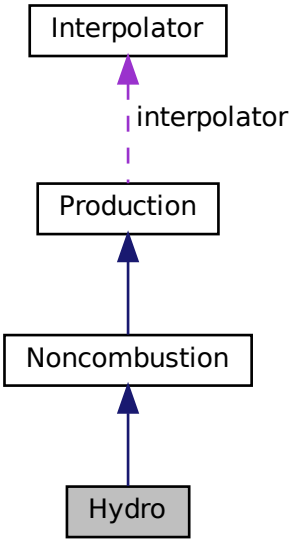
A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not).

`#include <Hydro.h>`

Inheritance diagram for Hydro:



Collaboration diagram for Hydro:



Public Member Functions

- Hydro (void)

  Constructor (dummy) for the Hydro class.
- Hydro (int, double, HydroInputs, std::vector< double > *)

  Constructor (intended) for the Hydro class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double requestProductionkW (int, double, double, double)

  Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).
- double commit (int, double, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Hydro (void)

  Destructor for the Hydro class.

Public Attributes

- HydroTurbineType turbine_type

  The type of hydroelectric turbine model to use.
- double fluid_density_kgm3

  The density [kg/m3] of the hydroelectric working fluid.
- double net_head_m

  The net head [m] of the asset.
- double reservoir_capacity_m3

  The capacity [m3] of the hydro reservoir.
- double init_reservoir_state

  The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- double stored_volume_m3

  The volume [m3] of stored fluid.
- double minimum_power_kW

  The minimum power [kW] that the asset can produce. Corresponds to minimum productive flow.
- double minimum_flow_m3hr

  The minimum required flow [m3/hr] for the asset to produce. Corresponds to minimum power.
- double maximum_flow_m3hr

  The maximum productive flow [m3/hr] that the asset can support.
- std::vector< double > turbine_flow_vec_m3hr

  A vector of the turbine flow [m3/hr] at each point in the modelling time series.
- std::vector< double > spill_rate_vec_m3hr

  A vector of the spill rate [m3/hr] at each point in the modelling time series.
- std::vector< double > stored_volume_vec_m3

  A vector of the stored volume [m3] in the reservoir at each point in the modelling time series.

Private Member Functions

- void __checkInputs (HydroInputs)

  Helper method to check inputs to the Hydro constructor.
- void __initInterpolator (void)

  Helper method to set up turbine and generator efficiency interpolation.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic hydroelectric capital cost.
- double __getGenericOpMaintCost (void)

  Helper method (private) to generate a generic hydroelectric operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __getEfficiencyFactor (double)

  Helper method to compute the efficiency factor (product of turbine and generator efficiencies).
- double __getMinimumFlowm3hr (void)

  Helper method to compute and return the minimum required flow for production, based on turbine type.
- double __getMaximumFlowm3hr (void)

  Helper method to compute and return the maximum productive flow, based on turbine type.
- double __flowToPower (double)

  Helper method to translate a given flow into a corresponding power output.
- double __powerToFlow (double)

  Helper method to translate a given power output into a corresponding flow.
- double __getAvailableFlow (double, double)

  Helper method to determine what flow is currently available to the turbine.
- double __getAcceptableFlow (double)

  Helper method to determine what flow is currently acceptable by the reservoir.
- void __updateState (int, double, double, double)

  Helper method to update and log flow and reservoir state.
- void __writeSummary (std::string)

  Helper method to write summary results for Hydro.
- void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

  Helper method to write time series results for Hydro.

4.8.1 Detailed Description

A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not).

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Hydro() [1/2]

```
Hydro::Hydro (
            void  )
```

Constructor (dummy) for the Hydro class.

```
834 {
835     return;
836 }   /* Hydro() */
```

4.8.2.2 Hydro() [2/2]

```
Hydro::Hydro (
            int n_points,
            double n_years,
            HydroInputs hydro_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Hydro class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
hydro_inputs	A structure of Hydro constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
868   :
869 Noncombustion(
870     n_points,
871     n_years,
872     hydro_inputs.noncombustion_inputs,
873     time_vec_hrs_ptr
874 )
875 {
876     //  1. check inputs
877     this->__checkInputs(hydro_inputs);
878 
879     //  2. set attributes
880     this->type = NoncombustionType :: HYDRO;
881     this->type_str = "HYDRO";
882 
883     this->resource_key = hydro_inputs.resource_key;
884 
885     this->turbine_type = hydro_inputs.turbine_type;
886 
887     this->fluid_density_kgm3 = hydro_inputs.fluid_density_kgm3;
888     this->net_head_m = hydro_inputs.net_head_m;
889 
890     this->reservoir_capacity_m3 = hydro_inputs.reservoir_capacity_m3;
891     this->init_reservoir_state = hydro_inputs.init_reservoir_state;
892     this->stored_volume_m3 =
893         hydro_inputs.init_reservoir_state * hydro_inputs.reservoir_capacity_m3;
894 
895     this->minimum_power_kW = 0.1 * this->capacity_kW;   // <-- NEED TO DOUBLE CHECK THAT THIS MAKES
      SENSE IN GENERAL
896 
897     this->__initInterpolator();
898 
899     this->minimum_flow_m3hr = this->__getMinimumFlowm3hr();
900     this->maximum_flow_m3hr = this->__getMaximumFlowm3hr();
901 
902     this->turbine_flow_vec_m3hr.resize(this->n_points, 0);
903     this->spill_rate_vec_m3hr.resize(this->n_points, 0);
904     this->stored_volume_vec_m3.resize(this->n_points, 0);
905 
906     if (hydro_inputs.capital_cost < 0) {
907         this->capital_cost = this->__getGenericCapitalCost();
908     }
909     else {
910         this->capital_cost = hydro_inputs.capital_cost;
911     }
912 
913     if (hydro_inputs.operation_maintenance_cost_kWh < 0) {
914         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
915     }
916     else {
917         this->operation_maintenance_cost_kWh =
918             hydro_inputs.operation_maintenance_cost_kWh;
919     }
920 
921     if (not this->is_sunk) {
922         this->capital_cost_vec[0] = this->capital_cost;
923     }
924 
925     return;
926 }   /* Hydro() */
```

4.8.2.3 ~Hydro()

```
Hydro::~Hydro (
            void  )
```

Destructor for the Hydro class.

```
1100 {
1101     //  1. destruction print
1102     if (this->print_flag) {
1103         std::cout « "Hydro object at " « this « " destroyed" « std::endl;
1104     }
1105
1106     return;
1107 }   /* ~Hydro() */
```

4.8.3 Member Function Documentation

4.8.3.1 __checkInputs()

```
void Hydro::__checkInputs (
            HydroInputs hydro_inputs )  [private]
```

Helper method to check inputs to the Hydro constructor.

Parameters

hydro_inputs	A structure of Hydro constructor inputs.

```
39 {
40     //  1. check fluid_density_kgm3
41     if (hydro_inputs.fluid_density_kgm3 <= 0) {
42         std::string error_str = "ERROR:  Hydro():  fluid_density_kgm3 must be > 0";
43
44         #ifdef _WIN32
45             std::cout « error_str « std::endl;
46         #endif
47
48         throw std::invalid_argument(error_str);
49     }
50
51     //  2. check net_head_m
52     if (hydro_inputs.net_head_m <= 0) {
53         std::string error_str = "ERROR:  Hydro():  net_head_m must be > 0";
54
55         #ifdef _WIN32
56             std::cout « error_str « std::endl;
57         #endif
58
59         throw std::invalid_argument(error_str);
60     }
61
62     //  3. check reservoir_capacity_m3
63     if (hydro_inputs.reservoir_capacity_m3 < 0) {
64         std::string error_str = "ERROR:  Hydro():  reservoir_capacity_m3 must be >= 0";
65
66         #ifdef _WIN32
67             std::cout « error_str « std::endl;
68         #endif
69
70         throw std::invalid_argument(error_str);
71     }
72
```

```
73      //  4. check init_reservoir_state
74      if (
75          hydro_inputs.init_reservoir_state < 0 or
76          hydro_inputs.init_reservoir_state > 1
77      ) {
78          std::string error_str = "ERROR:  Hydro():  init_reservoir_state must be in ";
79          error_str += "the closed interval [0, 1]";
80
81          #ifdef _WIN32
82              std::cout « error_str « std::endl;
83          #endif
84
85          throw std::invalid_argument(error_str);
86      }
87
88      return;
89  }   /* __checkInputs() */
```

4.8.3.2 __flowToPower()

```
double Hydro::__flowToPower (
            double flow_m3hr )  [private]
```

Helper method to translate a given flow into a corresponding power output.

Ref: Truelove [2023b]

Parameters

flow_m3hr	The flow [m3/hr] through the turbine.

Returns

The power output [kW] corresponding to a given flow [m3/hr].

```
427 {
428      //  1. return on less than minimum flow
429      if (flow_m3hr < this->minimum_flow_m3hr) {
430          return 0;
431      }
432
433      //  2. interpolate flow to power
434      double power_kW = this->interpolator.interp1D(
435          HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
436          flow_m3hr
437      );
438
439      return power_kW;
440  }   /* __flowToPower() */
```

4.8.3.3 __getAcceptableFlow()

```
double Hydro::__getAcceptableFlow (
            double dt_hrs )  [private]
```

Helper method to determine what flow is currently acceptable by the reservoir.

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

The flow [m3/hr] currently acceptable by the reservoir.

```
529 {
530     //  1. if no reservoir, return
531     if (this->reservoir_capacity_m3 <= 0) {
532         return 0;
533     }
534 
535     //  2. compute acceptable based on room in reservoir
536     double acceptable_m3hr = (this->reservoir_capacity_m3 - this->stored_volume_m3) /
537         dt_hrs;
538 
539     return acceptable_m3hr;
540 }   /* __getAcceptableFlow() */
```

4.8.3.4 __getAvailableFlow()

```
double Hydro::__getAvailableFlow (
            double dt_hrs,
            double hydro_resource_m3hr )  [private]
```

Helper method to determine what flow is currently available to the turbine.

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.
hydro_resource_m3hr	The currently available hydro flow resource [m3/hr].

Returns

The flow [m3/hr] currently available through the turbine.

```
496 {
497     //  1. init to flow available from stored volume in reservoir
498     double flow_m3hr = this->stored_volume_m3 / dt_hrs;
499 
500     //  2. add flow available from resource
501     flow_m3hr += hydro_resource_m3hr;
502 
503     //  3. cap at maximum flow
504     if (flow_m3hr > this->maximum_flow_m3hr) {
505         flow_m3hr = this->maximum_flow_m3hr;
506     }
507 
508     return flow_m3hr;
509 }   /* __getAvailableFlow() */
```

4.8.3.5 __getEfficiencyFactor()

```
double Hydro::__getEfficiencyFactor (
            double power_kW )  [private]
```

Helper method to compute the efficiency factor (product of turbine and generator efficiencies).

Ref: Truelove [2023b]

Parameters

power_kW	The power requested of the hydro plant.

Returns

The product of the turbine and generator efficiencies.

```
325 {
326     //  1. return on zero
327     if (power_kW <= 0) {
328         return 0;
329     }
330 
331     //  2. compute power ratio (clip to [0, 1])
332     double power_ratio = power_kW / this->capacity_kW;
333 
334     if (power_ratio < 0) {
335         power_ratio = 0;
336     }
337 
338     else if (power_ratio > 1) {
339         power_ratio = 1;
340     }
341 
342 
343     //  3. init efficiency factor to the turbine efficiency
344     double efficiency_factor = this->interpolator.interp1D(
345         HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
346         power_ratio
347     );
348 
349     //  4. include generator efficiency
350     efficiency_factor *= this->interpolator.interp1D(
351         HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
352         power_ratio
353     );
354 
355     return efficiency_factor;
356 }   /* __getEfficiencyFactor() */
```

4.8.3.6 __getGenericCapitalCost()

```
double Hydro::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic hydroelectric capital cost.

This model was obtained by way of ...

Returns

A generic capital cost for the hydroelectric asset [CAD].

```
274 {
275     double capital_cost_per_kW = 1000; //<-- WIP: need something better here!
276 
277     return capital_cost_per_kW * this->capacity_kW + 15000000; //<-- WIP: need something better here!
278 }   /* __getGenericCapitalCost() */
```

4.8.3.7 __getGenericOpMaintCost()

```
double Hydro::__getGenericOpMaintCost (
            void ) [private]
```

Helper method (private) to generate a generic hydroelectric operation and maintenance cost. This is a cost incurred per unit energy produced.

This model was obtained by way of ...

Returns

A generic operation and maintenance cost, per unit energy produced, for the hydroelectric asset [CAD/kWh].

```
299 {
300     double operation_maintenance_cost_kWh = 0.05;  //<-- WIP: need something better here!
301
302     return operation_maintenance_cost_kWh;
303 }   /* __getGenericOpMaintCost() */
```

4.8.3.8 __getMaximumFlowm3hr()

```
double Hydro::__getMaximumFlowm3hr (
            void ) [private]
```

Helper method to compute and return the maximum productive flow, based on turbine type.

This helper method assumes that the maximum flow is that which is associated with a power ratio of 1.

Ref: Truelove [2023b]

Returns

The maximum productive flow [m3/hr].

```
404 {
405     return this->__powerToFlow(this->capacity_kW);
406 }   /* __getMaximumFlowm3hr() */
```

4.8.3.9 __getMinimumFlowm3hr()

```
double Hydro::__getMinimumFlowm3hr (
            void ) [private]
```

Helper method to compute and return the minimum required flow for production, based on turbine type.

This helper method assumes that the minimum flow is that which is associated with a power ratio of 0.1. See constructor for initialization of minimum_power_kW.

Ref: Truelove [2023b]

Returns

The minimum required flow [m3/hr] for production.

```
379 {
380     return this->__powerToFlow(this->minimum_power_kW);
381 }   /* __getMinimumFlowm3hr() */
```

4.8.3.10 __initInterpolator()

```
void Hydro::__initInterpolator (
            void  )  [private]
```

Helper method to set up turbine and generator efficiency interpolation.

Ref: Truelove [2023b]

```
106 {
107     //  1. set up generator efficiency interpolation
108     InterpolatorStruct1D generator_interp_struct_1D;
109 
110     generator_interp_struct_1D.n_points = 12;
111 
112     generator_interp_struct_1D.x_vec = {
113         0,   0.1, 0.2,  0.3, 0.4, 0.5,
114         0.6, 0.7, 0.75, 0.8, 0.9, 1
115     };
116 
117     generator_interp_struct_1D.min_x = 0;
118     generator_interp_struct_1D.max_x = 1;
119 
120     generator_interp_struct_1D.y_vec = {
121         0.000, 0.800, 0.900, 0.913,
122         0.925, 0.943, 0.947, 0.950,
123         0.953, 0.954, 0.956, 0.958
124     };
125 
126     this->interpolator.interp_map_1D.insert(
127         std::pair<int, InterpolatorStruct1D>(
128             HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
129             generator_interp_struct_1D
130         )
131     );
132 
133     //  2. set up turbine efficiency interpolation
134     InterpolatorStruct1D turbine_interp_struct_1D;
135 
136     turbine_interp_struct_1D.n_points = 11;
137 
138     turbine_interp_struct_1D.x_vec = {
139         0,   0.1, 0.2, 0.3, 0.4,
140         0.5, 0.6, 0.7, 0.8, 0.9,
141         1
142     };
143 
144     turbine_interp_struct_1D.min_x = 0;
145     turbine_interp_struct_1D.max_x = 1;
146 
147     std::vector<double> efficiency_vec;
148 
149     switch (this->turbine_type) {
150         case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
151             efficiency_vec = {
152                 0.000, 0.780, 0.855, 0.875, 0.890,
153                 0.900, 0.908, 0.913, 0.918, 0.908,
154                 0.880
155             };
156 
157             break;
158         }
159 
160         case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
161             efficiency_vec = {
162                 0.000, 0.400, 0.625, 0.745, 0.810,
163                 0.845, 0.880, 0.900, 0.910, 0.900,
164                 0.850
165             };
166 
167             break;
168         }
169 
170         case(HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
171             efficiency_vec = {
172                 0.000, 0.265, 0.460, 0.550, 0.650,
173                 0.740, 0.805, 0.845, 0.900, 0.880,
174                 0.850
175             };
176 
177             break;
178         }
```

```
179
180         default: {
181             std::string error_str = "ERROR:  Hydro():  turbine type ";
182             error_str += std::to_string(this->turbine_type);
183             error_str += " not recognized";
184
185             #ifdef _WIN32
186                 std::cout « error_str « std::endl;
187             #endif
188
189             throw std::runtime_error(error_str);
190
191             break;
192         }
193     }
194
195     turbine_interp_struct_1D.y_vec = efficiency_vec;
196
197     this->interpolator.interp_map_1D.insert(
198         std::pair<int, InterpolatorStruct1D>(
199             HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
200             turbine_interp_struct_1D
201         )
202     );
203
204     //  3. set up flow to power interpolation
205     InterpolatorStruct1D flow_to_power_interp_struct_1D;
206
207     double power_ratio = 0.1;
208     std::vector<double> power_ratio_vec (91, 0);
209
210     for (size_t i = 0; i < power_ratio_vec.size(); i++) {
211         power_ratio_vec[i] = power_ratio;
212
213         power_ratio += 0.01;
214
215         if (power_ratio < 0) {
216             power_ratio = 0;
217         }
218
219         else if (power_ratio > 1) {
220             power_ratio = 1;
221         }
222     }
223
224     flow_to_power_interp_struct_1D.n_points = power_ratio_vec.size();
225
226     std::vector<double> flow_vec_m3hr;
227     std::vector<double> power_vec_kW;
228     flow_vec_m3hr.resize(power_ratio_vec.size(), 0);
229     power_vec_kW.resize(power_ratio_vec.size(), 0);
230
231     for (size_t i = 0; i < power_ratio_vec.size(); i++) {
232         flow_vec_m3hr[i] = this->__powerToFlow(power_ratio_vec[i] * this->capacity_kW);
233         power_vec_kW[i] = power_ratio_vec[i] * this->capacity_kW;
234         /*
235         std::cout « flow_vec_m3hr[i] « "\t" « power_vec_kW[i] « " (" «
236             power_ratio_vec[i] « ")" « std::endl;
237         */
238     }
239
240     flow_to_power_interp_struct_1D.x_vec = flow_vec_m3hr;
241
242     flow_to_power_interp_struct_1D.min_x = flow_vec_m3hr[0];
243     flow_to_power_interp_struct_1D.max_x = flow_vec_m3hr[flow_vec_m3hr.size() - 1];
244
245     flow_to_power_interp_struct_1D.y_vec = power_vec_kW;
246
247     this->interpolator.interp_map_1D.insert(
248         std::pair<int, InterpolatorStruct1D>(
249             HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
250             flow_to_power_interp_struct_1D
251         )
252     );
253
254     return;
255 }   /* __initInterpolator() */
```

4.8.3.11 __powerToFlow()

```
double Hydro::__powerToFlow (
            double power_kW )  [private]
```

Helper method to translate a given power output into a corresponding flow.

Ref: Truelove [2023b]

Parameters

power_kW	The power output [kW] of the hydroelectric generator.

Returns

```
461 {
462     //  1. return on zero power
463     if (power_kW <= 0) {
464         return 0;
465     }
466 
467     //  2. get efficiency factor
468     double efficiency_factor = this->__getEfficiencyFactor(power_kW);
469 
470     //  3. compute flow
471     double flow_m3hr = 3600 * 1000 * power_kW;
472     flow_m3hr /= efficiency_factor * this->fluid_density_kgm3 * 9.81 * this->net_head_m;
473 
474     return flow_m3hr;
475 }   /* __powerToFlow() */
```

4.8.3.12 __updateState()

```
void Hydro::__updateState (
             int timestep,
             double dt_hrs,
             double production_kW,
             double hydro_resource_m3hr )  [private]
```

Helper method to update and log flow and reservoir state.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
hydro_resource_m3hr	The currently available hydro flow resource [m3/hr].

```
573 {
574     //  1. get turbine flow, log
575     double flow_m3hr = 0;
576 
577     if (production_kW >= this->minimum_power_kW) {
578         flow_m3hr = this->__powerToFlow(production_kW);
579     }
580 
581     double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);
582 
583     if (flow_m3hr > available_flow_m3hr) {
584         flow_m3hr = available_flow_m3hr;
585     }
586 
587     this->turbine_flow_vec_m3hr[timestep] = flow_m3hr;
588 
589     //  3. compute net reservoir flow
```

```
590     double net_flow_m3hr = hydro_resource_m3hr - flow_m3hr;
591
592     //  4. compute flow acceptable by reservoir
593     double acceptable_flow_m3hr = this->__getAcceptableFlow(dt_hrs);
594
595     //  5. compute spill, update net flow (if applicable), log
596     double spill_m3hr = 0;
597
598     if (acceptable_flow_m3hr < net_flow_m3hr) {
599         spill_m3hr = net_flow_m3hr - acceptable_flow_m3hr;
600         net_flow_m3hr = acceptable_flow_m3hr;
601     }
602
603     this->spill_rate_vec_m3hr[timestep] = spill_m3hr;
604
605     //  6. update reservoir state, log
606     this->stored_volume_m3 += net_flow_m3hr * dt_hrs;
607     this->stored_volume_vec_m3[timestep] = this->stored_volume_m3;
608
609     return;
610 }   /* __updateState() */
```

4.8.3.13 __writeSummary()

```
void Hydro::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Hydro.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Noncombustion.

```
628 {
629     //  1. create filestream
630     write_path += "summary_results.md";
631     std::ofstream ofs;
632     ofs.open(write_path, std::ofstream::out);
633
634     //  2. write to summary results (markdown)
635     ofs « "# ";
636     ofs « std::to_string(int(ceil(this->capacity_kW)));
637     ofs « " kW HYDRO Summary Results\n";
638     ofs « "\n--------\n\n";
639
640     //  2.1. Production attributes
641     ofs « "## Production Attributes\n";
642     ofs « "\n";
643
644     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
645     ofs « "\n";
646
647     ofs « "Production Override: (N = 0 / Y = 1): "
648         « this->normalized_production_series_given « "  \n";
649     if (this->normalized_production_series_given) {
650         ofs « "Path to Normalized Production Time Series: "
651             « this->path_2_normalized_production_time_series « "  \n";
652     }
653     ofs « "\n";
654
655     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
656     ofs « "Capital Cost: " « this->capital_cost « "  \n";
657     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
658         « " per kWh produced  \n";
659     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
660         « "  \n";
661     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
662         « "  \n";
663     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
664     ofs « "\n";
```

```
665
666     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
667     ofs « "\n--------\n\n";
668
669     //  2.2. Noncombustion attributes
670     ofs « "## Noncombustion Attributes\n";
671     ofs « "\n";
672
673     //...
674
675     ofs « "\n--------\n\n";
676
677     //  2.3. Hydro attributes
678     ofs « "## Hydro Attributes\n";
679     ofs « "\n";
680
681     ofs « "Fluid Density: " « this->fluid_density_kgm3 « " kg/m3  \n";
682     ofs « "Net Head: " « this->net_head_m « " m  \n";
683     ofs « "\n";
684
685     ofs « "Reservoir Volume: " « this->reservoir_capacity_m3 « " m3  \n";
686     ofs « "Reservoir Initial State: " « this->init_reservoir_state « "  \n";
687     ofs « "\n";
688
689     ofs « "Turbine Type: ";
690     switch(this->turbine_type) {
691         case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
692             ofs « "PELTON";
693
694             break;
695         }
696
697         case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
698             ofs « "FRANCIS";
699
700             break;
701         }
702
703         case(HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
704             ofs « "KAPLAN";
705
706             break;
707         }
708
709         default: {
710             // write nothing!
711
712             break;
713         }
714     }
715     ofs « "  \n";
716     ofs « "\n";
717     ofs « "Minimum Flow: " « this->minimum_flow_m3hr « " m3/hr  \n";
718     ofs « "Maximum Flow: " « this->maximum_flow_m3hr « " m3/hr  \n";
719     ofs « "\n";
720     ofs « "Minimum Production: " « this->minimum_power_kW « " kW  \n";
721     ofs « "\n";
722
723     ofs « "\n--------\n\n";
724
725     //  2.4. Hydro Results
726     ofs « "## Results\n";
727     ofs « "\n";
728
729     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
730     ofs « "\n";
731
732     ofs « "Total Dispatch: " « this->total_dispatch_kWh
733         « " kWh  \n";
734
735     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
736         « " per kWh dispatched  \n";
737     ofs « "\n";
738
739     ofs « "Running Hours: " « this->running_hours « "  \n";
740     ofs « "Replacements: " « this->n_replacements « "  \n";
741
742     //...
743
744     ofs « "\n--------\n\n";
745
746     ofs.close();
747     return;
748 }   /* __writeSummary() */
```

4.8.3.14 __writeTimeSeries()

```
void Hydro::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Hydro.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
max_lines	The maximum number of lines of output to write.

Reimplemented from Noncombustion.

```
778 {
779     //  1. create filestream
780     write_path += "time_series_results.csv";
781     std::ofstream ofs;
782     ofs.open(write_path, std::ofstream::out);
783 
784     //  2. write time series results (comma separated value)
785     ofs « "Time (since start of data) [hrs],";
786     ofs « "Production [kW],";
787     ofs « "Dispatch [kW],";
788     ofs « "Storage [kW],";
789     ofs « "Curtailment [kW],";
790     ofs « "Is Running (N = 0 / Y = 1),";
791     ofs « "Turbine Flow [m3/hr],";
792     ofs « "Spill Rate [m3/hr],";
793     ofs « "Stored Volume [m3],";
794     ofs « "Capital Cost (actual),";
795     ofs « "Operation and Maintenance Cost (actual),";
796     ofs « "\n";
797 
798     for (int i = 0; i < max_lines; i++) {
799         ofs « time_vec_hrs_ptr->at(i) « ",";
800         ofs « this->production_vec_kW[i] « ",";
801         ofs « this->dispatch_vec_kW[i] « ",";
802         ofs « this->storage_vec_kW[i] « ",";
803         ofs « this->curtailment_vec_kW[i] « ",";
804         ofs « this->is_running_vec[i] « ",";
805         ofs « this->turbine_flow_vec_m3hr[i] « ",";
806         ofs « this->spill_rate_vec_m3hr[i] « ",";
807         ofs « this->stored_volume_vec_m3[i] « ",";
808         ofs « this->capital_cost_vec[i] « ",";
809         ofs « this->operation_maintenance_cost_vec[i] « ",";
810         ofs « "\n";
811     }
812 
813     ofs.close();
814     return;
815 }   /* __writeTimeSeries() */
```

4.8.3.15 commit()

```
double Hydro::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW,
            double hydro_resource_m3hr )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Noncombustion.

```
1067 {
1068     //  1. invoke base class method
1069     load_kW = Noncombustion :: commit(
1070         timestep,
1071         dt_hrs,
1072         production_kW,
1073         load_kW
1074     );
1075 
1076     //  2. update state and record
1077     this->__updateState(
1078         timestep,
1079         dt_hrs,
1080         production_kW,
1081         hydro_resource_m3hr
1082     );
1083 
1084     return load_kW;
1085 }   /* commit() */
```

4.8.3.16 handleReplacement()

```
void Hydro::handleReplacement (
             int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Noncombustion.

```
944 {
945     //  1. reset attributes
946     //...
947 
948     //  2. invoke base class method
949     Noncombustion :: handleReplacement(timestep);
950 
951     return;
952 }   /* __handleReplacement() */
```

4.8.3.17 requestProductionkW()

```
double Hydro::requestProductionkW (
             int timestep,
```

```
        double dt_hrs,
        double request_kW,
        double hydro_resource_m3hr )  [virtual]
```

Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
request_kW	The requested production [kW].
hydro_resource_m3hr	The currently available hydro flow resource [m3/hr].

Returns

The production [kW] delivered by the hydro generator.

Reimplemented from Noncombustion.

```
988 {
989     //  0. given production time series override
990     if (this->normalized_production_series_given) {
991         double production_kW = Production :: getProductionkW(timestep);
992 
993         return production_kW;
994     }
995 
996     //  1. return on request of zero
997     if (request_kW <= 0) {
998         return 0;
999     }
1000 
1001     //  2. if request is less than minimum power, set to minimum power
1002     if (request_kW < this->minimum_power_kW) {
1003         request_kW = this->minimum_power_kW;
1004     }
1005 
1006     //  3. check available flow, return if less than minimum flow
1007     double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);
1008 
1009     if (available_flow_m3hr < this->minimum_flow_m3hr) {
1010         return 0;
1011     }
1012 
1013     //  4. init production to request, enforce capacity constraint (which also accounts
1014     //     for maximum flow constraint).
1015     double production_kW = request_kW;
1016 
1017     if (production_kW > this->capacity_kW) {
1018         production_kW = this->capacity_kW;
1019     }
1020 
1021     //  5. map production to flow
1022     double flow_m3hr = this->__powerToFlow(production_kW);
1023 
1024     //  6. if flow is in excess of available, then adjust production accordingly
1025     if (flow_m3hr > available_flow_m3hr) {
1026         production_kW = this->__flowToPower(available_flow_m3hr);
1027     }
1028 
1029     return production_kW;
1030 }   /* requestProductionkW() */
```

4.8.4 Member Data Documentation

4.8.4.1 fluid_density_kgm3

```
double Hydro::fluid_density_kgm3
```

The density [kg/m3] of the hydroelectric working fluid.

4.8.4.2 init_reservoir_state

```
double Hydro::init_reservoir_state
```

The initial state of the reservoir (where state is volume of stored fluid divided by capacity).

4.8.4.3 maximum_flow_m3hr

```
double Hydro::maximum_flow_m3hr
```

The maximum productive flow [m3/hr] that the asset can support.

4.8.4.4 minimum_flow_m3hr

```
double Hydro::minimum_flow_m3hr
```

The minimum required flow [m3/hr] for the asset to produce. Corresponds to minimum power.

4.8.4.5 minimum_power_kW

```
double Hydro::minimum_power_kW
```

The minimum power [kW] that the asset can produce. Corresponds to minimum productive flow.

4.8.4.6 net_head_m

```
double Hydro::net_head_m
```

The net head [m] of the asset.

4.8.4.7 reservoir_capacity_m3

```
double Hydro::reservoir_capacity_m3
```

The capacity [m3] of the hydro reservoir.

4.8.4.8 spill_rate_vec_m3hr

```
std::vector<double> Hydro::spill_rate_vec_m3hr
```

A vector of the spill rate [m3/hr] at each point in the modelling time series.

4.8.4.9 stored_volume_m3

```
double Hydro::stored_volume_m3
```

The volume [m3] of stored fluid.

4.8.4.10 stored_volume_vec_m3

```
std::vector<double> Hydro::stored_volume_vec_m3
```

A vector of the stored volume [m3] in the reservoir at each point in the modelling time series.

4.8.4.11 turbine_flow_vec_m3hr

```
std::vector<double> Hydro::turbine_flow_vec_m3hr
```

A vector of the turbine flow [m3/hr] at each point in the modelling time series.

4.8.4.12 turbine_type

```
HydroTurbineType Hydro::turbine_type
```

The type of hydroelectric turbine model to use.

The documentation for this class was generated from the following files:

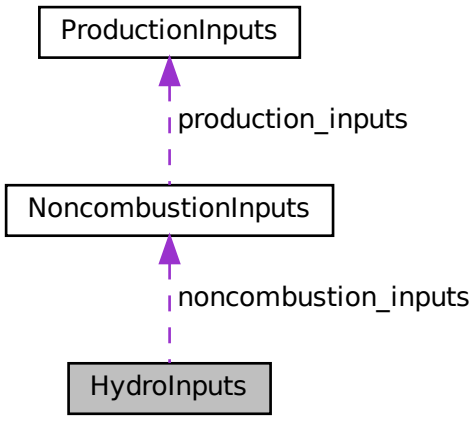
- header/Production/Noncombustion/Hydro.h
- source/Production/Noncombustion/Hydro.cpp

4.9 HydroInputs Struct Reference

A structure which bundles the necessary inputs for the Hydro constructor. Provides default values for every necessary input. Note that this structure encapsulates NoncombustionInputs.

`#include <Hydro.h>`

Collaboration diagram for HydroInputs:



Public Attributes

- NoncombustionInputs noncombustion_inputs

  An encapsulated NoncombustionInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double fluid_density_kgm3 = 1000

  The density [kg/m3] of the hydroelectric working fluid.
- double net_head_m = 500

  The net head [m] of the asset.
- double reservoir_capacity_m3 = 0

  The capacity [m3] of the hydro reservoir.
- double init_reservoir_state = 0

  The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- HydroTurbineType turbine_type = HydroTurbineType :: HYDRO_TURBINE_PELTON

  The type of hydroelectric turbine model to use.

4.9.1 Detailed Description

A structure which bundles the necessary inputs for the Hydro constructor. Provides default values for every necessary input. Note that this structure encapsulates NoncombustionInputs.

4.9.2 Member Data Documentation

4.9.2.1 capital_cost

```
double HydroInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.9.2.2 fluid_density_kgm3

```
double HydroInputs::fluid_density_kgm3 = 1000
```

The density [kg/m3] of the hydroelectric working fluid.

4.9.2.3 init_reservoir_state

```
double HydroInputs::init_reservoir_state = 0
```

The initial state of the reservoir (where state is volume of stored fluid divided by capacity).

4.9.2.4 net_head_m

```
double HydroInputs::net_head_m = 500
```

The net head [m] of the asset.

4.9.2.5 noncombustion_inputs

```
NoncombustionInputs HydroInputs::noncombustion_inputs
```

An encapsulated NoncombustionInputs instance.

4.9.2.6 operation_maintenance_cost_kWh

```
double HydroInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.9.2.7 reservoir_capacity_m3

```
double HydroInputs::reservoir_capacity_m3 = 0
```

The capacity [m3] of the hydro reservoir.

4.9.2.8 resource_key

```
int HydroInputs::resource_key = 0
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

4.9.2.9 turbine_type

```
HydroTurbineType HydroInputs::turbine_type = HydroTurbineType ::  HYDRO_TURBINE_PELTON
```

The type of hydroelectric turbine model to use.

The documentation for this struct was generated from the following file:

- header/Production/Noncombustion/Hydro.h

4.10 Interpolator Class Reference

A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies.

```
#include <Interpolator.h>
```

Public Member Functions

- Interpolator (void)

  Constructor for the Interpolator class.
- void addData1D (int, std::string)

  Method to add 1D interpolation data to the Interpolator.
- void addData2D (int, std::string)

  Method to add 2D interpolation data to the Interpolator.
- double interp1D (int, double)

  Method to perform a 1D interpolation.
- double interp2D (int, double, double)

  Method to perform a 2D interpolation.
- ∼Interpolator (void)

  Destructor for the Interpolator class.

Public Attributes

- std::map< int, InterpolatorStruct1D > interp_map_1D

  A map <int, InterpolatorStruct1D> of given 1D interpolation data.
- std::map< int, std::string > path_map_1D

  A map <int, string> of the paths (either relative or absolute) to the given 1D interpolation data.
- std::map< int, InterpolatorStruct2D > interp_map_2D

  A map <int, InterpolatorStruct2D> of given 2D interpolation data.
- std::map< int, std::string > path_map_2D

  A map <int, string> of the paths (either relative or absolute) to the given 2D interpolation data.

Private Member Functions

- void __checkDataKey1D (int)

  Helper method to check if given data key (1D) is already in use.
- void __checkDataKey2D (int)

  Helper method to check if given data key (2D) is already in use.
- void __checkBounds1D (int, double)

  Helper method to check that the given 1D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void __checkBounds2D (int, double, double)

  Helper method to check that the given 2D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void __throwReadError (std::string, int)

  Helper method to throw a read error whenever non-numeric data is encountered where only numeric data should be.
- bool __isNonNumeric (std::string)

  Helper method to determine if given string is non-numeric (i.e., contains.
- int __getInterpolationIndex (double, std::vector< double > ∗)

  Helper method to get appropriate interpolation index into given vector.
- std::vector< std::string > __splitCommaSeparatedString (std::string, std::string="||")

  Helper method to split a comma-separated string into a vector of substrings.
- std::vector< std::vector< std::string > > __getDataStringMatrix (std::string)
- void __readData1D (int, std::string)

  Helper method to read the given 1D interpolation data into Interpolator.
- void __readData2D (int, std::string)

  Helper method to read the given 2D interpolation data into Interpolator.

4.10.1 Detailed Description

A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 Interpolator()

```
Interpolator::Interpolator (
            void  )
```

Constructor for the Interpolator class.

```
682 {
683     //...
684
685     return;
686 }   /* Interpolator() */
```

4.10.2.2 ~Interpolator()

```
Interpolator::~Interpolator (
            void  )
```

Destructor for the Interpolator class.

```
868 {
869     //...
870
871     return;
872 }   /* ~Interpolator() */
```

4.10.3 Member Function Documentation

4.10.3.1 __checkBounds1D()

```
void Interpolator::__checkBounds1D (
            int data_key,
            double interp_x )  [private]
```

Helper method to check that the given 1D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.

Parameters

data_key	A key associated with the given interpolation data.
interp↩	The query value to be interpolated.

```
108 {
109     //  1. key error
110     if (this->interp_map_1D.count(data_key) == 0) {
111         std::string error_str = "ERROR:  Interpolator::interp1D()  ";
112         error_str += "data key ";
113         error_str += std::to_string(data_key);
114         error_str += " has not been registered";
115 
116         #ifdef _WIN32
117             std::cout « error_str « std::endl;
118         #endif
119 
120         throw std::invalid_argument(error_str);
121     }
122 
123     //  2. bounds error
124     if (
125         interp_x < this->interp_map_1D[data_key].min_x or
126         interp_x > this->interp_map_1D[data_key].max_x
127     ) {
128         std::string error_str = "ERROR:  Interpolator::interp1D()  ";
129         error_str += "interpolation value ";
130         error_str += std::to_string(interp_x);
131         error_str += " is outside of the given interpolation data domain [";
132         error_str += std::to_string(this->interp_map_1D[data_key].min_x);
133         error_str += " , ";
134         error_str += std::to_string(this->interp_map_1D[data_key].max_x);
135         error_str += "]";
136 
137         #ifdef _WIN32
138             std::cout « error_str « std::endl;
139         #endif
140 
141         throw std::invalid_argument(error_str);
142     }
143 
144     return;
145 }   /* __checkBounds1D() */
```

4.10.3.2 __checkBounds2D()

```
void Interpolator::__checkBounds2D (
            int data_key,
            double interp_x,
            double interp_y )  [private]
```

Helper method to check that the given 2D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.

Parameters

data_key	A key associated with the given interpolation data.
interp_x	The first query value to be interpolated.
interp_y	The second query value to be interpolated.

```
168 {
169     //  1. key error
170     if (this->interp_map_2D.count(data_key) == 0) {
171         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
172         error_str += "data key ";
173         error_str += std::to_string(data_key);
174         error_str += " has not been registered";
175 
176         #ifdef _WIN32
177             std::cout « error_str « std::endl;
178         #endif
179 
180         throw std::invalid_argument(error_str);
181     }
```

```
182
183     //  2. bounds error (x_interp)
184     if (
185         interp_x < this->interp_map_2D[data_key].min_x or
186         interp_x > this->interp_map_2D[data_key].max_x
187     ) {
188         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
189         error_str += "interpolation value interp_x = ";
190         error_str += std::to_string(interp_x);
191         error_str += " is outside of the given interpolation data domain [";
192         error_str += std::to_string(this->interp_map_2D[data_key].min_x);
193         error_str += " , ";
194         error_str += std::to_string(this->interp_map_2D[data_key].max_x);
195         error_str += "]";
196
197         #ifdef _WIN32
198             std::cout << error_str << std::endl;
199         #endif
200
201         throw std::invalid_argument(error_str);
202     }
203
204     //  2. bounds error (y_interp)
205     if (
206         interp_y < this->interp_map_2D[data_key].min_y or
207         interp_y > this->interp_map_2D[data_key].max_y
208     ) {
209         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
210         error_str += "interpolation value interp_y = ";
211         error_str += std::to_string(interp_y);
212         error_str += " is outside of the given interpolation data domain [";
213         error_str += std::to_string(this->interp_map_2D[data_key].min_y);
214         error_str += " , ";
215         error_str += std::to_string(this->interp_map_2D[data_key].max_y);
216         error_str += "]";
217
218         #ifdef _WIN32
219             std::cout << error_str << std::endl;
220         #endif
221
222         throw std::invalid_argument(error_str);
223     }
224
225     return;
226 }   /* __checkBounds2D() */
```

4.10.3.3 __checkDataKey1D()

```
void Interpolator::__checkDataKey1D (
            int data_key )  [private]
```

Helper method to check if given data key (1D) is already in use.

Parameters

data_key	The key associated with the given 1D interpolation data.

```
40 {
41     if (this->interp_map_1D.count(data_key) > 0) {
42         std::string error_str = "ERROR:  Interpolator::addData1D()  ";
43         error_str += "data key (1D) ";
44         error_str += std::to_string(data_key);
45         error_str += " is already in use";
46
47         #ifdef _WIN32
48             std::cout << error_str << std::endl;
49         #endif
50
51         throw std::invalid_argument(error_str);
52     }
53
54     return;
55 }   /* __checkDataKey1D() */
```

4.10.3.4 __checkDataKey2D()

```
void Interpolator::__checkDataKey2D (
            int data_key )  [private]
```

Helper method to check if given data key (2D) is already in use.

Parameters

data_key	The key associated with the given 2D interpolation data.

```
72 {
73     if (this->interp_map_2D.count(data_key) > 0) {
74         std::string error_str = "ERROR:  Interpolator::addData2D()  ";
75         error_str += "data key (2D) ";
76         error_str += std::to_string(data_key);
77         error_str += " is already in use";
78 
79         #ifdef _WIN32
80             std::cout << error_str << std::endl;
81         #endif
82 
83         throw std::invalid_argument(error_str);
84     }
85 
86     return;
87 }   /* __checkDataKey2D() */
```

4.10.3.5 __getDataStringMatrix()

```
std::vector< std::vector< std::string > > Interpolator::__getDataStringMatrix (
            std::string path_2_data )  [private]
```

```
401 {
402     //  1. create input file stream
403     std::ifstream ifs;
404     ifs.open(path_2_data);
405 
406     //  2. check that open() worked
407     if (not ifs.is_open()) {
408         std::string error_str = "ERROR:  Interpolator::__getDataStringMatrix()  ";
409         error_str += " failed to open ";
410         error_str += path_2_data;
411 
412         #ifdef _WIN32
413             std::cout << error_str << std::endl;
414         #endif
415 
416         throw std::invalid_argument(error_str);
417     }
418 
419     //  3. read file line by line
420     bool is_header = true;
421     std::string line;
422     std::vector<std::string> line_split_vec;
423     std::vector<std::vector<std::string>> string_matrix;
424 
425     while (not ifs.eof()) {
426         std::getline(ifs, line);
427 
428         if (is_header) {
429             is_header = false;
430             continue;
431         }
432 
433         line_split_vec = this->__splitCommaSeparatedString(line);
434 
435         if (not line_split_vec.empty()) {
436             string_matrix.push_back(line_split_vec);
437         }
438     }
439 
440     ifs.close();
441     return string_matrix;
442 }   /* __getDataStringMatrix() */
```

4.10.3.6 __getInterpolationIndex()

```
int Interpolator::__getInterpolationIndex (
            double interp_x,
            std::vector< double > * x_vec_ptr )  [private]
```

Helper method to get appropriate interpolation index into given vector.

Parameters

interp_x	The query value to be interpolated.
x_vec_ptr	A pointer to the given vector of interpolation data.

Returns

The appropriate interpolation index into the given vector.

```
318 {
319     int idx = 0;
320     while (
321         not (interp_x >= x_vec_ptr->at(idx) and interp_x <= x_vec_ptr->at(idx + 1))
322     ) {
323         idx++;
324     }
325
326     return idx;
327 }   /* __getInterpolationIndex() */
```

4.10.3.7 __isNonNumeric()

```
bool Interpolator::__isNonNumeric (
            std::string str )  [private]
```

Helper method to determine if given string is non-numeric (i.e., contains.

Parameters

str	The string being tested.

Returns

A boolean indicating if the given string is non-numeric.

```
283 {
284     for (size_t i = 0; i < str.size(); i++) {;
285         if (isalpha(str[i])) {
286             return true;
287         }
288     }
289
290     return false;
291 }   /* __isAlpha() */
```

4.10.3.8 __readData1D()

```
void Interpolator::__readData1D (
            int data_key,
            std::string path_2_data )  [private]
```

Helper method to read the given 1D interpolation data into Interpolator.

Parameters

data_key	A key associated with the given interpolation data.
path_2_data	The path (either relative or absolute) to the given interpolation data.

```
462 {
463     //  1. get string matrix
464     std::vector<std::vector<std::string>> string_matrix =
465         this->__getDataStringMatrix(path_2_data);
466 
467     //  2. read string matrix contents into 1D interpolation struct
468     InterpolatorStruct1D interp_struct_1D;
469 
470     interp_struct_1D.n_points = string_matrix.size();
471     interp_struct_1D.x_vec.resize(interp_struct_1D.n_points, 0);
472     interp_struct_1D.y_vec.resize(interp_struct_1D.n_points, 0);
473 
474     for (int i = 0; i < interp_struct_1D.n_points; i++) {
475         try {
476             interp_struct_1D.x_vec[i] = std::stod(string_matrix[i][0]);
477             interp_struct_1D.y_vec[i] = std::stod(string_matrix[i][1]);
478         }
479 
480         catch (...) {
481             this->__throwReadError(path_2_data, 1);
482         }
483     }
484 
485     interp_struct_1D.min_x = interp_struct_1D.x_vec[0];
486     interp_struct_1D.max_x = interp_struct_1D.x_vec[interp_struct_1D.n_points - 1];
487 
488     //  3. write struct to map
489     this->interp_map_1D.insert(
490         std::pair<int, InterpolatorStruct1D>(data_key, interp_struct_1D)
491     );
492 
493     /*
494     // ==== TEST PRINT ==== //
495     std::cout << std::endl;
496     std::cout << path_2_data << std::endl;
497     std::cout << "--------" << std::endl;
498 
499     std::cout << "n_points: " << this->interp_map_1D[data_key].n_points << std::endl;
500 
501     std::cout << "x_vec: [";
502     for (
503         int i = 0;
504         i < this->interp_map_1D[data_key].n_points;
505         i++
506     ) {
507         std::cout << this->interp_map_1D[data_key].x_vec[i] << ", ";
508     }
509     std::cout << "]" << std::endl;
510 
511     std::cout << "y_vec: [";
512     for (
513         int i = 0;
514         i < this->interp_map_1D[data_key].n_points;
515         i++
516     ) {
517         std::cout << this->interp_map_1D[data_key].y_vec[i] << ", ";
518     }
519     std::cout << "]" << std::endl;
520 
521     std::cout << std::endl;
522     // ==== END TEST PRINT ==== //
523     //*/
524 
525     return;
526 }   /* __readData1D() */
```

4.10.3.9 __readData2D()

```
void Interpolator::__readData2D (
            int data_key,
            std::string path_2_data )  [private]
```

Helper method to read the given 2D interpolation data into Interpolator.

Parameters

data_key	A key associated with the given interpolation data.
path_2_data	The path (either relative or absolute) to the given interpolation data.

```
546 {
547     //  1. get string matrix
548     std::vector<std::vector<std::string>> string_matrix =
549         this->__getDataStringMatrix(path_2_data);
550 
551     //  2. read string matrix contents into 2D interpolation map
552     InterpolatorStruct2D interp_struct_2D;
553 
554     interp_struct_2D.n_rows = string_matrix.size() - 1;
555     interp_struct_2D.n_cols = string_matrix[0].size() - 1;
556 
557     interp_struct_2D.x_vec.resize(interp_struct_2D.n_cols, 0);
558     interp_struct_2D.y_vec.resize(interp_struct_2D.n_rows, 0);
559 
560     interp_struct_2D.z_matrix.resize(interp_struct_2D.n_rows, {});
561 
562     for (int i = 0; i < interp_struct_2D.n_rows; i++) {
563         interp_struct_2D.z_matrix[i].resize(interp_struct_2D.n_cols, 0);
564     }
565 
566     for (size_t i = 1; i < string_matrix[0].size(); i++) {
567         try {
568             interp_struct_2D.x_vec[i - 1] = std::stod(string_matrix[0][i]);
569         }
570 
571         catch (...) {
572             this->__throwReadError(path_2_data, 2);
573         }
574     }
575 
576     interp_struct_2D.min_x = interp_struct_2D.x_vec[0];
577     interp_struct_2D.max_x = interp_struct_2D.x_vec[interp_struct_2D.n_cols - 1];
578 
579     for (size_t i = 1; i < string_matrix.size(); i++) {
580         try {
581             interp_struct_2D.y_vec[i - 1] = std::stod(string_matrix[i][0]);
582         }
583 
584         catch (...) {
585             this->__throwReadError(path_2_data, 2);
586         }
587     }
588 
589     interp_struct_2D.min_y = interp_struct_2D.y_vec[0];
590     interp_struct_2D.max_y = interp_struct_2D.y_vec[interp_struct_2D.n_rows - 1];
591 
592     for (size_t i = 1; i < string_matrix.size(); i++) {
593         for (size_t j = 1; j < string_matrix[0].size(); j++) {
594             try {
595                 interp_struct_2D.z_matrix[i - 1][j - 1] = std::stod(string_matrix[i][j]);
596             }
597 
598             catch (...) {
599                 this->__throwReadError(path_2_data, 2);
600             }
601         }
602     }
603 
604     //  3. write struct to map
605     this->interp_map_2D.insert(
606         std::pair<int, InterpolatorStruct2D>(data_key, interp_struct_2D)
607     );
608 
609     /*
610     // ==== TEST PRINT ==== //
611     std::cout << std::endl;
612     std::cout << path_2_data << std::endl;
```

```
613     std::cout « "--------" « std::endl;
614
615     std::cout « "n_rows: " « this->interp_map_2D[data_key].n_rows « std::endl;
616     std::cout « "n_cols: " « this->interp_map_2D[data_key].n_cols « std::endl;
617
618     std::cout « "x_vec: [";
619     for (
620         int i = 0;
621         i < this->interp_map_2D[data_key].n_cols;
622         i++
623     ) {
624         std::cout « this->interp_map_2D[data_key].x_vec[i] « ", ";
625     }
626     std::cout « "]" « std::endl;
627
628     std::cout « "y_vec: [";
629     for (
630         int i = 0;
631         i < this->interp_map_2D[data_key].n_rows;
632         i++
633     ) {
634         std::cout « this->interp_map_2D[data_key].y_vec[i] « ", ";
635     }
636     std::cout « "]" « std::endl;
637
638     std::cout « "z_matrix:" « std::endl;
639     for (
640         int i = 0;
641         i < this->interp_map_2D[data_key].n_rows;
642         i++
643     ) {
644         std::cout « "\t[";
645
646         for (
647             int j = 0;
648             j < this->interp_map_2D[data_key].n_cols;
649             j++
650         ) {
651             std::cout « this->interp_map_2D[data_key].z_matrix[i][j] « ", ";
652         }
653
654         std::cout « "]" « std::endl;
655     }
656     std::cout « std::endl;
657
658     std::cout « std::endl;
659     // ==== END TEST PRINT ==== //
660     //*/
661
662     return;
663 }   /* __readData2D() */
```

4.10.3.10 __splitCommaSeparatedString()

```
std::vector< std::string > Interpolator::__splitCommaSeparatedString (
            std::string str,
            std::string break_str = "||" )  [private]
```

Helper method to split a comma-separated string into a vector of substrings.

Parameters

str	The string to be split.
break_str	A string which triggers the function to break. What has been split up to the point of the break is then returned.

Returns

A vector of substrings, which follows from splitting the given string in a comma separated manner.

```
356 {
357     std::vector<std::string> str_split_vec;
358 
359     size_t idx = 0;
360     std::string substr;
361 
362     while ((idx = str.find(',')) != std::string::npos) {
363         substr = str.substr(0, idx);
364 
365         if (substr == break_str) {
366             break;
367         }
368 
369         str_split_vec.push_back(substr);
370 
371         str.erase(0, idx + 1);
372     }
373 
374     return str_split_vec;
375 }   /* __splitCommaSeparatedString() */
```

4.10.3.11 __throwReadError()

```
void Interpolator::__throwReadError (
            std::string path_2_data,
            int dimensions )  [private]
```

Helper method to throw a read error whenever non-numeric data is encountered where only numeric data should be.

Parameters

path_2_data	The path (either relative or absolute) to the given interpolation data.
dimensions	The dimensionality of the data being read.

```
247 {
248     std::string error_str = "ERROR:  Interpolator::addData";
249     error_str += std::to_string(dimensions);
250     error_str += "D()  ";
251     error_str += " failed to read ";
252     error_str += path_2_data;
253     error_str += " (this is probably a std::stod() error; is there non-numeric ";
254     error_str += "data where only numeric data should be?)";
255 
256     #ifdef _WIN32
257         std::cout << error_str << std::endl;
258     #endif
259 
260     throw std::runtime_error(error_str);
261 
262     return;
263 }   /* __throwReadError() */
```

4.10.3.12 addData1D()

```
void Interpolator::addData1D (
            int data_key,
            std::string path_2_data )
```

Method to add 1D interpolation data to the Interpolator.

Parameters

data_key	A key used to index into the Interpolator.
path_2_data	A path (either relative or absolute) to the given 1D interpolation data.

```
706 {
707     //  1. check key
708     this->__checkDataKey1D(data_key);
709 
710     //  2. read data into map
711     this->__readData1D(data_key, path_2_data);
712 
713     //  3. record path
714     this->path_map_1D.insert(std::pair<int, std::string>(data_key, path_2_data));
715 
716     return;
717 }   /* addData1D() */
```

4.10.3.13 addData2D()

```
void Interpolator::addData2D (
             int data_key,
             std::string path_2_data )
```

Method to add 2D interpolation data to the Interpolator.

Parameters

data_key	A key used to index into the Interpolator.
path_2_data	A path (either relative or absolute) to the given 2D interpolation data.

```
737 {
738     //  1. check key
739     this->__checkDataKey2D(data_key);
740 
741     //  2. read data into map
742     this->__readData2D(data_key, path_2_data);
743 
744     //  3. record path
745     this->path_map_2D.insert(std::pair<int, std::string>(data_key, path_2_data));
746 
747     return;
748 }   /* addData2D() */
```

4.10.3.14 interp1D()

```
double Interpolator::interp1D (
             int data_key,
             double interp_x )
```

Method to perform a 1D interpolation.

Parameters

data_key	A key used to index into the Interpolator.
interp_x	The query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.

Returns

An interpolation of the given query value.

```
770 {
771     //  1. check bounds
772     this->__checkBounds1D(data_key, interp_x);
773 
774     //  2. get interpolation index
775     int idx = this->__getInterpolationIndex(
776         interp_x,
777         &(this->interp_map_1D[data_key].x_vec)
778     );
779 
780     //  3. perform interpolation
781     double x_0 = this->interp_map_1D[data_key].x_vec[idx];
782     double x_1 = this->interp_map_1D[data_key].x_vec[idx + 1];
783 
784     double y_0 = this->interp_map_1D[data_key].y_vec[idx];
785     double y_1 = this->interp_map_1D[data_key].y_vec[idx + 1];
786 
787     double interp_y = ((y_1 - y_0) / (x_1 - x_0)) * (interp_x - x_0) + y_0;
788 
789     return interp_y;
790 }   /* interp1D() */
```

4.10.3.15 interp2D()

```
double Interpolator::interp2D (
            int data_key,
            double interp_x,
            double interp_y )
```

Method to perform a 2D interpolation.

Parameters

data_key	A key used to index into the Interpolator.
interp_x	The first query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.
interp_y	The second query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.

Returns

An interpolation of the given query values.

```
815 {
816     //  1. check bounds
817     this->__checkBounds2D(data_key, interp_x, interp_y);
818 
819     //  2. get interpolation indices
820     int idx_x = this->__getInterpolationIndex(
821         interp_x,
822         &(this->interp_map_2D[data_key].x_vec)
823     );
824 
825     int idx_y = this->__getInterpolationIndex(
826         interp_y,
827         &(this->interp_map_2D[data_key].y_vec)
828     );
829 
830     //  3. perform first horizontal interpolation
831     double x_0 = this->interp_map_2D[data_key].x_vec[idx_x];
832     double x_1 = this->interp_map_2D[data_key].x_vec[idx_x + 1];
833 
834     double z_0 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x];
835     double z_1 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x + 1];
```

```
836
837     double interp_z_0 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
838
839     //  4. perform second horizontal interpolation
840     z_0 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x];
841     z_1 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x + 1];
842
843     double interp_z_1 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
844
845     //  5. perform vertical interpolation
846     double y_0 = this->interp_map_2D[data_key].y_vec[idx_y];
847     double y_1 = this->interp_map_2D[data_key].y_vec[idx_y + 1];
848
849     double interp_z =
850         ((interp_z_1 - interp_z_0) / (y_1 - y_0)) * (interp_y - y_0) + interp_z_0;
851
852     return interp_z;
853 }   /* interp2D() */
```

4.10.4 Member Data Documentation

4.10.4.1 interp_map_1D

`std::map<int, InterpolatorStruct1D> Interpolator::interp_map_1D`

A map <int, InterpolatorStruct1D> of given 1D interpolation data.

4.10.4.2 interp_map_2D

`std::map<int, InterpolatorStruct2D> Interpolator::interp_map_2D`

A map <int, InterpolatorStruct2D> of given 2D interpolation data.

4.10.4.3 path_map_1D

`std::map<int, std::string> Interpolator::path_map_1D`

A map <int, string> of the paths (either relative or absolute) to the given 1D interpolation data.

4.10.4.4 path_map_2D

`std::map<int, std::string> Interpolator::path_map_2D`

A map <int, string> of the paths (either relative or absolute) to the given 2D interpolation data.

The documentation for this class was generated from the following files:

- header/Interpolator.h
- source/Interpolator.cpp

4.11 InterpolatorStruct1D Struct Reference

A struct which holds two parallel vectors for use in 1D interpolation.

`#include <Interpolator.h>`

Public Attributes

- int n_points = 0
  The number of data points in each parallel vector.
- std::vector< double > x_vec = {}
  A vector of independent data.
- double min_x = 0
  The minimum (i.e., first) element of x_vec.
- double max_x = 0
  The maximum (i.e., last) element of x_vec.
- std::vector< double > y_vec = {}
  A vector of dependent data.

4.11.1 Detailed Description

A struct which holds two parallel vectors for use in 1D interpolation.

4.11.2 Member Data Documentation

4.11.2.1 max_x

```
double InterpolatorStruct1D::max_x = 0
```

The maximum (i.e., last) element of x_vec.

4.11.2.2 min_x

```
double InterpolatorStruct1D::min_x = 0
```

The minimum (i.e., first) element of x_vec.

4.11.2.3 n_points

```
int InterpolatorStruct1D::n_points = 0
```

The number of data points in each parallel vector.

4.11.2.4 x_vec

```
std::vector<double> InterpolatorStruct1D::x_vec = {}
```

A vector of independent data.

4.11.2.5 y_vec

```
std::vector<double> InterpolatorStruct1D::y_vec = {}
```

A vector of dependent data.

The documentation for this struct was generated from the following file:

- header/Interpolator.h

4.12 InterpolatorStruct2D Struct Reference

A struct which holds two parallel vectors and a matrix for use in 2D interpolation.

```
#include <Interpolator.h>
```

Public Attributes

- int n_rows = 0
  The number of rows in the matrix (also the length of y_vec)
- int n_cols = 0
  The number of cols in the matrix (also the length of x_vec)
- std::vector< double > x_vec = {}
  A vector of independent data (columns).
- double min_x = 0
  The minimum (i.e., first) element of x_vec.
- double max_x = 0
  The maximum (i.e., last) element of x_vec.
- std::vector< double > y_vec = {}
  A vector of independent data (rows).
- double min_y = 0
  The minimum (i.e., first) element of y_vec.
- double max_y = 0
  The maximum (i.e., last) element of y_vec.
- std::vector< std::vector< double > > z_matrix = {}
  A matrix of dependent data.

4.12.1 Detailed Description

A struct which holds two parallel vectors and a matrix for use in 2D interpolation.

4.12.2 Member Data Documentation

4.12.2.1 max_x

```
double InterpolatorStruct2D::max_x = 0
```

The maximum (i.e., last) element of x_vec.

4.12.2.2 max_y

```
double InterpolatorStruct2D::max_y = 0
```

The maximum (i.e., last) element of y_vec.

4.12.2.3 min_x

```
double InterpolatorStruct2D::min_x = 0
```

The minimum (i.e., first) element of x_vec.

4.12.2.4 min_y

```
double InterpolatorStruct2D::min_y = 0
```

The minimum (i.e., first) element of y_vec.

4.12.2.5 n_cols

```
int InterpolatorStruct2D::n_cols = 0
```

The number of cols in the matrix (also the length of x_vec)

4.12.2.6 n_rows

```
int InterpolatorStruct2D::n_rows = 0
```

The number of rows in the matrix (also the length of y_vec)

4.12.2.7 x_vec

```
std::vector<double> InterpolatorStruct2D::x_vec = {}
```

A vector of independent data (columns).

4.12.2.8 y_vec

```
std::vector<double> InterpolatorStruct2D::y_vec = {}
```

A vector of independent data (rows).

4.12.2.9 z_matrix

```
std::vector<std::vector<double> > InterpolatorStruct2D::z_matrix = {}
```

A matrix of dependent data.

The documentation for this struct was generated from the following file:

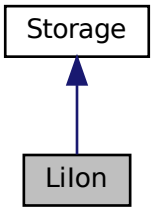
- header/Interpolator.h

4.13 LiIon Class Reference

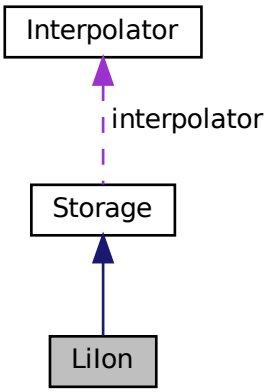
A derived class of Storage which models energy storage by way of lithium-ion batteries.

`#include <LiIon.h>`

Inheritance diagram for LiIon:



Collaboration diagram for LiIon:



Public Member Functions

- LiIon (void)

  Constructor (dummy) for the LiIon class.
- LiIon (int, double, LiIonInputs)

  Constructor (intended) for the LiIon class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double getAvailablekW (double)

Method to get the discharge power currently available from the asset.

- double getAcceptablekW (double)

  Method to get the charge power currently acceptable by the asset.

- void commitCharge (int, double, double)

  Method which takes in the charging power for the current timestep and records.

- double commitDischarge (int, double, double, double)

  Method which takes in the discharging power for the current timestep and records. Returns the load remaining after discharge.

- ~LiIon (void)

  Destructor for the LiIon class.

Public Attributes

- bool power_degradation_flag

  A flag which indicates whether or not power degradation should be modelled.

- double dynamic_energy_capacity_kWh

  The dynamic (i.e. degrading) energy capacity [kWh] of the asset.

- double dynamic_power_capacity_kW

  The dynamic (i.e. degrading) power capacity [kW] of the asset.

- double SOH

  The state of health of the asset.

- double replace_SOH

  The state of health at which the asset is considered "dead" and must be replaced.

- double degradation_alpha

  A dimensionless acceleration coefficient used in modelling energy capacity degradation.

- double degradation_beta

  A dimensionless acceleration exponent used in modelling energy capacity degradation.

- double degradation_B_hat_cal_0

  A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.

- double degradation_r_cal

  A dimensionless constant used in modelling energy capacity degradation.

- double degradation_Ea_cal_0

  A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.

- double degradation_a_cal

  A pre-exponential factor [J/mol] used in modelling energy capacity degradation.

- double degradation_s_cal

  A dimensionless constant used in modelling energy capacity degradation.

- double gas_constant_JmolK

  The universal gas constant [J/mol.K].

- double temperature_K

  The absolute environmental temperature [K] of the lithium ion battery energy storage system.

- double init_SOC

  The initial state of charge of the asset.

- double min_SOC

  The minimum state of charge of the asset. Will toggle is_depleted when reached.

- double hysteresis_SOC

  The state of charge the asset must achieve to toggle is_depleted.

- double max_SOC

  The maximum state of charge of the asset.

- double charging_efficiency

The charging efficiency of the asset.

- double discharging_efficiency

  The discharging efficiency of the asset.

- std::vector< double > SOH_vec

  A vector of the state of health of the asset at each point in the modelling time series.

Private Member Functions

- void __checkInputs (LiIonInputs)

  Helper method to check inputs to the LiIon constructor.

- double __getGenericCapitalCost (void)

  Helper method to generate a generic lithium ion battery energy storage system capital cost.

- double __getGenericOpMaintCost (void)

  Helper method to generate a generic lithium ion battery energy storage system operation and maintenance cost. This is a cost incurred per unit energy charged/discharged.

- void __toggleDepleted (void)

  Helper method to toggle the is_depleted attribute of LiIon.

- void __handleDegradation (int, double, double)

  Helper method to apply degradation modelling and update attributes.

- void __modelDegradation (double, double)

  Helper method to model energy capacity degradation as a function of operating state.

- double __getBcal (double)

  Helper method to compute and return the base pre-exponential factor for a given state of charge.

- double __getEacal (double)

  Helper method to compute and return the activation energy value for a given state of charge.

- void __writeSummary (std::string)

  Helper method to write summary results for LiIon.

- void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

  Helper method to write time series results for LiIon.

4.13.1 Detailed Description

A derived class of Storage which models energy storage by way of lithium-ion batteries.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 LiIon() [1/2]

```
LiIon::LiIon (
            void  )
```

Constructor (dummy) for the LiIon class.

```
649 {
650     return;
651 }   /* LiIon() */
```

4.13.2.2 LiIon() [2/2]

```
LiIon::LiIon (
            int n_points,
            double n_years,
            LiIonInputs liion_inputs )
```

Constructor (intended) for the LiIon class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
liion_inputs	A structure of LiIon constructor inputs.

```
679   :
680 Storage(
681     n_points,
682     n_years,
683     liion_inputs.storage_inputs
684 )
685 {
686     //  1. check inputs
687     this->__checkInputs(liion_inputs);
688
689     //  2. set attributes
690     this->type = StorageType :: LIION;
691     this->type_str = "LIION";
692
693     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
694     this->dynamic_power_capacity_kW = this->power_capacity_kW;
695
696     this->SOH = 1;
697     this->power_degradation_flag = liion_inputs.power_degradation_flag;
698     this->replace_SOH = liion_inputs.replace_SOH;
699
700     this->degradation_alpha = liion_inputs.degradation_alpha;
701     this->degradation_beta = liion_inputs.degradation_beta;
702     this->degradation_B_hat_cal_0 = liion_inputs.degradation_B_hat_cal_0;
703     this->degradation_r_cal = liion_inputs.degradation_r_cal;
704     this->degradation_Ea_cal_0 = liion_inputs.degradation_Ea_cal_0;
705     this->degradation_a_cal = liion_inputs.degradation_a_cal;
706     this->degradation_s_cal = liion_inputs.degradation_s_cal;
707     this->gas_constant_JmolK = liion_inputs.gas_constant_JmolK;
708     this->temperature_K = liion_inputs.temperature_K;
709
710     this->init_SOC = liion_inputs.init_SOC;
711     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
712
713     this->min_SOC = liion_inputs.min_SOC;
714     this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
715     this->max_SOC = liion_inputs.max_SOC;
716
717     this->charging_efficiency = liion_inputs.charging_efficiency;
718     this->discharging_efficiency = liion_inputs.discharging_efficiency;
719
720     if (liion_inputs.capital_cost < 0) {
721         this->capital_cost = this->__getGenericCapitalCost();
722     }
723     else {
724         this->capital_cost = liion_inputs.capital_cost;
725     }
726
727     if (liion_inputs.operation_maintenance_cost_kWh < 0) {
728         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
729     }
730     else {
731         this->operation_maintenance_cost_kWh =
732             liion_inputs.operation_maintenance_cost_kWh;
733     }
734
735     if (not this->is_sunk) {
736         this->capital_cost_vec[0] = this->capital_cost;
737     }
738
739     this->SOH_vec.resize(this->n_points, 0);
740
741     //  3. construction print
```

```
742     if (this->print_flag) {
743         std::cout « "LiIon object constructed at " « this « std::endl;
744     }
745
746     return;
747 }   /* LiIon() */
```

4.13.2.3 ~LiIon()

```
LiIon::~LiIon (
            void  )
```

Destructor for the LiIon class.

```
1004 {
1005     //  1. destruction print
1006     if (this->print_flag) {
1007         std::cout « "LiIon object at " « this « " destroyed" « std::endl;
1008     }
1009
1010     return;
1011 }   /* ~LiIon() */
```

4.13.3 Member Function Documentation

4.13.3.1 __checkInputs()

```
void LiIon::__checkInputs (
            LiIonInputs liion_inputs )  [private]
```

Helper method to check inputs to the LiIon constructor.

Parameters

liion_inputs	A structure of LiIon constructor inputs.

```
39 {
40     //  1. check replace_SOH
41     if (liion_inputs.replace_SOH < 0 or liion_inputs.replace_SOH > 1) {
42         std::string error_str = "ERROR:  LiIon():  replace_SOH must be in the closed ";
43         error_str += "interval [0, 1]";
44
45         #ifdef _WIN32
46             std::cout « error_str « std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     //  2. check init_SOC
53     if (liion_inputs.init_SOC < 0 or liion_inputs.init_SOC > 1) {
54         std::string error_str = "ERROR:  LiIon():  init_SOC must be in the closed ";
55         error_str += "interval [0, 1]";
56
57         #ifdef _WIN32
58             std::cout « error_str « std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     //  3. check min_SOC
```

```
65      if (liion_inputs.min_SOC < 0 or liion_inputs.min_SOC > 1) {
66          std::string error_str = "ERROR:  LiIon():  min_SOC must be in the closed ";
67          error_str += "interval [0, 1]";
68
69          #ifdef _WIN32
70              std::cout << error_str << std::endl;
71          #endif
72
73          throw std::invalid_argument(error_str);
74      }
75
76      //  4. check hysteresis_SOC
77      if (liion_inputs.hysteresis_SOC < 0 or liion_inputs.hysteresis_SOC > 1) {
78          std::string error_str = "ERROR:  LiIon():  hysteresis_SOC must be in the closed ";
79          error_str += "interval [0, 1]";
80
81          #ifdef _WIN32
82              std::cout << error_str << std::endl;
83          #endif
84
85          throw std::invalid_argument(error_str);
86      }
87
88      //  5. check max_SOC
89      if (liion_inputs.max_SOC < 0 or liion_inputs.max_SOC > 1) {
90          std::string error_str = "ERROR:  LiIon():  max_SOC must be in the closed ";
91          error_str += "interval [0, 1]";
92
93          #ifdef _WIN32
94              std::cout << error_str << std::endl;
95          #endif
96
97          throw std::invalid_argument(error_str);
98      }
99
100     //  6. check charging_efficiency
101     if (liion_inputs.charging_efficiency <= 0 or liion_inputs.charging_efficiency > 1) {
102         std::string error_str = "ERROR:  LiIon():  charging_efficiency must be in the ";
103         error_str += "half-open interval (0, 1]";
104
105         #ifdef _WIN32
106             std::cout << error_str << std::endl;
107         #endif
108
109         throw std::invalid_argument(error_str);
110     }
111
112     //  7. check discharging_efficiency
113     if (
114         liion_inputs.discharging_efficiency <= 0 or
115         liion_inputs.discharging_efficiency > 1
116     ) {
117         std::string error_str = "ERROR:  LiIon():  discharging_efficiency must be in the ";
118         error_str += "half-open interval (0, 1]";
119
120         #ifdef _WIN32
121             std::cout << error_str << std::endl;
122         #endif
123
124         throw std::invalid_argument(error_str);
125     }
126
127     //  8. check degradation_alpha
128     if (liion_inputs.degradation_alpha <= 0) {
129         std::string error_str = "ERROR:  LiIon():  degradation_alpha must be > 0";
130
131         #ifdef _WIN32
132             std::cout << error_str << std::endl;
133         #endif
134
135         throw std::invalid_argument(error_str);
136     }
137
138     //  9. check degradation_beta
139     if (liion_inputs.degradation_beta <= 0) {
140         std::string error_str = "ERROR:  LiIon():  degradation_beta must be > 0";
141
142         #ifdef _WIN32
143             std::cout << error_str << std::endl;
144         #endif
145
146         throw std::invalid_argument(error_str);
147     }
148
149     //  10. check degradation_B_hat_cal_0
150     if (liion_inputs.degradation_B_hat_cal_0 <= 0) {
151         std::string error_str = "ERROR:  LiIon():  degradation_B_hat_cal_0 must be > 0";
```

```
152
153         #ifdef _WIN32
154             std::cout << error_str << std::endl;
155         #endif
156
157         throw std::invalid_argument(error_str);
158     }
159
160     //  11. check degradation_r_cal
161     if (liion_inputs.degradation_r_cal < 0) {
162         std::string error_str = "ERROR:  LiIon():  degradation_r_cal must be >= 0";
163
164         #ifdef _WIN32
165             std::cout << error_str << std::endl;
166         #endif
167
168         throw std::invalid_argument(error_str);
169     }
170
171     //  12. check degradation_Ea_cal_0
172     if (liion_inputs.degradation_Ea_cal_0 <= 0) {
173         std::string error_str = "ERROR:  LiIon():  degradation_Ea_cal_0 must be > 0";
174
175         #ifdef _WIN32
176             std::cout << error_str << std::endl;
177         #endif
178
179         throw std::invalid_argument(error_str);
180     }
181
182     //  13. check degradation_a_cal
183     if (liion_inputs.degradation_a_cal < 0) {
184         std::string error_str = "ERROR:  LiIon():  degradation_a_cal must be >= 0";
185
186         #ifdef _WIN32
187             std::cout << error_str << std::endl;
188         #endif
189
190         throw std::invalid_argument(error_str);
191     }
192
193     //  14. check degradation_s_cal
194     if (liion_inputs.degradation_s_cal < 0) {
195         std::string error_str = "ERROR:  LiIon():  degradation_s_cal must be >= 0";
196
197         #ifdef _WIN32
198             std::cout << error_str << std::endl;
199         #endif
200
201         throw std::invalid_argument(error_str);
202     }
203
204     //  15. check gas_constant_JmolK
205     if (liion_inputs.gas_constant_JmolK <= 0) {
206         std::string error_str = "ERROR:  LiIon():  gas_constant_JmolK must be > 0";
207
208         #ifdef _WIN32
209             std::cout << error_str << std::endl;
210         #endif
211
212         throw std::invalid_argument(error_str);
213     }
214
215     //  16. check temperature_K
216     if (liion_inputs.temperature_K < 0) {
217         std::string error_str = "ERROR:  LiIon():  temperature_K must be >= 0";
218
219         #ifdef _WIN32
220             std::cout << error_str << std::endl;
221         #endif
222
223         throw std::invalid_argument(error_str);
224     }
225
226     return;
227 }   /* __checkInputs() */
```

4.13.3.2 __getBcal()

```
double LiIon::__getBcal (
            double SOC )  [private]
```

Helper method to compute and return the base pre-exponential factor for a given state of charge.

Ref: Truelove [2023a]

Parameters

SOC	The current state of charge of the asset.

Returns

The base pre-exponential factor for the given state of charge.

```
431 {
432     double B_cal = this->degradation_B_hat_cal_0 *
433         exp(this->degradation_r_cal * SOC);
434 
435     return B_cal;
436 }   /* __getBcal() */
```

4.13.3.3 __getEacal()

```
double LiIon::__getEacal (
            double SOC )  [private]
```

Helper method to compute and return the activation energy value for a given state of charge.

Ref: Truelove [2023a]

Parameters

SOC	The current state of charge of the asset.

Returns

The activation energy value for the given state of charge.

```
458 {
459     double Ea_cal = this->degradation_Ea_cal_0;
460 
461     Ea_cal -= this->degradation_a_cal *
462         (exp(this->degradation_s_cal * SOC) - 1);
463 
464     return Ea_cal;
465 }   /* __getEacal( */
```

4.13.3.4 __getGenericCapitalCost()

```
double LiIon::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic lithium ion battery energy storage system capital cost.

This model was obtained by way of surveying an assortment of published lithium ion battery energy storage system costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD].

Returns

A generic capital cost for the lithium ion battery energy storage system [CAD].

```
250 {
251     double capital_cost_per_kWh = 250 * pow(this->energy_capacity_kWh, -0.15) + 650;
252 
253     return capital_cost_per_kWh * this->energy_capacity_kWh;
254 }   /* __getGenericCapitalCost() */
```

4.13.3.5 __getGenericOpMaintCost()

```
double LiIon::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method to generate a generic lithium ion battery energy storage system operation and maintenance cost. This is a cost incurred per unit energy charged/discharged.

This model was obtained by way of surveying an assortment of published lithium ion battery energy storage system costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Returns

A generic operation and maintenance cost, per unit energy charged/discharged, for the lithium ion battery energy storage system [CAD/kWh].

```
278 {
279     return 0.01;
280 }   /* __getGenericOpMaintCost() */
```

4.13.3.6 __handleDegradation()

```
void LiIon::__handleDegradation (
            int timestep,
            double dt_hrs,
            double charging_discharging_kW )  [private]
```

Helper method to apply degradation modelling and update attributes.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
charging_discharging_kW	The charging/discharging power [kw] being sent to the asset.

```
348 {
349     //  1. model degradation
350     this->__modelDegradation(dt_hrs, charging_discharging_kW);
351 
352     //  2. update and record
353     this->SOH_vec[timestep] = this->SOH;
354     this->dynamic_energy_capacity_kWh = this->SOH * this->energy_capacity_kWh;
355 
356     if (this->power_degradation_flag) {
357         this->dynamic_power_capacity_kW = this->SOH * this->power_capacity_kW;
358     }
```

```
359
360     return;
361 }   /* __handleDegradation() */
```

4.13.3.7 __modelDegradation()

```
void LiIon::__modelDegradation (
            double dt_hrs,
            double charging_discharging_kW )  [private]
```

Helper method to model energy capacity degradation as a function of operating state.

Ref: Truelove [2023a]

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.
charging_discharging_kW	The charging/discharging power [kw] being sent to the asset.

```
384 {
385     //  1. compute SOC
386     double SOC = this->charge_kWh / this->energy_capacity_kWh;
387
388     //  2. compute C-rate and corresponding acceleration factor
389     double C_rate = charging_discharging_kW / this->power_capacity_kW;
390
391     double C_acceleration_factor =
392         1 + this->degradation_alpha * pow(C_rate, this->degradation_beta);
393
394     //  3. compute dSOH / dt
395     double B_cal = __getBcal(SOC);
396     double Ea_cal = __getEacal(SOC);
397
398     double dSOH_dt = B_cal *
399         exp((-1 * Ea_cal) / (this->gas_constant_JmolK * this->temperature_K));
400
401     dSOH_dt *= dSOH_dt;
402     dSOH_dt *= 1 / (2 * this->SOH);
403     dSOH_dt *= C_acceleration_factor;
404
405     //  4. update state of health
406     this->SOH -= dSOH_dt * dt_hrs;
407
408     return;
409 }   /* __modelDegradation() */
```

4.13.3.8 __toggleDepleted()

```
void LiIon::__toggleDepleted (
            void  )  [private]
```

Helper method to toggle the is_depleted attribute of LiIon.

```
295 {
296     if (this->is_depleted) {
297         double hysteresis_charge_kWh = this->hysteresis_SOC * this->energy_capacity_kWh;
298
299         if (hysteresis_charge_kWh > this->dynamic_energy_capacity_kWh) {
300             hysteresis_charge_kWh = this->dynamic_energy_capacity_kWh;
301         }
302
303         if (this->charge_kWh >= hysteresis_charge_kWh) {
304             this->is_depleted = false;
```

```
305         }
306     }
307
308     else {
309         double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
310
311         if (this->charge_kWh <= min_charge_kWh) {
312             this->is_depleted = true;
313         }
314     }
315
316     return;
317 }   /* __toggleDepleted() */
```

4.13.3.9 __writeSummary()

```
void LiIon::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for LiIon.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Storage.

```
483 {
484     //  1. create filestream
485     write_path += "summary_results.md";
486     std::ofstream ofs;
487     ofs.open(write_path, std::ofstream::out);
488
489     //  2. write summary results (markdown)
490     ofs « "# ";
491     ofs « std::to_string(int(ceil(this->power_capacity_kW)));
492     ofs « " kW ";
493     ofs « std::to_string(int(ceil(this->energy_capacity_kWh)));
494     ofs « " kWh LIION Summary Results\n";
495     ofs « "\n--------\n\n";
496
497     //  2.1. Storage attributes
498     ofs « "## Storage Attributes\n";
499     ofs « "\n";
500     ofs « "Power Capacity: " « this->power_capacity_kW « " kW  \n";
501     ofs « "Energy Capacity: " « this->energy_capacity_kWh « " kWh  \n";
502     ofs « "\n";
503
504     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
505     ofs « "Capital Cost: " « this->capital_cost « "  \n";
506     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
507         « " per kWh charged/discharged  \n";
508     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
509         « "  \n";
510     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
511         « "  \n";
512     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
513
514     ofs « "\n--------\n\n";
515
516     //  2.2. LiIon attributes
517     ofs « "## LiIon Attributes\n";
518     ofs « "\n";
519
520     ofs « "Charging Efficiency: " « this->charging_efficiency « "  \n";
521     ofs « "Discharging Efficiency: " « this->discharging_efficiency « "  \n";
522     ofs « "\n";
523
524     ofs « "Initial State of Charge: " « this->init_SOC « "  \n";
525     ofs « "Minimum State of Charge: " « this->min_SOC « "  \n";
526     ofs « "Hyteresis State of Charge: " « this->hysteresis_SOC « "  \n";
527     ofs « "Maximum State of Charge: " « this->max_SOC « "  \n";
```

```
528     ofs « "\n";
529
530     ofs « "Replacement State of Health: " « this->replace_SOH « "  \n";
531     ofs « "\n";
532
533     ofs « "Degradation Acceleration Coeff.: " « this->degradation_alpha « "  \n";
534     ofs « "Degradation Acceleration Exp.: " « this->degradation_beta « "  \n";
535     ofs « "Degradation Base Pre-Exponential Factor: "
536         « this->degradation_B_hat_cal_0 « " 1/sqrt(hrs)  \n";
537     ofs « "Degradation Dimensionless Constant (r_cal): "
538         « this->degradation_r_cal « "  \n";
539     ofs « "Degradation Base Activation Energy: "
540         « this->degradation_Ea_cal_0 « " J/mol  \n";
541     ofs « "Degradation Pre-Exponential Factor: "
542         « this->degradation_a_cal « " J/mol  \n";
543     ofs « "Degradation Dimensionless Constant (s_cal): "
544         « this->degradation_s_cal « "  \n";
545     ofs « "Universal Gas Constant: " « this->gas_constant_JmolK
546         « " J/mol.K  \n";
547     ofs « "Absolute Environmental Temperature: " « this->temperature_K « " K  \n";
548
549     ofs « "\n--------\n\n";
550
551     //  2.3. LiIon Results
552     ofs « "## Results\n";
553     ofs « "\n";
554
555     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
556     ofs « "\n";
557
558     ofs « "Total Discharge: " « this->total_discharge_kWh
559         « " kWh  \n";
560
561     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
562         « " per kWh dispatched  \n";
563     ofs « "\n";
564
565     ofs « "Replacements: " « this->n_replacements « "  \n";
566
567     ofs « "\n--------\n\n";
568     ofs.close();
569     return;
570 }   /* __writeSummary() */
```

4.13.3.10 __writeTimeSeries()

```
void LiIon::__writeTimeSeries (
              std::string write_path,
              std::vector< double > * time_vec_hrs_ptr,
              int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for LiIon.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
max_lines	The maximum number of lines of output to write.

Reimplemented from Storage.

```
601 {
602     //  1. create filestream
603     write_path += "time_series_results.csv";
604     std::ofstream ofs;
605     ofs.open(write_path, std::ofstream::out);
606
607     //  2. write time series results (comma separated value)
608     ofs « "Time (since start of data) [hrs],";
609     ofs « "Charging Power [kW],";
```

```
610     ofs « "Discharging Power [kW],";
611     ofs « "Charge (at end of timestep) [kWh],";
612     ofs « "State of Health (at end of timestep) [ ],";
613     ofs « "Capital Cost (actual),";
614     ofs « "Operation and Maintenance Cost (actual),";
615     ofs « "\n";
616 
617     for (int i = 0; i < max_lines; i++) {
618         ofs « time_vec_hrs_ptr->at(i) « ",";
619         ofs « this->charging_power_vec_kW[i] « ",";
620         ofs « this->discharging_power_vec_kW[i] « ",";
621         ofs « this->charge_vec_kWh[i] « ",";
622         ofs « this->SOH_vec[i] « ",";
623         ofs « this->capital_cost_vec[i] « ",";
624         ofs « this->operation_maintenance_cost_vec[i] « ",";
625         ofs « "\n";
626     }
627 
628     ofs.close();
629     return;
630 }   /* __writeTimeSeries() */
```

4.13.3.11 commitCharge()

```
void LiIon::commitCharge (
            int timestep,
            double dt_hrs,
            double charge_kW )  [virtual]
```

Method which takes in the charging power for the current timestep and records.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
charging_kW	The charging power [kw] being sent to the asset.

Reimplemented from Storage.

```
895 {
896     //  1. record charging power
897     this->charging_power_vec_kW[timestep] = charging_kW;
898 
899     //  2. update charge and record
900     this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
901     this->charge_vec_kWh[timestep] = this->charge_kWh;
902 
903     //  3. toggle depleted flag (if applicable)
904     this->__toggleDepleted();
905 
906     //  4. model degradation
907     this->__handleDegradation(timestep, dt_hrs, charging_kW);
908 
909     //  5. trigger replacement (if applicable)
910     if (this->SOH <= this->replace_SOH) {
911         this->handleReplacement(timestep);
912     }
913 
914     //  6. capture operation and maintenance costs (if applicable)
915     if (charging_kW > 0) {
916         this->operation_maintenance_cost_vec[timestep] = charging_kW * dt_hrs *
917             this->operation_maintenance_cost_kWh;
918     }
919 
920     this->power_kW= 0;
921     return;
922 }   /* commitCharge() */
```

4.13.3.12 commitDischarge()

```
double LiIon::commitDischarge (
            int timestep,
            double dt_hrs,
            double discharging_kW,
            double load_kW )  [virtual]
```

Method which takes in the discharging power for the current timestep and records. Returns the load remaining after discharge.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
discharging_kW	The discharging power [kw] being drawn from the asset.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the discharge is deducted from it.

Reimplemented from Storage.

```
958 {
959     //  1. record discharging power, update total
960     this->discharging_power_vec_kW[timestep] = discharging_kW;
961     this->total_discharge_kWh += discharging_kW * dt_hrs;
962 
963     //  2. update charge and record
964     this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
965     this->charge_vec_kWh[timestep] = this->charge_kWh;
966 
967     //  3. update load
968     load_kW -= discharging_kW;
969 
970     //  4. toggle depleted flag (if applicable)
971     this->__toggleDepleted();
972 
973     //  5. model degradation
974     this->__handleDegradation(timestep, dt_hrs, discharging_kW);
975 
976     //  6. trigger replacement (if applicable)
977     if (this->SOH <= this->replace_SOH) {
978         this->handleReplacement(timestep);
979     }
980 
981     //  7. capture operation and maintenance costs (if applicable)
982     if (discharging_kW > 0) {
983         this->operation_maintenance_cost_vec[timestep] = discharging_kW * dt_hrs *
984             this->operation_maintenance_cost_kWh;
985     }
986 
987     this->power_kW = 0;
988     return load_kW;
989 }   /* commitDischarge() */
```

4.13.3.13 getAcceptablekW()

```
double LiIon::getAcceptablekW (
            double dt_hrs )  [virtual]
```

Method to get the charge power currently acceptable by the asset.

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

The charging power [kW] currently acceptable by the asset.

Reimplemented from Storage.

```
839 {
840     //  1. get max charge
841     double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;
842 
843     if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
844         max_charge_kWh = this->dynamic_energy_capacity_kWh;
845     }
846 
847     //  2. compute acceptable power
848     //     (accounting for the power currently being charged/discharged by the asset)
849     double acceptable_kW =
850         (max_charge_kWh - this->charge_kWh) /
851         (this->charging_efficiency * dt_hrs);
852 
853     acceptable_kW -= this->power_kW;
854 
855     if (acceptable_kW <= 0) {
856         return 0;
857     }
858 
859     //  3. apply power constraint
860     if (acceptable_kW > this->dynamic_power_capacity_kW) {
861         acceptable_kW = this->dynamic_power_capacity_kW;
862     }
863 
864     return acceptable_kW;
865 }   /* getAcceptablekW( */
```

4.13.3.14 getAvailablekW()

```
double LiIon::getAvailablekW (
            double dt_hrs )  [virtual]
```

Method to get the discharge power currently available from the asset.

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

The discharging power [kW] currently available from the asset.

Reimplemented from Storage.

```
798 {
799     //  1. get min charge
800     double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
801 
802     //  2. compute available power
803     //     (accounting for the power currently being charged/discharged by the asset)
804     double available_kW =
805         ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
806         dt_hrs;
```

```
807
808     available_kW -= this->power_kW;
809
810     if (available_kW <= 0) {
811         return 0;
812     }
813
814     //  3. apply power constraint
815     if (available_kW > this->dynamic_power_capacity_kW) {
816         available_kW = this->dynamic_power_capacity_kW;
817     }
818
819     return available_kW;
820 }   /* getAvailablekW() */
```

4.13.3.15 handleReplacement()

```
void LiIon::handleReplacement (
             int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Storage.

```
765 {
766     //  1. reset attributes
767     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
768     this->dynamic_power_capacity_kW = this->power_capacity_kW;
769     this->SOH = 1;
770
771     // 2. invoke base class method
772     Storage::handleReplacement(timestep);
773
774     //  3. correct attributes
775     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
776     this->is_depleted = false;
777
778     return;
779 }   /* __handleReplacement() */
```

4.13.4 Member Data Documentation

4.13.4.1 charging_efficiency

```
double LiIon::charging_efficiency
```

The charging efficiency of the asset.

4.13.4.2 degradation_a_cal

```
double LiIon::degradation_a_cal
```

A pre-exponential factor [J/mol] used in modelling energy capacity degradation.

4.13.4.3 degradation_alpha

```
double LiIon::degradation_alpha
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.13.4.4 degradation_B_hat_cal_0

```
double LiIon::degradation_B_hat_cal_0
```

A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.

4.13.4.5 degradation_beta

```
double LiIon::degradation_beta
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.13.4.6 degradation_Ea_cal_0

```
double LiIon::degradation_Ea_cal_0
```

A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.

4.13.4.7 degradation_r_cal

```
double LiIon::degradation_r_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.13.4.8 degradation_s_cal

```
double LiIon::degradation_s_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.13.4.9 discharging_efficiency

```
double LiIon::discharging_efficiency
```

The discharging efficiency of the asset.

4.13.4.10 dynamic_energy_capacity_kWh

```
double LiIon::dynamic_energy_capacity_kWh
```

The dynamic (i.e. degrading) energy capacity [kWh] of the asset.

4.13.4.11 dynamic_power_capacity_kW

```
double LiIon::dynamic_power_capacity_kW
```

The dynamic (i.e. degrading) power capacity [kW] of the asset.

4.13.4.12 gas_constant_JmolK

```
double LiIon::gas_constant_JmolK
```

The universal gas constant [J/mol.K].

4.13.4.13 hysteresis_SOC

```
double LiIon::hysteresis_SOC
```

The state of charge the asset must achieve to toggle is_depleted.

4.13.4.14 init_SOC

```
double LiIon::init_SOC
```

The initial state of charge of the asset.

4.13.4.15 max_SOC

```
double LiIon::max_SOC
```

The maximum state of charge of the asset.

4.13.4.16 min_SOC

```
double LiIon::min_SOC
```

The minimum state of charge of the asset. Will toggle is_depleted when reached.

4.13.4.17 power_degradation_flag

```
bool LiIon::power_degradation_flag
```

A flag which indicates whether or not power degradation should be modelled.

4.13.4.18 replace_SOH

```
double LiIon::replace_SOH
```

The state of health at which the asset is considered "dead" and must be replaced.

4.13.4.19 SOH

```
double LiIon::SOH
```

The state of health of the asset.

4.13.4.20 SOH_vec

```
std::vector<double> LiIon::SOH_vec
```

A vector of the state of health of the asset at each point in the modelling time series.

4.13.4.21 temperature_K

```
double LiIon::temperature_K
```

The absolute environmental temperature [K] of the lithium ion battery energy storage system.

The documentation for this class was generated from the following files:

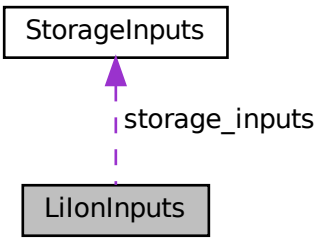
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

4.14 LiIonInputs Struct Reference

A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.

```
#include <LiIon.h>
```

Collaboration diagram for LiIonInputs:



Public Attributes

- StorageInputs storage_inputs

  An encapsulated StorageInputs instance.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double init_SOC = 0.5

  The initial state of charge of the asset.
- double min_SOC = 0.15

  The minimum state of charge of the asset. Will toggle is_depleted when reached.
- double hysteresis_SOC = 0.5

  The state of charge the asset must achieve to toggle is_depleted.
- double max_SOC = 0.9

  The maximum state of charge of the asset.
- double charging_efficiency = 0.9

  The charging efficiency of the asset.
- double discharging_efficiency = 0.9

  The discharging efficiency of the asset.
- double replace_SOH = 0.8

  The state of health at which the asset is considered "dead" and must be replaced.
- bool power_degradation_flag = false

  A flag which indicates whether or not power degradation should be modelled.
- double degradation_alpha = 8.935

  A dimensionless acceleration coefficient used in modelling energy capacity degradation.
- double degradation_beta = 1

  A dimensionless acceleration exponent used in modelling energy capacity degradation.
- double degradation_B_hat_cal_0 = 5.22226e6

  A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.
- double degradation_r_cal = 0.4361

  A dimensionless constant used in modelling energy capacity degradation.
- double degradation_Ea_cal_0 = 5.279e4

  A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.
- double degradation_a_cal = 100

  A pre-exponential factor [J/mol] used in modelling energy capacity degradation.
- double degradation_s_cal = 2

  A dimensionless constant used in modelling energy capacity degradation.
- double gas_constant_JmolK = 8.31446

  The universal gas constant [J/mol.K].
- double temperature_K = 273 + 20

  The absolute environmental temperature [K] of the lithium ion battery energy storage system.

4.14.1 Detailed Description

A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.

Ref: Truelove [2023a]

4.14.2 Member Data Documentation

4.14.2.1 capital_cost

```
double LiIonInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.14.2.2 charging_efficiency

```
double LiIonInputs::charging_efficiency = 0.9
```

The charging efficiency of the asset.

4.14.2.3 degradation_a_cal

```
double LiIonInputs::degradation_a_cal = 100
```

A pre-exponential factor [J/mol] used in modelling energy capacity degradation.

4.14.2.4 degradation_alpha

```
double LiIonInputs::degradation_alpha = 8.935
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.14.2.5 degradation_B_hat_cal_0

```
double LiIonInputs::degradation_B_hat_cal_0 = 5.22226e6
```

A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.

4.14.2.6 degradation_beta

```
double LiIonInputs::degradation_beta = 1
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.14.2.7 degradation_Ea_cal_0

```
double LiIonInputs::degradation_Ea_cal_0 = 5.279e4
```

A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.

4.14.2.8 degradation_r_cal

```
double LiIonInputs::degradation_r_cal = 0.4361
```

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.9 degradation_s_cal

```
double LiIonInputs::degradation_s_cal = 2
```

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.10 discharging_efficiency

```
double LiIonInputs::discharging_efficiency = 0.9
```

The discharging efficiency of the asset.

4.14.2.11 gas_constant_JmolK

```
double LiIonInputs::gas_constant_JmolK = 8.31446
```

The universal gas constant [J/mol.K].

4.14.2.12 hysteresis_SOC

```
double LiIonInputs::hysteresis_SOC = 0.5
```

The state of charge the asset must achieve to toggle is_depleted.

4.14.2.13 init_SOC

```
double LiIonInputs::init_SOC = 0.5
```

The initial state of charge of the asset.

4.14.2.14 max_SOC

```
double LiIonInputs::max_SOC = 0.9
```

The maximum state of charge of the asset.

4.14.2.15 min_SOC

```
double LiIonInputs::min_SOC = 0.15
```

The minimum state of charge of the asset. Will toggle is_depleted when reached.

4.14.2.16 operation_maintenance_cost_kWh

```
double LiIonInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.14.2.17 power_degradation_flag

```
bool LiIonInputs::power_degradation_flag = false
```

A flag which indicates whether or not power degradation should be modelled.

4.14.2.18 replace_SOH

```
double LiIonInputs::replace_SOH = 0.8
```

The state of health at which the asset is considered "dead" and must be replaced.

4.14.2.19 storage_inputs

```
StorageInputs LiIonInputs::storage_inputs
```

An encapsulated StorageInputs instance.

4.14.2.20 temperature_K

```
double LiIonInputs::temperature_K = 273 + 20
```

The absolute environmental temperature [K] of the lithium ion battery energy storage system.

The documentation for this struct was generated from the following file:

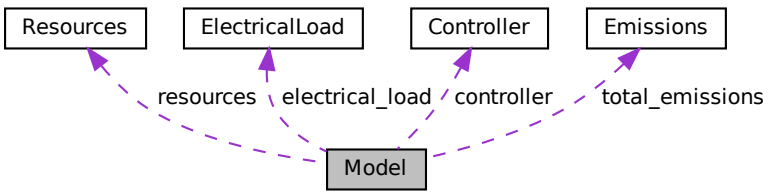
- header/Storage/LiIon.h

4.15 Model Class Reference

A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

```
#include <Model.h>
```

Collaboration diagram for Model:



Public Member Functions

- Model (void)

  Constructor (dummy) for the Model class.
- Model (ModelInputs)

  Constructor (intended) for the Model class.
- void addDiesel (DieselInputs)

  Method to add a Diesel asset to the Model.
- void addResource (NoncombustionType, std::string, int)

  A method to add a renewable resource time series to the Model.
- void addResource (RenewableType, std::string, int)

  A method to add a renewable resource time series to the Model.
- void addHydro (HydroInputs)

  Method to add a Hydro asset to the Model.
- void addSolar (SolarInputs)

  Method to add a Solar asset to the Model.
- void addTidal (TidalInputs)

  Method to add a Tidal asset to the Model.
- void addWave (WaveInputs)

  Method to add a Wave asset to the Model.
- void addWind (WindInputs)

  Method to add a Wind asset to the Model.
- void addLiIon (LiIonInputs)

  Method to add a LiIon asset to the Model.
- void run (void)

  A method to run the Model.
- void reset (void)

  Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors and resets select Model attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.
- void clear (void)

  Method to clear all attributes of the Model object.
- void writeResults (std::string, int=-1)

  Method which writes Model results to an output directory. Also calls out to writeResults() for each contained asset.
- ~Model (void)

  Destructor for the Model class.

Public Attributes

- double total_fuel_consumed_L

  The total fuel consumed [L] over a model run.
- Emissions total_emissions

  An Emissions structure for holding total emissions [kg].
- double net_present_cost

  The net present cost of the Model (undefined currency).
- double total_renewable_dispatch_kWh

  The total energy dispatched [kWh] by all renewable assets over the Model run.
- double total_dispatch_discharge_kWh

  The total energy dispatched/discharged [kWh] over the Model run.
- double levellized_cost_of_energy_kWh

  The levellized cost of energy, per unit energy dispatched/discharged, of the Model [1/kWh] (undefined currency).

- Controller controller

  Controller component of Model.
- ElectricalLoad electrical_load

  ElectricalLoad component of Model.
- Resources resources

  Resources component of Model.
- std::vector< Combustion * > combustion_ptr_vec

  A vector of pointers to the various Combustion assets in the Model.
- std::vector< Noncombustion * > noncombustion_ptr_vec

  A vector of pointers to the various Noncombustion assets in the Model.
- std::vector< Renewable * > renewable_ptr_vec

  A vector of pointers to the various Renewable assets in the Model.
- std::vector< Storage * > storage_ptr_vec

  A vector of pointers to the various Storage assets in the Model.

Private Member Functions

- void __checkInputs (ModelInputs)

  Helper method (private) to check inputs to the Model constructor.
- void __computeFuelAndEmissions (void)

  Helper method to compute the total fuel consumption and emissions over the Model run.
- void __computeNetPresentCost (void)

  Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs. Also tallies up total dispatch and discharge.
- void __computeLevellizedCostOfEnergy (void)

  Helper method to compute the overall levellized cost of energy, for the Model run, from the asset-wise levellized costs of energy.
- void __computeEconomics (void)

  Helper method to compute key economic metrics for the Model run.
- void __writeSummary (std::string)

  Helper method to write summary results for Model.
- void __writeTimeSeries (std::string, int=-1)

  Helper method to write time series results for Model.

4.15.1 Detailed Description

A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Model() [1/2]

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

Constructor (dummy) for the Model class.

```
573 {
574     return;
575 }   /* Model() */
```

4.15.2.2 Model() [2/2]

```
Model::Model (
              ModelInputs model_inputs )
```

Constructor (intended) for the Model class.

Parameters

model_inputs	A structure of Model constructor inputs.

```
592 {
593     //  1. check inputs
594     this->__checkInputs(model_inputs);
595 
596     //  2. read in electrical load data
597     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
598 
599     //  3. set control mode
600     this->controller.setControlMode(model_inputs.control_mode);
601 
602     //  4. set public attributes
603     this->total_fuel_consumed_L = 0;
604     this->net_present_cost = 0;
605     this->total_dispatch_discharge_kWh = 0;
606     this->total_renewable_dispatch_kWh = 0;
607     this->levellized_cost_of_energy_kWh = 0;
608 
609     return;
610 }   /* Model() */
```

4.15.2.3 ~Model()

```
Model::~Model (
              void  )
```

Destructor for the Model class.

```
1129 {
1130     this->clear();
1131     return;
1132 }   /* ~Model() */
```

4.15.3 Member Function Documentation

4.15.3.1 __checkInputs()

```
void Model::__checkInputs (
            ModelInputs model_inputs )  [private]
```

Helper method (private) to check inputs to the Model constructor.

Parameters

model_inputs	A structure of Model constructor inputs.

```
40 {
41     //  1. check path_2_electrical_load_time_series
42     if (model_inputs.path_2_electrical_load_time_series.empty()) {
43         std::string error_str = "ERROR:  Model()  path_2_electrical_load_time_series ";
44         error_str += "cannot be empty";
45 
46         #ifdef _WIN32
47             std::cout « error_str « std::endl;
48         #endif
49 
50         throw std::invalid_argument(error_str);
51     }
52 
53     return;
54 }   /* __checkInputs() */
```

4.15.3.2 __computeEconomics()

```
void Model::__computeEconomics (
            void  )  [private]
```

Helper method to compute key economic metrics for the Model run.

```
240 {
241     this->__computeNetPresentCost();
242     this->__computeLevellizedCostOfEnergy();
243 
244     return;
245 }   /* __computeEconomics() */
```

4.15.3.3 __computeFuelAndEmissions()

```
void Model::__computeFuelAndEmissions (
            void  )  [private]
```

Helper method to compute the total fuel consumption and emissions over the Model run.

```
70 {
71     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
72         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
73 
74         this->total_fuel_consumed_L +=
75             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
76 
77         this->total_emissions.CO2_kg +=
78             this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
79 
80         this->total_emissions.CO_kg +=
81             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
82 
83         this->total_emissions.NOx_kg +=
84             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
85 
86         this->total_emissions.SOx_kg +=
```

```
87              this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
88
89          this->total_emissions.CH4_kg +=
90              this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
91
92          this->total_emissions.PM_kg +=
93              this->combustion_ptr_vec[i]->total_emissions.PM_kg;
94      }
95
96      return;
97  }   /* __computeFuelAndEmissions() */
```

4.15.3.4 __computeLevellizedCostOfEnergy()

```
void Model::__computeLevellizedCostOfEnergy (
              void  )  [private]
```

Helper method to compute the overall levellized cost of energy, for the Model run, from the asset-wise levellized costs of energy.

```
187 {
188     //  1. account for Combustion economics in levellized cost of energy
189     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
190         this->levellized_cost_of_energy_kWh +=
191             (
192                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
193                 this->combustion_ptr_vec[i]->total_dispatch_kWh
194             ) / this->total_dispatch_discharge_kWh;
195     }
196
197     //  2. account for Noncombustion economics in levellized cost of energy
198     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
199         this->levellized_cost_of_energy_kWh +=
200             (
201                 this->noncombustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
202                 this->noncombustion_ptr_vec[i]->total_dispatch_kWh
203             ) / this->total_dispatch_discharge_kWh;
204     }
205
206     //  3. account for Renewable economics in levellized cost of energy
207     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
208         this->levellized_cost_of_energy_kWh +=
209             (
210                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
211                 this->renewable_ptr_vec[i]->total_dispatch_kWh
212             ) / this->total_dispatch_discharge_kWh;
213     }
214
215     //  4. account for Storage economics in levellized cost of energy
216     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
217         this->levellized_cost_of_energy_kWh +=
218             (
219                 this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
220                 this->storage_ptr_vec[i]->total_discharge_kWh
221             ) / this->total_dispatch_discharge_kWh;
222     }
223
224     return;
225 }   /* __computeLevellizedCostOfEnergy() */
```

4.15.3.5 __computeNetPresentCost()

```
void Model::__computeNetPresentCost (
              void  )  [private]
```

Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs. Also tallies up total dispatch and discharge.

```
114 {
115     //  1. account for Combustion economics in net present cost
```

```
116     //     increment total dispatch
117     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
118         this->combustion_ptr_vec[i]->computeEconomics(
119             &(this->electrical_load.time_vec_hrs)
120         );
121 
122         this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
123 
124         this->total_dispatch_discharge_kWh +=
125             this->combustion_ptr_vec[i]->total_dispatch_kWh;
126     }
127 
128     //  2. account for Noncombustion economics in net present cost
129     //     increment total dispatch
130     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
131         this->noncombustion_ptr_vec[i]->computeEconomics(
132             &(this->electrical_load.time_vec_hrs)
133         );
134 
135         this->net_present_cost += this->noncombustion_ptr_vec[i]->net_present_cost;
136 
137         this->total_dispatch_discharge_kWh +=
138             this->noncombustion_ptr_vec[i]->total_dispatch_kWh;
139     }
140 
141     //  3. account for Renewable economics in net present cost,
142     //     increment total dispatch
143     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
144         this->renewable_ptr_vec[i]->computeEconomics(
145             &(this->electrical_load.time_vec_hrs)
146         );
147 
148         this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
149 
150         this->total_dispatch_discharge_kWh +=
151             this->renewable_ptr_vec[i]->total_dispatch_kWh;
152 
153         this->total_renewable_dispatch_kWh +=
154             this->renewable_ptr_vec[i]->total_dispatch_kWh;
155     }
156 
157     //  4. account for Storage economics in net present cost
158     //     increment total dispatch
159     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
160         this->storage_ptr_vec[i]->computeEconomics(
161             &(this->electrical_load.time_vec_hrs)
162         );
163 
164         this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
165 
166         this->total_dispatch_discharge_kWh +=
167             this->storage_ptr_vec[i]->total_discharge_kWh;
168     }
169 
170     return;
171 }   /* __computeNetPresentCost() */
```

4.15.3.6 __writeSummary()

```
void Model::__writeSummary (
            std::string write_path )  [private]
```

Helper method to write summary results for Model.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

```
263 {
264     //  1. create subdirectory
265     write_path += "Model/";
266     std::filesystem::create_directory(write_path);
267 
```

```
268     //  2. create filestream
269     write_path += "summary_results.md";
270     std::ofstream ofs;
271     ofs.open(write_path, std::ofstream::out);
272 
273     //  3. write summary results (markdown)
274     ofs « "# Model Summary Results\n";
275     ofs « "\n--------\n\n";
276 
277     //  3.1. ElectricalLoad
278     ofs « "## Electrical Load\n";
279     ofs « "\n";
280     ofs « "Path: " «
281         this->electrical_load.path_2_electrical_load_time_series « "  \n";
282     ofs « "Data Points: " « this->electrical_load.n_points « "  \n";
283     ofs « "Years: " « this->electrical_load.n_years « "  \n";
284     ofs « "Min: " « this->electrical_load.min_load_kW « " kW  \n";
285     ofs « "Mean: " « this->electrical_load.mean_load_kW « " kW  \n";
286     ofs « "Max: " « this->electrical_load.max_load_kW « " kW  \n";
287     ofs « "\n--------\n\n";
288 
289     //  3.2. Controller
290     ofs « "## Controller\n";
291     ofs « "\n";
292     ofs « "Control Mode: " « this->controller.control_string « "  \n";
293     ofs « "\n--------\n\n";
294 
295     //  3.3. Resources (1D)
296     ofs « "## 1D Renewable Resources\n";
297     ofs « "\n";
298 
299     std::map<int, std::string>::iterator string_map_1D_iter =
300         this->resources.string_map_1D.begin();
301     std::map<int, std::string>::iterator path_map_1D_iter =
302         this->resources.path_map_1D.begin();
303 
304     while (
305         string_map_1D_iter != this->resources.string_map_1D.end() and
306         path_map_1D_iter != this->resources.path_map_1D.end()
307     ) {
308         ofs « "Resource Key: " « string_map_1D_iter->first « "  \n";
309         ofs « "Type: " « string_map_1D_iter->second « "  \n";
310         ofs « "Path: " « path_map_1D_iter->second « "  \n";
311         ofs « "\n";
312 
313         string_map_1D_iter++;
314         path_map_1D_iter++;
315     }
316 
317     ofs « "\n--------\n\n";
318 
319     //  3.4. Resources (2D)
320     ofs « "## 2D Renewable Resources\n";
321     ofs « "\n";
322 
323     std::map<int, std::string>::iterator string_map_2D_iter =
324         this->resources.string_map_2D.begin();
325     std::map<int, std::string>::iterator path_map_2D_iter =
326         this->resources.path_map_2D.begin();
327 
328     while (
329         string_map_2D_iter != this->resources.string_map_2D.end() and
330         path_map_2D_iter != this->resources.path_map_2D.end()
331     ) {
332         ofs « "Resource Key: " « string_map_2D_iter->first « "  \n";
333         ofs « "Type: " « string_map_2D_iter->second « "  \n";
334         ofs « "Path: " « path_map_2D_iter->second « "  \n";
335         ofs « "\n";
336 
337         string_map_2D_iter++;
338         path_map_2D_iter++;
339     }
340 
341     ofs « "\n--------\n\n";
342 
343     //  3.5. Combustion
344     ofs « "## Combustion Assets\n";
345     ofs « "\n";
346 
347     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
348         ofs « "Asset Index: " « i « "  \n";
349         ofs « "Type: " « this->combustion_ptr_vec[i]->type_str « "  \n";
350         ofs « "Capacity: " « this->combustion_ptr_vec[i]->capacity_kW « " kW  \n";
351         ofs « "\n";
352     }
353 
354     ofs « "\n--------\n\n";
```

```
355
356     //  3.6. Noncombustion
357     ofs « "## Noncombustion Assets\n";
358     ofs « "\n";
359
360     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
361         ofs « "Asset Index: " « i « "  \n";
362         ofs « "Type: " « this->noncombustion_ptr_vec[i]->type_str « "  \n";
363         ofs « "Capacity: " « this->noncombustion_ptr_vec[i]->capacity_kW « " kW  \n";
364
365         if (this->noncombustion_ptr_vec[i]->type == NoncombustionType :: HYDRO) {
366             ofs « "Reservoir Capacity: " «
367                 ((Hydro*)(this->noncombustion_ptr_vec[i]))->reservoir_capacity_m3 «
368                 " m3  \n";
369         }
370
371         ofs « "\n";
372     }
373
374     ofs « "\n--------\n\n";
375
376     //  3.7. Renewable
377     ofs « "## Renewable Assets\n";
378     ofs « "\n";
379
380     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
381         ofs « "Asset Index: " « i « "  \n";
382         ofs « "Type: " « this->renewable_ptr_vec[i]->type_str « "  \n";
383         ofs « "Capacity: " « this->renewable_ptr_vec[i]->capacity_kW « " kW  \n";
384         ofs « "\n";
385     }
386
387     ofs « "\n--------\n\n";
388
389     //  3.8. Storage
390     ofs « "## Storage Assets\n";
391     ofs « "\n";
392
393     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
394         ofs « "Asset Index: " « i « "  \n";
395         ofs « "Type: " « this->storage_ptr_vec[i]->type_str « "  \n";
396         ofs « "Power Capacity: " « this->storage_ptr_vec[i]->power_capacity_kW
397             « " kW  \n";
398         ofs « "Energy Capacity: " « this->storage_ptr_vec[i]->energy_capacity_kWh
399             « " kWh  \n";
400         ofs « "\n";
401     }
402
403     ofs « "\n--------\n\n";
404
405     //  3.9. Model Results
406     ofs « "## Results\n";
407     ofs « "\n";
408
409     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
410     ofs « "\n";
411
412     ofs « "Total Dispatch + Discharge: " « this->total_dispatch_discharge_kWh
413         « " kWh  \n";
414
415     ofs « "Renewable Penetration: "
416         « this->total_renewable_dispatch_kWh / this->total_dispatch_discharge_kWh
417         « "  \n";
418     ofs « "\n";
419
420     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
421         « " per kWh dispatched/discharged  \n";
422     ofs « "\n";
423
424     ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
425         « "(Annual Average: " «
426             this->total_fuel_consumed_L / this->electrical_load.n_years
427         « " L/yr)  \n";
428     ofs « "\n";
429
430     ofs « "Total Carbon Dioxide (CO2) Emissions: " «
431         this->total_emissions.CO2_kg « " kg "
432         « "(Annual Average: " «
433             this->total_emissions.CO2_kg / this->electrical_load.n_years
434         « " kg/yr)  \n";
435
436     ofs « "Total Carbon Monoxide (CO) Emissions: " «
437         this->total_emissions.CO_kg « " kg "
438         « "(Annual Average: " «
439             this->total_emissions.CO_kg / this->electrical_load.n_years
440         « " kg/yr)  \n";
441
```

```
442     ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
443         this->total_emissions.NOx_kg « " kg "
444         « "(Annual Average: " «
445             this->total_emissions.NOx_kg / this->electrical_load.n_years
446         « " kg/yr)  \n";
447
448     ofs « "Total Sulfur Oxides (SOx) Emissions: " «
449         this->total_emissions.SOx_kg « " kg "
450         « "(Annual Average: " «
451             this->total_emissions.SOx_kg / this->electrical_load.n_years
452         « " kg/yr)  \n";
453
454     ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
455         « "(Annual Average: " «
456             this->total_emissions.CH4_kg / this->electrical_load.n_years
457         « " kg/yr)  \n";
458
459     ofs « "Total Particulate Matter (PM) Emissions: " «
460         this->total_emissions.PM_kg « " kg "
461         « "(Annual Average: " «
462             this->total_emissions.PM_kg / this->electrical_load.n_years
463         « " kg/yr)  \n";
464
465     ofs « "\n--------\n\n";
466
467     ofs.close();
468     return;
469 }   /* __writeSummary() */
```

4.15.3.7 __writeTimeSeries()

```
void Model::__writeTimeSeries (
              std::string write_path,
              int max_lines = -1 )  [private]
```

Helper method to write time series results for Model.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
max_lines	The maximum number of lines of output to write.

```
489 {
490     //  1. create filestream
491     write_path += "Model/time_series_results.csv";
492     std::ofstream ofs;
493     ofs.open(write_path, std::ofstream::out);
494
495     //  2. write time series results header (comma separated value)
496     ofs « "Time (since start of data) [hrs],";
497     ofs « "Electrical Load [kW],";
498     ofs « "Net Load [kW],";
499     ofs « "Missed Load [kW],";
500
501     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
502         ofs « this->renewable_ptr_vec[i]->capacity_kW « " kW "
503             « this->renewable_ptr_vec[i]->type_str « " Dispatch [kW],";
504     }
505
506     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
507         ofs « this->storage_ptr_vec[i]->power_capacity_kW « " kW "
508             « this->storage_ptr_vec[i]->energy_capacity_kWh « " kWh "
509             « this->storage_ptr_vec[i]->type_str « " Discharge [kW],";
510     }
511
512     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
513         ofs « this->noncombustion_ptr_vec[i]->capacity_kW « " kW "
514             « this->noncombustion_ptr_vec[i]->type_str « " Dispatch [kW],";
515     }
516
517     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
518         ofs « this->combustion_ptr_vec[i]->capacity_kW « " kW "
```

```
519             « this->combustion_ptr_vec[i]->type_str « " Dispatch [kW],";
520     }
521
522     ofs « "\n";
523
524     //  3. write time series results values (comma separated value)
525     for (int i = 0; i < max_lines; i++) {
526         //  3.1. load values
527         ofs « this->electrical_load.time_vec_hrs[i] « ",";
528         ofs « this->electrical_load.load_vec_kW[i] « ",";
529         ofs « this->controller.net_load_vec_kW[i] « ",";
530         ofs « this->controller.missed_load_vec_kW[i] « ",";
531
532         //  3.2. asset-wise dispatch/discharge
533         for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
534             ofs « this->renewable_ptr_vec[j]->dispatch_vec_kW[i] « ",";
535         }
536
537         for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
538             ofs « this->storage_ptr_vec[j]->discharging_power_vec_kW[i] « ",";
539         }
540
541         for (size_t j = 0; j < this->noncombustion_ptr_vec.size(); j++) {
542             ofs « this->noncombustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
543         }
544
545         for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
546             ofs « this->combustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
547         }
548
549         ofs « "\n";
550     }
551
552     ofs.close();
553     return;
554 }   /* __writeTimeSeries() */
```

4.15.3.8 addDiesel()

```
void Model::addDiesel (
            DieselInputs diesel_inputs )
```

Method to add a Diesel asset to the Model.

Parameters

diesel_inputs	A structure of Diesel constructor inputs.

```
627 {
628     Combustion* diesel_ptr = new Diesel(
629         this->electrical_load.n_points,
630         this->electrical_load.n_years,
631         diesel_inputs,
632         &(this->electrical_load.time_vec_hrs)
633     );
634
635     this->combustion_ptr_vec.push_back(diesel_ptr);
636
637     return;
638 }   /* addDiesel() */
```

4.15.3.9 addHydro()

```
void Model::addHydro (
            HydroInputs hydro_inputs )
```

Method to add a Hydro asset to the Model.

Parameters

hydro_inputs	A structure of Hydro constructor inputs.

```
731 {
732     Noncombustion* hydro_ptr = new Hydro(
733         this->electrical_load.n_points,
734         this->electrical_load.n_years,
735         hydro_inputs,
736         &(this->electrical_load.time_vec_hrs)
737     );
738 
739     this->noncombustion_ptr_vec.push_back(hydro_ptr);
740 
741     return;
742 }   /* addHydro() */
```

4.15.3.10 addLiIon()

```
void Model::addLiIon (
             LiIonInputs liion_inputs )
```

Method to add a LiIon asset to the Model.

Parameters

liion_inputs	A structure of LiIon constructor inputs.

```
871 {
872     Storage* liion_ptr = new LiIon(
873         this->electrical_load.n_points,
874         this->electrical_load.n_years,
875         liion_inputs
876     );
877 
878     this->storage_ptr_vec.push_back(liion_ptr);
879 
880     return;
881 }   /* addLiIon() */
```

4.15.3.11 addResource() [1/2]

```
void Model::addResource (
             NoncombustionType noncombustion_type,
             std::string path_2_resource_data,
             int resource_key )
```

A method to add a renewable resource time series to the Model.

Parameters

noncombustion_type	The type of renewable resource being added to the Model.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.

```
667 {
668     resources.addResource(
669         noncombustion_type,
670         path_2_resource_data,
671         resource_key,
672         &(this->electrical_load)
673     );
674 
675     return;
676 }   /* addResource() */
```

4.15.3.12 addResource() [2/2]

```
void Model::addResource (
              RenewableType renewable_type,
              std::string path_2_resource_data,
              int resource_key )
```

A method to add a renewable resource time series to the Model.

Parameters

renewable_type	The type of renewable resource being added to the Model.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.

```
705 {
706     resources.addResource(
707         renewable_type,
708         path_2_resource_data,
709         resource_key,
710         &(this->electrical_load)
711     );
712 
713     return;
714 }   /* addResource() */
```

4.15.3.13 addSolar()

```
void Model::addSolar (
              SolarInputs solar_inputs )
```

Method to add a Solar asset to the Model.

Parameters

solar_inputs	A structure of Solar constructor inputs.

```
759 {
760     Renewable* solar_ptr = new Solar(
761         this->electrical_load.n_points,
762         this->electrical_load.n_years,
763         solar_inputs,
764         &(this->electrical_load.time_vec_hrs)
765     );
766 
767     this->renewable_ptr_vec.push_back(solar_ptr);
```

```
768
769     return;
770 }   /* addSolar() */
```

4.15.3.14 addTidal()

```
void Model::addTidal (
            TidalInputs tidal_inputs )
```

Method to add a Tidal asset to the Model.

Parameters

tidal_inputs	A structure of Tidal constructor inputs.

```
787 {
788     Renewable* tidal_ptr = new Tidal(
789         this->electrical_load.n_points,
790         this->electrical_load.n_years,
791         tidal_inputs,
792         &(this->electrical_load.time_vec_hrs)
793     );
794
795     this->renewable_ptr_vec.push_back(tidal_ptr);
796
797     return;
798 }   /* addTidal() */
```

4.15.3.15 addWave()

```
void Model::addWave (
            WaveInputs wave_inputs )
```

Method to add a Wave asset to the Model.

Parameters

wave_inputs	A structure of Wave constructor inputs.

```
815 {
816     Renewable* wave_ptr = new Wave(
817         this->electrical_load.n_points,
818         this->electrical_load.n_years,
819         wave_inputs,
820         &(this->electrical_load.time_vec_hrs)
821     );
822
823     this->renewable_ptr_vec.push_back(wave_ptr);
824
825     return;
826 }   /* addWave() */
```

4.15.3.16 addWind()

```
void Model::addWind (
            WindInputs wind_inputs )
```

Method to add a Wind asset to the Model.

Parameters

wind_inputs	A structure of Wind constructor inputs.

```
843 {
844     Renewable* wind_ptr = new Wind(
845         this->electrical_load.n_points,
846         this->electrical_load.n_years,
847         wind_inputs,
848         &(this->electrical_load.time_vec_hrs)
849     );
850 
851     this->renewable_ptr_vec.push_back(wind_ptr);
852 
853     return;
854 }   /* addWind() */
```

4.15.3.17 clear()

```
void Model::clear (
              void  )
```

Method to clear all attributes of the Model object.

```
998 {
999     //  1. reset
1000     this->reset();
1001 
1002     //  2. clear components
1003     controller.clear();
1004     electrical_load.clear();
1005     resources.clear();
1006 
1007     return;
1008 }   /* clear() */
```

4.15.3.18 reset()

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

Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors and resets select Model attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

```
940 {
941     //  1. clear combustion_ptr_vec
942     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
943         delete this->combustion_ptr_vec[i];
944     }
945     this->combustion_ptr_vec.clear();
946 
947     //  2. clear noncombustion_ptr_vec
948     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
949         delete this->noncombustion_ptr_vec[i];
950     }
951     this->noncombustion_ptr_vec.clear();
952 
953     //  3. clear renewable_ptr_vec
954     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
955         delete this->renewable_ptr_vec[i];
956     }
957     this->renewable_ptr_vec.clear();
958 
959     //  4. clear storage_ptr_vec
```

```
960     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
961         delete this->storage_ptr_vec[i];
962     }
963     this->storage_ptr_vec.clear();
964 
965     //  5. reset components and attributes
966     this->controller.clear();
967 
968     this->total_fuel_consumed_L = 0;
969 
970     this->total_emissions.CO2_kg = 0;
971     this->total_emissions.CO_kg = 0;
972     this->total_emissions.NOx_kg = 0;
973     this->total_emissions.SOx_kg = 0;
974     this->total_emissions.CH4_kg = 0;
975     this->total_emissions.PM_kg = 0;
976 
977     this->net_present_cost = 0;
978     this->total_dispatch_discharge_kWh = 0;
979     this->total_renewable_dispatch_kWh = 0;
980     this->levellized_cost_of_energy_kWh = 0;
981 
982     return;
983 }   /* reset() */
```

4.15.3.19 run()

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

A method to run the Model.

```
896 {
897     // 1. init Controller
898     this->controller.init(
899         &(this->electrical_load),
900         &(this->renewable_ptr_vec),
901         &(this->resources),
902         &(this->combustion_ptr_vec)
903     );
904 
905     //  2. apply dispatch control
906     this->controller.applyDispatchControl(
907         &(this->electrical_load),
908         &(this->resources),
909         &(this->combustion_ptr_vec),
910         &(this->noncombustion_ptr_vec),
911         &(this->renewable_ptr_vec),
912         &(this->storage_ptr_vec)
913     );
914 
915     //  3. compute total fuel consumption and emissions
916     this->__computeFuelAndEmissions();
917 
918     //  4. compute key economic metrics
919     this->__computeEconomics();
920 
921     return;
922 }   /* run() */
```

4.15.3.20 writeResults()

```
void Model::writeResults (
              std::string write_path,
              int max_lines = -1 )
```

Method which writes Model results to an output directory. Also calls out to writeResults() for each contained asset.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
max_lines	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```
1036 {
1037     //  1. handle sentinel
1038     if (max_lines < 0) {
1039         max_lines = this->electrical_load.n_points;
1040     }
1041
1042     //  2. check for pre-existing, warn (and remove), then create
1043     if (write_path.back() != '/') {
1044         write_path += '/';
1045     }
1046
1047     if (std::filesystem::is_directory(write_path)) {
1048         std::string warning_str = "WARNING:  Model::writeResults():  ";
1049         warning_str += write_path;
1050         warning_str += " already exists, contents will be overwritten!";
1051
1052         std::cout « warning_str « std::endl;
1053
1054         std::filesystem::remove_all(write_path);
1055     }
1056
1057     std::filesystem::create_directory(write_path);
1058
1059     //  3. write summary
1060     this->__writeSummary(write_path);
1061
1062     //  4. write time series
1063     if (max_lines > this->electrical_load.n_points) {
1064         max_lines = this->electrical_load.n_points;
1065     }
1066
1067     if (max_lines > 0) {
1068         this->__writeTimeSeries(write_path, max_lines);
1069     }
1070
1071     //  5. call out to Combustion :: writeResults()
1072     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
1073         this->combustion_ptr_vec[i]->writeResults(
1074             write_path,
1075             &(this->electrical_load.time_vec_hrs),
1076             i,
1077             max_lines
1078         );
1079     }
1080
1081     //  6. call out to Noncombustion :: writeResults()
1082     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
1083         this->noncombustion_ptr_vec[i]->writeResults(
1084             write_path,
1085             &(this->electrical_load.time_vec_hrs),
1086             i,
1087             max_lines
1088         );
1089     }
1090
1091     //  7. call out to Renewable :: writeResults()
1092     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
1093         this->renewable_ptr_vec[i]->writeResults(
1094             write_path,
1095             &(this->electrical_load.time_vec_hrs),
1096             &(this->resources.resource_map_1D),
1097             &(this->resources.resource_map_2D),
1098             i,
1099             max_lines
1100         );
1101     }
1102
1103     //  8. call out to Storage :: writeResults()
1104     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
1105         this->storage_ptr_vec[i]->writeResults(
1106             write_path,
1107             &(this->electrical_load.time_vec_hrs),
1108             i,
1109             max_lines
1110         );
1111     }
1112
```

```
1113     return;
1114 }   /* writeResults() */
```

4.15.4 Member Data Documentation

4.15.4.1 combustion_ptr_vec

```
std::vector<Combustion*> Model::combustion_ptr_vec
```

A vector of pointers to the various Combustion assets in the Model.

4.15.4.2 controller

```
Controller Model::controller
```

Controller component of Model.

4.15.4.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

ElectricalLoad component of Model.

4.15.4.4 levellized_cost_of_energy_kWh

```
double Model::levellized_cost_of_energy_kWh
```

The levellized cost of energy, per unit energy dispatched/discharged, of the Model [1/kWh] (undefined currency).

4.15.4.5 net_present_cost

```
double Model::net_present_cost
```

The net present cost of the Model (undefined currency).

4.15.4.6 noncombustion_ptr_vec

```
std::vector<Noncombustion*> Model::noncombustion_ptr_vec
```

A vector of pointers to the various Noncombustion assets in the Model.

4.15.4.7 renewable_ptr_vec

```
std::vector<Renewable*> Model::renewable_ptr_vec
```

A vector of pointers to the various Renewable assets in the Model.

4.15.4.8 resources

```
Resources Model::resources
```

Resources component of Model.

4.15.4.9 storage_ptr_vec

```
std::vector<Storage*> Model::storage_ptr_vec
```

A vector of pointers to the various Storage assets in the Model.

4.15.4.10 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

The total energy dispatched/discharged [kWh] over the Model run.

4.15.4.11 total_emissions

```
Emissions Model::total_emissions
```

An Emissions structure for holding total emissions [kg].

4.15.4.12 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

The total fuel consumed [L] over a model run.

4.15.4.13 total_renewable_dispatch_kWh

```
double Model::total_renewable_dispatch_kWh
```

The total energy dispatched [kWh] by all renewable assets over the Model run.

The documentation for this class was generated from the following files:

- header/Model.h
- source/Model.cpp

4.16 ModelInputs Struct Reference

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).

```
#include <Model.h>
```

Public Attributes

- std::string path_2_electrical_load_time_series = ""

  A string defining the path (either relative or absolute) to the given electrical load time series.
- ControlMode control_mode = ControlMode :: LOAD_FOLLOWING

  The control mode to be applied by the Controller object.

4.16.1 Detailed Description

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).

4.16.2 Member Data Documentation

4.16.2.1 control_mode

```
ControlMode ModelInputs::control_mode = ControlMode ::  LOAD_FOLLOWING
```

The control mode to be applied by the Controller object.

4.16.2.2 path_2_electrical_load_time_series

```
std::string ModelInputs::path_2_electrical_load_time_series = ""
```

A string defining the path (either relative or absolute) to the given electrical load time series.

The documentation for this struct was generated from the following file:

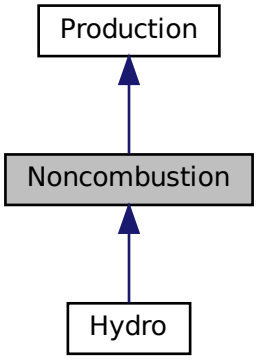
- header/Model.h

4.17 Noncombustion Class Reference

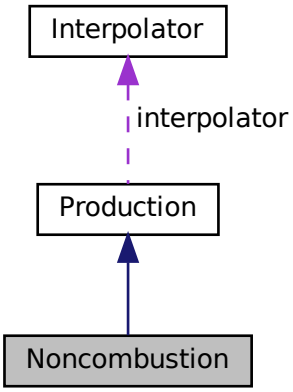
The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

```
#include <Noncombustion.h>
```

Inheritance diagram for Noncombustion:



Collaboration diagram for Noncombustion:



Public Member Functions

- Noncombustion (void)

  Constructor (dummy) for the Noncombustion class.
- Noncombustion (int, double, NoncombustionInputs, std::vector< double > *)

  Constructor (intended) for the Noncombustion class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeEconomics (std::vector< double > *)

  Helper method to compute key economic metrics for the Model run.
- virtual double requestProductionkW (int, double, double)
- virtual double requestProductionkW (int, double, double, double)
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual double commit (int, double, double, double, double)
- void writeResults (std::string, std::vector< double > *, int, int=-1)

  Method which writes Noncombustion results to an output directory.
- virtual ~Noncombustion (void)

  Destructor for the Noncombustion class.

Public Attributes

- NoncombustionType type

  The type (NoncombustionType) of the asset.
- int resource_key

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

Private Member Functions

- void __checkInputs (NoncombustionInputs)

  Helper method to check inputs to the Noncombustion constructor.
- void __handleStartStop (int, double, double)

  Helper method to handle the starting/stopping of the Noncombustion asset.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

4.17.1 Detailed Description

The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 Noncombustion() [1/2]

```
Noncombustion::Noncombustion (
            void  )
```

Constructor (dummy) for the Noncombustion class.

```
103 {
104     return;
105 }   /* Noncombustion() */
```

4.17.2.2 Noncombustion() [2/2]

```
Noncombustion::Noncombustion (
            int n_points,
            double n_years,
            NoncombustionInputs noncombustion_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Noncombustion class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
noncombustion_inputs	A structure of Noncombustion constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
137   :
138 Production(
139     n_points,
```

```
140     n_years,
141     noncombustion_inputs.production_inputs,
142     time_vec_hrs_ptr
143 )
144 {
145     //  1. check inputs
146     this->__checkInputs(noncombustion_inputs);
147 
148     //  2. set attributes
149     //...
150 
151     //  3. construction print
152     if (this->print_flag) {
153         std::cout « "Noncombustion object constructed at " « this « std::endl;
154     }
155 
156     return;
157 }   /* Noncombustion() */
```

4.17.2.3 ~Noncombustion()

```
Noncombustion::~Noncombustion (
             void  )  [virtual]
```

Destructor for the Noncombustion class.

```
348 {
349     //  1. destruction print
350     if (this->print_flag) {
351         std::cout « "Noncombustion object at " « this « " destroyed" « std::endl;
352     }
353 
354     return;
355 }   /* ~Noncombustion() */
```

4.17.3 Member Function Documentation

4.17.3.1 __checkInputs()

```
void Noncombustion::__checkInputs (
             NoncombustionInputs noncombustion_inputs )  [private]
```

Helper method to check inputs to the Noncombustion constructor.

Parameters

noncombustion_inputs	A structure of Noncombustion constructor inputs.

```
40 {
41     //...
42 
43     return;
44 }   /* __checkInputs() */
```

4.17.3.2 __handleStartStop()

```
void Noncombustion::__handleStartStop (
```

```
            int timestep,
            double dt_hrs,
            double production_kW )  [private]
```

Helper method to handle the starting/stopping of the Noncombustion asset.

```
67 {
68     if (this->is_running) {
69         // handle stopping
70         if (production_kW <= 0) {
71             this->is_running = false;
72         }
73     }
74 
75     else {
76         // handle starting
77         if (production_kW > 0) {
78             this->is_running = true;
79             this->n_starts++;
80         }
81     }
82 
83     return;
84 }   /* __handleStartStop() */
```

4.17.3.3 __writeSummary()

```
virtual void Noncombustion::__writeSummary (
            std::string  )  [inline], [private], [virtual]
```

Reimplemented in Hydro.

```
70 {return;}
```

4.17.3.4 __writeTimeSeries()

```
virtual void Noncombustion::__writeTimeSeries (
            std::string ,
            std::vector< double > * ,
            int  = -1 )  [inline], [private], [virtual]
```

Reimplemented in Hydro.

```
75          {return;}
```

4.17.3.5 commit() [1/2]

```
double Noncombustion::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Production.

```
243 {
244     //  1. handle start/stop
245     this->__handleStartStop(timestep, dt_hrs, production_kW);
246 
247     //  2. invoke base class method
248     load_kW = Production :: commit(
249         timestep,
250         dt_hrs,
251         production_kW,
252         load_kW
253     );
254 
255 
256     //...
257 
258     return load_kW;
259 }   /* commit() */
```

4.17.3.6 commit() [2/2]

```
virtual double Noncombustion::commit (
             int ,
             double ,
             double ,
             double ,
             double  )  [inline], [virtual]
```

Reimplemented in Hydro.

```
96 {return 0;}
```

4.17.3.7 computeEconomics()

```
void Noncombustion::computeEconomics (
             std::vector< double > * time_vec_hrs_ptr )  [virtual]
```

Helper method to compute key economic metrics for the Model run.

Ref: HOMER [2023b]

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
202 {
203     //  1. invoke base class method
204     Production :: computeEconomics(time_vec_hrs_ptr);
205 
206     return;
207 }   /* computeEconomics() */
```

4.17.3.8 handleReplacement()

```
void Noncombustion::handleReplacement (
              int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Hydro.

```
175 {
176     //  1. reset attributes
177     //...
178 
179     //  2. invoke base class method
180     Production :: handleReplacement(timestep);
181 
182     return;
183 }   /* __handleReplacement() */
```

4.17.3.9 requestProductionkW() [1/2]

```
virtual double Noncombustion::requestProductionkW (
              int ,
              double ,
              double  )  [inline], [virtual]
92 {return 0;}
```

4.17.3.10 requestProductionkW() [2/2]

```
virtual double Noncombustion::requestProductionkW (
              int ,
              double ,
              double ,
              double  )  [inline], [virtual]
```

Reimplemented in Hydro.

```
93 {return 0;}
```

4.17.3.11 writeResults()

```
void Noncombustion::writeResults (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            int combustion_index,
            int max_lines = -1 )
```

Method which writes Noncombustion results to an output directory.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
noncombustion_index	An integer which corresponds to the index of the Noncombustion asset in the Model.
max_lines	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```
295 {
296     //  1. handle sentinel
297     if (max_lines < 0) {
298         max_lines = this->n_points;
299     }
300 
301     //  2. create subdirectories
302     write_path += "Production/";
303     if (not std::filesystem::is_directory(write_path)) {
304         std::filesystem::create_directory(write_path);
305     }
306 
307     write_path += "Noncombustion/";
308     if (not std::filesystem::is_directory(write_path)) {
309         std::filesystem::create_directory(write_path);
310     }
311 
312     write_path += this->type_str;
313     write_path += "_";
314     write_path += std::to_string(int(ceil(this->capacity_kW)));
315     write_path += "kW_idx";
316     write_path += std::to_string(combustion_index);
317     write_path += "/";
318     std::filesystem::create_directory(write_path);
319 
320     //  3. write summary
321     this->__writeSummary(write_path);
322 
323     //  4. write time series
324     if (max_lines > this->n_points) {
325         max_lines = this->n_points;
326     }
327 
328     if (max_lines > 0) {
329         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
330     }
331 
332     return;
333 }   /* writeResults() */
```

4.17.4 Member Data Documentation

4.17.4.1 resource_key

```
int Noncombustion::resource_key
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

4.17.4.2 type

```
NoncombustionType Noncombustion::type
```

The type (NoncombustionType) of the asset.

The documentation for this class was generated from the following files:

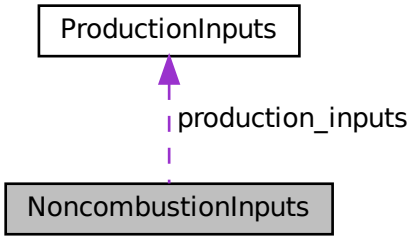
- header/Production/Noncombustion/Noncombustion.h
- source/Production/Noncombustion/Noncombustion.cpp

4.18 NoncombustionInputs Struct Reference

A structure which bundles the necessary inputs for the Noncombustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

```
#include <Noncombustion.h>
```

Collaboration diagram for NoncombustionInputs:



Public Attributes

- ProductionInputs production_inputs

  An encapsulated ProductionInputs instance.

4.18.1 Detailed Description

A structure which bundles the necessary inputs for the Noncombustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

4.18.2 Member Data Documentation

4.18.2.1 production_inputs

```
ProductionInputs NoncombustionInputs::production_inputs
```

An encapsulated ProductionInputs instance.

The documentation for this struct was generated from the following file:

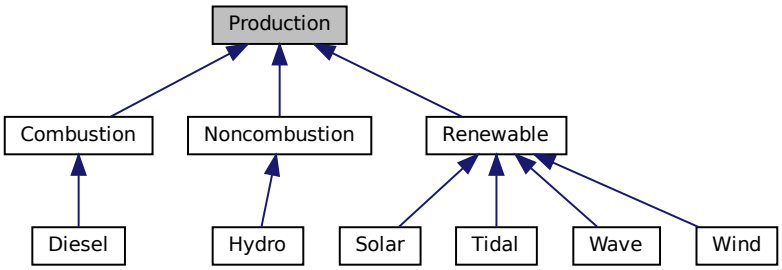
- header/Production/Noncombustion/Noncombustion.h

4.19 Production Class Reference

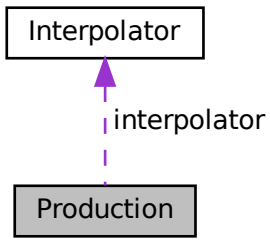
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

```
#include <Production.h>
```

Inheritance diagram for Production:



Collaboration diagram for Production:



Public Member Functions

- Production (void)

  Constructor (dummy) for the Production class.
- Production (int, double, ProductionInputs, std::vector< double > *)

  Constructor (intended) for the Production class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeRealDiscountAnnual (double, double)

  Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void computeEconomics (std::vector< double > *)

  Helper method to compute key economic metrics for the Model run.
- double getProductionkW (int)

  A method to simply fetch the normalized production at a particular point in the given normalized production time series, multiply by the rated capacity of the asset, and return.
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual ~Production (void)

  Destructor for the Production class.

Public Attributes

- Interpolator interpolator

  Interpolator component of Production.
- bool print_flag

  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_running

  A boolean which indicates whether or not the asset is running.
- bool is_sunk

  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- bool normalized_production_series_given

  A boolen which indicates whether or not a normalized production time series is given.
- int n_points

  The number of points in the modelling time series.
- int n_starts

  The number of times the asset has been started.
- int n_replacements

  The number of times the asset has been replaced.
- double n_years

  The number of years being modelled.
- double running_hours

  The number of hours for which the assset has been operating.
- double replace_running_hrs

  The number of running hours after which the asset must be replaced.
- double capacity_kW

  The rated production capacity [kW] of the asset.
- double nominal_inflation_annual

  The nominal, annual inflation rate to use in computing model economics.

- double nominal_discount_annual

  The nominal, annual discount rate to use in computing model economics.
- double real_discount_annual

  The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double capital_cost

  The capital cost of the asset (undefined currency).
- double operation_maintenance_cost_kWh

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.
- double net_present_cost

  The net present cost of this asset.
- double total_dispatch_kWh

  The total energy dispatched [kWh] over the Model run.
- double levellized_cost_of_energy_kWh

  The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatch.
- std::string type_str

  A string describing the type of the asset.
- std::string path_2_normalized_production_time_series

  A string defining the path (either relative or absolute) to the given normalized production time series.
- std::vector< bool > is_running_vec

  A boolean vector for tracking if the asset is running at a particular point in time.
- std::vector< double > normalized_production_vec

  A vector of normalizd production [ ] at each point in the modelling time series.
- std::vector< double > production_vec_kW

  A vector of production [kW] at each point in the modelling time series.
- std::vector< double > dispatch_vec_kW

  A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.
- std::vector< double > storage_vec_kW

  A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.
- std::vector< double > curtailment_vec_kW

  A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.
- std::vector< double > capital_cost_vec

  A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > operation_maintenance_cost_vec

  A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- void __checkInputs (int, double, ProductionInputs)

  Helper method to check inputs to the Production constructor.
- void __checkTimePoint (double, double)

  Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.
- void __throwLengthError (void)

  Helper method to throw data length error (if not the same as the given electrical load time series).
- void __checkNormalizedProduction (double)

Helper method to check that given data values are everywhere contained in the closed interval [0, 1]. A normalized production time series is expected, so this must be true everywhere.

- void __readNormalizedProductionData (std::vector< double > *)

  Helper method to read in a given time series of normalized production.

4.19.1 Detailed Description

The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Production() [1/2]

```
Production::Production (
            void  )
```

Constructor (dummy) for the Production class.

```
282 {
283     return;
284 }   /* Production() */
```

4.19.2.2 Production() [2/2]

```
Production::Production (
            int n_points,
            double n_years,
            ProductionInputs production_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Production class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
production_inputs	A structure of Production constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
317 {
318     //  1. check inputs
319     this->__checkInputs(n_points, n_years, production_inputs);
320 
321     //  2. set attributes
322     this->print_flag = production_inputs.print_flag;
323     this->is_running = false;
324     this->is_sunk = production_inputs.is_sunk;
325     this->normalized_production_series_given = false;
326 
```

```
327     this->n_points = n_points;
328     this->n_starts = 0;
329     this->n_replacements = 0;
330 
331     this->n_years = n_years;
332 
333     this->running_hours = 0;
334     this->replace_running_hrs = production_inputs.replace_running_hrs;
335 
336     this->capacity_kW = production_inputs.capacity_kW;
337 
338     this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
339     this->nominal_discount_annual = production_inputs.nominal_discount_annual;
340 
341     this->real_discount_annual = this->computeRealDiscountAnnual(
342         production_inputs.nominal_inflation_annual,
343         production_inputs.nominal_discount_annual
344     );
345 
346     this->capital_cost = 0;
347     this->operation_maintenance_cost_kWh = 0;
348     this->net_present_cost = 0;
349     this->total_dispatch_kWh = 0;
350     this->levellized_cost_of_energy_kWh = 0;
351 
352     this->path_2_normalized_production_time_series = "";
353 
354     this->is_running_vec.resize(this->n_points, 0);
355 
356     this->normalized_production_vec.resize(this->n_points, 0);
357     this->production_vec_kW.resize(this->n_points, 0);
358     this->dispatch_vec_kW.resize(this->n_points, 0);
359     this->storage_vec_kW.resize(this->n_points, 0);
360     this->curtailment_vec_kW.resize(this->n_points, 0);
361 
362     this->capital_cost_vec.resize(this->n_points, 0);
363     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
364 
365     //  3. read in normalized production time series (if given)
366     if (not production_inputs.path_2_normalized_production_time_series.empty()) {
367         this->normalized_production_series_given = true;
368 
369         this->path_2_normalized_production_time_series =
370             production_inputs.path_2_normalized_production_time_series;
371 
372         this->__readNormalizedProductionData(time_vec_hrs_ptr);
373     }
374 
375     //  4. construction print
376     if (this->print_flag) {
377         std::cout « "Production object constructed at " « this « std::endl;
378     }
379 
380     return;
381 }   /* Production() */
```

4.19.2.3 ~Production()

```
Production::~Production (
            void  )  [virtual]
```

Destructor for the Production class.

```
630 {
631     //  1. destruction print
632     if (this->print_flag) {
633         std::cout « "Production object at " « this « " destroyed" « std::endl;
634     }
635 
636     return;
637 }   /* ~Production() */
```

4.19.3 Member Function Documentation

4.19.3.1 __checkInputs()

```
void Production::__checkInputs (
            int n_points,
            double n_years,
            ProductionInputs production_inputs )  [private]
```

Helper method to check inputs to the Production constructor.

Parameters

n_points	The number of points in the modelling time series.
production_inputs	A structure of Production constructor inputs.

```
45 {
46     //  1. check n_points
47     if (n_points <= 0) {
48         std::string error_str = "ERROR:  Production():  n_points must be > 0";
49 
50         #ifdef _WIN32
51             std::cout « error_str « std::endl;
52         #endif
53 
54         throw std::invalid_argument(error_str);
55     }
56 
57     //  2. check n_years
58     if (n_years <= 0) {
59         std::string error_str = "ERROR:  Production():  n_years must be > 0";
60 
61         #ifdef _WIN32
62             std::cout « error_str « std::endl;
63         #endif
64 
65         throw std::invalid_argument(error_str);
66     }
67 
68     //  3. check capacity_kW
69     if (production_inputs.capacity_kW <= 0) {
70         std::string error_str = "ERROR:  Production():  ";
71         error_str += "ProductionInputs::capacity_kW must be > 0";
72 
73         #ifdef _WIN32
74             std::cout « error_str « std::endl;
75         #endif
76 
77         throw std::invalid_argument(error_str);
78     }
79 
80     //  4. check replace_running_hrs
81     if (production_inputs.replace_running_hrs <= 0) {
82         std::string error_str = "ERROR:  Production():  ";
83         error_str += "ProductionInputs::replace_running_hrs must be > 0";
84 
85         #ifdef _WIN32
86             std::cout « error_str « std::endl;
87         #endif
88 
89         throw std::invalid_argument(error_str);
90     }
91 
92     return;
93 }   /* __checkInputs() */
```

4.19.3.2 __checkNormalizedProduction()

```
void Production::__checkNormalizedProduction (
            double normalized_production )  [private]
```

Helper method to check that given data values are everywhere contained in the closed interval [0, 1]. A normalized production time series is expected, so this must be true everywhere.

Parameters

normalized_production	The normalized production value to check

```
185 {
186     if (normalized_production < 0 or normalized_production > 1) {
187         std::string error_str = "ERROR:  Production():  ";
188         error_str += "the given normalized production time series at ";
189         error_str += this->path_2_normalized_production_time_series;
190         error_str += " contains normalized production values outside the closed ";
191         error_str += "interval [0, 1]";
192 
193         #ifdef _WIN32
194             std::cout « error_str « std::endl;
195         #endif
196 
197         throw std::runtime_error(error_str);
198     }
199 
200     return;
201 }   /* __throwValueError() */
```

4.19.3.3 __checkTimePoint()

```
void Production::__checkTimePoint (
            double time_received_hrs,
            double time_expected_hrs )  [private]
```

Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.

Parameters

time_received_hrs	The point in time received from the given data.
time_expected_hrs	The point in time expected (this comes from the electrical load time series).

```
121 {
122     if (time_received_hrs != time_expected_hrs) {
123         std::string error_str = "ERROR:  Production():  ";
124         error_str += "the given normalized production time series at ";
125         error_str += this->path_2_normalized_production_time_series;
126         error_str += " does not align with the ";
127         error_str += "previously given electrical load time series";
128 
129         #ifdef _WIN32
130             std::cout « error_str « std::endl;
131         #endif
132 
133         throw std::runtime_error(error_str);
134     }
135 
136     return;
137 }   /* __checkTimePoint() */
```

4.19.3.4 __readNormalizedProductionData()

```
void Production::__readNormalizedProductionData (
            std::vector< double > * time_vec_hrs_ptr )  [private]
```

Helper method to read in a given time series of normalized production.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
222 {
223     //  1. init CSV reader
224     io::CSVReader<2> CSV(this->path_2_normalized_production_time_series);
225 
226     CSV.read_header(
227         io::ignore_extra_column,
228         "Time (since start of data) [hrs]",
229         "Normalized Production [ ]"
230     );
231 
232     //  2. read in normalized performance data,
233     //     check values and check against time series (point-wise and length)
234     int n_points = 0;
235     double time_hrs = 0;
236     double time_expected_hrs = 0;
237     double normalized_production = 0;
238 
239     while (CSV.read_row(time_hrs, normalized_production)) {
240         //  2.1. check length of data
241         if (n_points > this->n_points) {
242             this->__throwLengthError();
243         }
244 
245         //  2.2. check normalized production value
246         this->__checkNormalizedProduction(normalized_production);
247 
248         //  2.3. check time point
249         time_expected_hrs = time_vec_hrs_ptr->at(n_points);
250         this->__checkTimePoint(time_hrs, time_expected_hrs);
251 
252         //  2.4. write to normalized production vector, increment n_points
253         this->normalized_production_vec[n_points] = normalized_production;
254         n_points++;
255     }
256 
257     //  3. check length of data
258     if (n_points != this->n_points) {
259         this->__throwLengthError();
260     }
261 
262     return;
263 }   /* __readNormalizedProductionData() */
```

4.19.3.5 __throwLengthError()

```
void Production::__throwLengthError (
            void  )  [private]
```

Helper method to throw data length error (if not the same as the given electrical load time series).

```
152 {
153     std::string error_str = "ERROR:  Production():  ";
154     error_str += "the given normalized production time series at ";
155     error_str += this->path_2_normalized_production_time_series;
156     error_str += " is not the same length as the previously given electrical";
157     error_str += " load time series";
158 
159     #ifdef _WIN32
160         std::cout << error_str << std::endl;
161     #endif
162 
163     throw std::runtime_error(error_str);
164 
165     return;
166 }   /* __throwLengthError() */
```

4.19.3.6 commit()

```
double Production::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented in Wind, Wave, Tidal, Solar, Renewable, Noncombustion, Diesel, and Combustion.

```
571 {
572     //  1. record production
573     this->production_vec_kW[timestep] = production_kW;
574 
575     //  2. compute and record dispatch and curtailment
576     double dispatch_kW = 0;
577     double curtailment_kW = 0;
578 
579     if (production_kW > load_kW) {
580         dispatch_kW = load_kW;
581         curtailment_kW = production_kW - dispatch_kW;
582     }
583 
584     else {
585         dispatch_kW = production_kW;
586     }
587 
588     this->dispatch_vec_kW[timestep] = dispatch_kW;
589     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
590     this->curtailment_vec_kW[timestep] = curtailment_kW;
591 
592     //  3. update load
593     load_kW -= dispatch_kW;
594 
595     //  4. update and log running attributes
596     if (this->is_running) {
597         //  4.1. log running state, running hours
598         this->is_running_vec[timestep] = this->is_running;
599         this->running_hours += dt_hrs;
600 
601         //  4.2. incur operation and maintenance costs
602         double produced_kWh = production_kW * dt_hrs;
603 
604         double operation_maintenance_cost =
605             this->operation_maintenance_cost_kWh * produced_kWh;
606         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
607     }
608 
609     //  5. trigger replacement, if applicable
610     if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
611         this->handleReplacement(timestep);
612     }
613 
614     return load_kW;
615 }   /* commit() */
```

4.19.3.7 computeEconomics()

```
void Production::computeEconomics (
            std::vector< double > * time_vec_hrs_ptr )  [virtual]
```

Helper method to compute key economic metrics for the Model run.

Ref: HOMER [2023b]
Ref: HOMER [2023g]
Ref: HOMER [2023i]
Ref: HOMER [2023a]

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

1. compute levellized cost of energy (per unit dispatched)

Reimplemented in Renewable, Noncombustion, and Combustion.

```
469 {
470     //  1. compute net present cost
471     double t_hrs = 0;
472     double real_discount_scalar = 0;
473 
474     for (int i = 0; i < this->n_points; i++) {
475         t_hrs = time_vec_hrs_ptr->at(i);
476 
477         real_discount_scalar = 1.0 / pow(
478             1 + this->real_discount_annual,
479             t_hrs / 8760
480         );
481 
482         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
483 
484         this->net_present_cost +=
485             real_discount_scalar * this->operation_maintenance_cost_vec[i];
486     }
487 
489     //     assuming 8,760 hours per year
490     if (this->total_dispatch_kWh <= 0) {
491         this->levellized_cost_of_energy_kWh = this->net_present_cost;
492     }
493 
494     else {
495         double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
496 
497         double capital_recovery_factor =
498             (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
499             (pow(1 + this->real_discount_annual, n_years) - 1);
500 
501         double total_annualized_cost = capital_recovery_factor *
502             this->net_present_cost;
503 
504         this->levellized_cost_of_energy_kWh =
505             (n_years * total_annualized_cost) /
506             this->total_dispatch_kWh;
507     }
508 
509     return;
510 }   /* computeEconomics() */
```

4.19.3.8 computeRealDiscountAnnual()

```
double Production::computeRealDiscountAnnual (
            double nominal_inflation_annual,
            double nominal_discount_annual )
```

Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

Ref: HOMER [2023h]
Ref: HOMER [2023b]

Parameters

nominal_inflation_annual	The nominal, annual inflation rate to use in computing model economics.
nominal_discount_annual	The nominal, annual discount rate to use in computing model economics.

Returns

The real, annual discount rate to use in computing model economics.

```
442 {
443     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
444     real_discount_annual /= 1 + nominal_inflation_annual;
445 
446     return real_discount_annual;
447 }   /* __computeRealDiscountAnnual() */
```

4.19.3.9 getProductionkW()

```
double Production::getProductionkW (
            int timestep )
```

A method to simply fetch the normalized production at a particular point in the given normalized production time series, multiply by the rated capacity of the asset, and return.

Returns

The production [kW] for the asset at the given point in time, as defined by the given normalized production time series.

```
530 {
531     double production_kW =
532         this->normalized_production_vec[timestep] * this->capacity_kW;
533 
534     return production_kW;
535 }   /* getProductionkW() */
```

4.19.3.10 handleReplacement()

```
void Production::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented in Wind, Wave, Tidal, Solar, Renewable, Noncombustion, Hydro, Diesel, and Combustion.

```
399 {
400     //  1. reset attributes
401     this->is_running = false;
402 
403     //  2. log replacement
404     this->n_replacements++;
405 
406     //  3. incur capital cost in timestep
407     this->capital_cost_vec[timestep] = this->capital_cost;
408 
409     return;
410 }   /* __handleReplacement() */
```

4.19.4 Member Data Documentation

4.19.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.19.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.19.4.3 capital_cost_vec

```
std::vector<double> Production::capital_cost_vec
```

A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

4.19.4.4 curtailment_vec_kW

```
std::vector<double> Production::curtailment_vec_kW
```

A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.

4.19.4.5 dispatch_vec_kW

```
std::vector<double> Production::dispatch_vec_kW
```

A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.

4.19.4.6 interpolator

```
Interpolator Production::interpolator
```

Interpolator component of Production.

4.19.4.7 is_running

```
bool Production::is_running
```

A boolean which indicates whether or not the asset is running.

4.19.4.8 is_running_vec

```
std::vector<bool> Production::is_running_vec
```

A boolean vector for tracking if the asset is running at a particular point in time.

4.19.4.9 is_sunk

```
bool Production::is_sunk
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

4.19.4.10 levellized_cost_of_energy_kWh

```
double Production::levellized_cost_of_energy_kWh
```

The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatch.

4.19.4.11 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.19.4.12 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.19.4.13 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.19.4.14 n_years

```
double Production::n_years
```

The number of years being modelled.

4.19.4.15 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.19.4.16 nominal_discount_annual

```
double Production::nominal_discount_annual
```

The nominal, annual discount rate to use in computing model economics.

4.19.4.17 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

The nominal, annual inflation rate to use in computing model economics.

4.19.4.18 normalized_production_series_given

```
bool Production::normalized_production_series_given
```

A boolen which indicates whether or not a normalized production time series is given.

4.19.4.19 normalized_production_vec

```
std::vector<double> Production::normalized_production_vec
```

A vector of normalizd production [ ] at each point in the modelling time series.

4.19.4.20 operation_maintenance_cost_kWh

```
double Production::operation_maintenance_cost_kWh
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.

4.19.4.21 operation_maintenance_cost_vec

```
std::vector<double> Production::operation_maintenance_cost_vec
```

A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

4.19.4.22 path_2_normalized_production_time_series

```
std::string Production::path_2_normalized_production_time_series
```

A string defining the path (either relative or absolute) to the given normalized production time series.

4.19.4.23 print_flag

```
bool Production::print_flag
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.19.4.24 production_vec_kW

```
std::vector<double> Production::production_vec_kW
```

A vector of production [kW] at each point in the modelling time series.

4.19.4.25 real_discount_annual

```
double Production::real_discount_annual
```

The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.

4.19.4.26 replace_running_hrs

```
double Production::replace_running_hrs
```

The number of running hours after which the asset must be replaced.

4.19.4.27 running_hours

```
double Production::running_hours
```

The number of hours for which the assset has been operating.

4.19.4.28 storage_vec_kW

```
std::vector<double> Production::storage_vec_kW
```

A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.

4.19.4.29 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

The total energy dispatched [kWh] over the Model run.

4.19.4.30 type_str

```
std::string Production::type_str
```

A string describing the type of the asset.

The documentation for this class was generated from the following files:

- header/Production/Production.h
- source/Production/Production.cpp

4.20 ProductionInputs Struct Reference

A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input.

```
#include <Production.h>
```

Public Attributes

- bool print_flag = false

  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_sunk = false

  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double capacity_kW = 100

  The rated production capacity [kW] of the asset.
- double nominal_inflation_annual = 0.02

  The nominal, annual inflation rate to use in computing model economics.
- double nominal_discount_annual = 0.04

  The nominal, annual discount rate to use in computing model economics.
- double replace_running_hrs = 90000

  The number of running hours after which the asset must be replaced.
- std::string path_2_normalized_production_time_series = ""

  A string defining the path (either relative or absolute) to the given normalized production time series.

4.20.1 Detailed Description

A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input.

4.20.2 Member Data Documentation

4.20.2.1 capacity_kW

```
double ProductionInputs::capacity_kW = 100
```

The rated production capacity [kW] of the asset.

4.20.2.2 is_sunk

```
bool ProductionInputs::is_sunk = false
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

4.20.2.3 nominal_discount_annual

```
double ProductionInputs::nominal_discount_annual = 0.04
```

The nominal, annual discount rate to use in computing model economics.

4.20.2.4 nominal_inflation_annual

```
double ProductionInputs::nominal_inflation_annual = 0.02
```

The nominal, annual inflation rate to use in computing model economics.

4.20.2.5 path_2_normalized_production_time_series

```
std::string ProductionInputs::path_2_normalized_production_time_series = ""
```

A string defining the path (either relative or absolute) to the given normalized production time series.

4.20.2.6 print_flag

```
bool ProductionInputs::print_flag = false
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.20.2.7 replace_running_hrs

```
double ProductionInputs::replace_running_hrs = 90000
```

The number of running hours after which the asset must be replaced.

The documentation for this struct was generated from the following file:

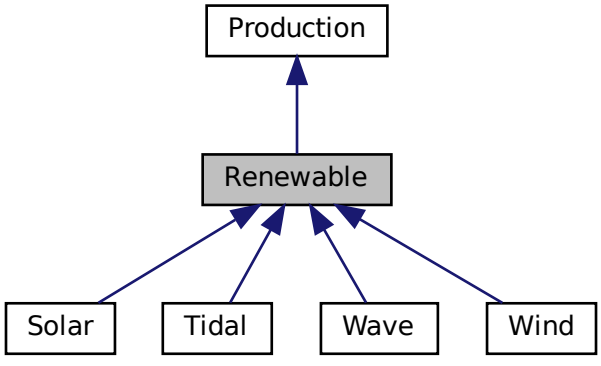
- header/Production/Production.h

4.21 Renewable Class Reference

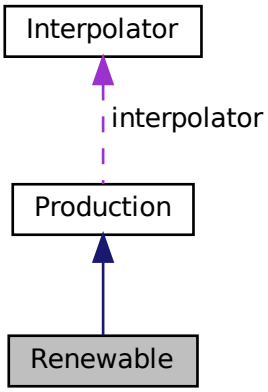
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#include <Renewable.h>
```

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



Public Member Functions

- Renewable (void)

  Constructor (dummy) for the Renewable class.
- Renewable (int, double, RenewableInputs, std::vector< double > *)

  Constructor (intended) for the Renewable class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeEconomics (std::vector< double > *)

  Helper method to compute key economic metrics for the Model run.
- virtual double computeProductionkW (int, double, double)
- virtual double computeProductionkW (int, double, double, double)
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- void writeResults (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int, int=-1)

  Method which writes Renewable results to an output directory.
- virtual ~Renewable (void)

  Destructor for the Renewable class.

Public Attributes

- RenewableType type

  The type (RenewableType) of the asset.
- int resource_key

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

Private Member Functions

- void __checkInputs (RenewableInputs)

  Helper method to check inputs to the Renewable constructor.
- void __handleStartStop (int, double, double)

  Helper method to handle the starting/stopping of the renewable asset.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

4.21.1 Detailed Description

The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Renewable() [1/2]

```
Renewable::Renewable (
            void  )
```

Constructor (dummy) for the Renewable class.

```
100 {
101     //...
102
103     return;
104 }   /* Renewable() */
```

4.21.2.2 Renewable() [2/2]

```
Renewable::Renewable (
            int n_points,
            double n_years,
            RenewableInputs renewable_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Renewable class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
renewable_inputs	A structure of Renewable constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
136   :
137 Production(
138     n_points,
139     n_years,
140     renewable_inputs.production_inputs,
141     time_vec_hrs_ptr
142 )
143 {
144     //  1. check inputs
145     this->__checkInputs(renewable_inputs);
146 
147     //  2. set attributes
148     //...
149 
150     //  3. construction print
151     if (this->print_flag) {
152         std::cout « "Renewable object constructed at " « this « std::endl;
153     }
154 
155     return;
156 }   /* Renewable() */
```

4.21.2.3 ~Renewable()

```
Renewable::~Renewable (
            void  )  [virtual]
```

Destructor for the Renewable class.

```
359 {
360     //  1. destruction print
361     if (this->print_flag) {
362         std::cout « "Renewable object at " « this « " destroyed" « std::endl;
363     }
364 
365     return;
366 }   /* ~Renewable() */
```

4.21.3 Member Function Documentation

4.21.3.1 __checkInputs()

```
void Renewable::__checkInputs (
            RenewableInputs renewable_inputs )  [private]
```

Helper method to check inputs to the Renewable constructor.

```
37 {
38     //...
39 
40     return;
41 }   /* __checkInputs() */
```

4.21.3.2 __handleStartStop()

```
void Renewable::__handleStartStop (
            int timestep,
            double dt_hrs,
            double production_kW )  [private]
```

Helper method to handle the starting/stopping of the renewable asset.

```
64 {
65     if (this->is_running) {
66         // handle stopping
67         if (production_kW <= 0) {
68             this->is_running = false;
69         }
70     }
71 
72     else {
73         // handle starting
74         if (production_kW > 0) {
75             this->is_running = true;
76             this->n_starts++;
77         }
78     }
79 
80     return;
81 }   /* __handleStartStop() */
```

4.21.3.3 __writeSummary()

```
virtual void Renewable::__writeSummary (
            std::string  )  [inline], [private], [virtual]
```

Reimplemented in Wind, Wave, Tidal, and Solar.

```
72 {return;}
```

4.21.3.4 __writeTimeSeries()

```
virtual void Renewable::__writeTimeSeries (
            std::string ,
            std::vector< double > * ,
            std::map< int, std::vector< double >> * ,
            std::map< int, std::vector< std::vector< double >>> * ,
            int  = -1 )  [inline], [private], [virtual]
```

Reimplemented in Wind, Wave, Tidal, and Solar.

```
79          {return;}
```

4.21.3.5 commit()

```
double Renewable::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
240 {
241     //  1. handle start/stop
242     this->__handleStartStop(timestep, dt_hrs, production_kW);
243 
244     //  2. invoke base class method
245     load_kW = Production :: commit(
246         timestep,
247         dt_hrs,
248         production_kW,
249         load_kW
250     );
251 
252 
253     //...
254 
255     return load_kW;
256 }   /* commit() */
```

4.21.3.6 computeEconomics()

```
void Renewable::computeEconomics (
             std::vector< double > * time_vec_hrs_ptr )  [virtual]
```

Helper method to compute key economic metrics for the Model run.

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
199 {
200     //  1. invoke base class method
201     Production :: computeEconomics(time_vec_hrs_ptr);
202 
203     return;
204 }   /* computeEconomics() */
```

4.21.3.7 computeProductionkW() [1/2]

```
virtual double Renewable::computeProductionkW (
             int ,
```

```
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Wind, Tidal, and Solar.

```
96 {return 0;}
```

4.21.3.8 computeProductionkW() [2/2]

```
virtual double Renewable::computeProductionkW (
            int ,
            double ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Wave.

```
97 {return 0;}
```

4.21.3.9 handleReplacement()

```
void Renewable::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
174 {
175     //  1. reset attributes
176     //...
177
178     //  2. invoke base class method
179     Production :: handleReplacement(timestep);
180
181     return;
182 }   /* __handleReplacement() */
```

4.21.3.10 writeResults()

```
void Renewable::writeResults (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int renewable_index,
            int max_lines = -1 )
```

Method which writes Renewable results to an output directory.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
renewable_index	An integer which corresponds to the index of the Renewable asset in the Model.
max_lines	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```
300 {
301     //  1. handle sentinel
302     if (max_lines < 0) {
303         max_lines = this->n_points;
304     }
305 
306     //  2. create subdirectories
307     write_path += "Production/";
308     if (not std::filesystem::is_directory(write_path)) {
309         std::filesystem::create_directory(write_path);
310     }
311 
312     write_path += "Renewable/";
313     if (not std::filesystem::is_directory(write_path)) {
314         std::filesystem::create_directory(write_path);
315     }
316 
317     write_path += this->type_str;
318     write_path += "_";
319     write_path += std::to_string(int(ceil(this->capacity_kW)));
320     write_path += "kW_idx";
321     write_path += std::to_string(renewable_index);
322     write_path += "/";
323     std::filesystem::create_directory(write_path);
324 
325     //  3. write summary
326     this->__writeSummary(write_path);
327 
328     //  4. write time series
329     if (max_lines > this->n_points) {
330         max_lines = this->n_points;
331     }
332 
333     if (max_lines > 0) {
334         this->__writeTimeSeries(
335             write_path,
336             time_vec_hrs_ptr,
337             resource_map_1D_ptr,
338             resource_map_2D_ptr,
339             max_lines
340         );
341     }
342 
343     return;
344 }   /* writeResults() */
```

4.21.4 Member Data Documentation

4.21.4.1 resource_key

```
int Renewable::resource_key
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

4.21.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

The documentation for this class was generated from the following files:

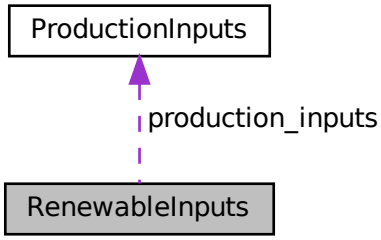
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

4.22 RenewableInputs Struct Reference

A structure which bundles the necessary inputs for the Renewable constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

```
#include <Renewable.h>
```

Collaboration diagram for RenewableInputs:



Public Attributes

- ProductionInputs production_inputs

  An encapsulated ProductionInputs instance.

4.22.1 Detailed Description

A structure which bundles the necessary inputs for the Renewable constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.

4.22.2 Member Data Documentation

4.22.2.1 production_inputs

```
ProductionInputs RenewableInputs::production_inputs
```

An encapsulated ProductionInputs instance.

The documentation for this struct was generated from the following file:

- header/Production/Renewable/Renewable.h

4.23 Resources Class Reference

A class which contains renewable resource data. Intended to serve as a component class of Model.

```
#include <Resources.h>
```

Public Member Functions

- Resources (void)

  Constructor for the Resources class.
- void addResource (NoncombustionType, std::string, int, ElectricalLoad *)

  A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).
- void addResource (RenewableType, std::string, int, ElectricalLoad *)

  A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).
- void clear (void)

  Method to clear all attributes of the Resources object.
- ~Resources (void)

  Destructor for the Resources class.

Public Attributes

- std::map< int, std::vector< double > > resource_map_1D

  A map <int, vector<double>> of given 1D renewable resource time series.
- std::map< int, std::string > string_map_1D

  A map <int, string> of descriptors for the type of the given 1D renewable resource time series.
- std::map< int, std::string > path_map_1D

  A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.
- std::map< int, std::vector< std::vector< double > > > resource_map_2D

  A map <int, vector<vector<double>>> of given 2D renewable resource time series.
- std::map< int, std::string > string_map_2D

  A map <int, string> of descriptors for the type of the given 2D renewable resource time series.
- std::map< int, std::string > path_map_2D

  A map <int, string> of the paths (either relative or absolute) to given 2D renewable resource time series.

Private Member Functions

- void __checkResourceKey1D (int, RenewableType)

  Helper method to check if given resource key (1D) is already in use.
- void __checkResourceKey2D (int, RenewableType)

  Helper method to check if given resource key (2D) is already in use.
- void __checkResourceKey1D (int, NoncombustionType)

  Helper method to check if given resource key (1D) is already in use.
- void __checkTimePoint (double, double, std::string, ElectricalLoad *)

  Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.
- void __throwLengthError (std::string, ElectricalLoad *)

  Helper method to throw data length error (if not the same as the given electrical load time series).
- void __readHydroResource (std::string, int, ElectricalLoad *)

  Helper method to handle reading a hydro resource time series into Resources.
- void __readSolarResource (std::string, int, ElectricalLoad *)

  Helper method to handle reading a solar resource time series into Resources.
- void __readTidalResource (std::string, int, ElectricalLoad *)

  Helper method to handle reading a tidal resource time series into Resources.
- void __readWaveResource (std::string, int, ElectricalLoad *)

  Helper method to handle reading a wave resource time series into Resources.
- void __readWindResource (std::string, int, ElectricalLoad *)

  Helper method to handle reading a wind resource time series into Resources.

4.23.1 Detailed Description

A class which contains renewable resource data. Intended to serve as a component class of Model.

4.23.2 Constructor & Destructor Documentation

4.23.2.1 Resources()

```
Resources::Resources (
            void  )
```

Constructor for the Resources class.

```
730 {
731     return;
732 }   /* Resources() */
```

4.23.2.2 ~Resources()

```
Resources::~Resources (
            void  )
```

Destructor for the Resources class.

```
942 {
943     this->clear();
944     return;
945 }   /* ~Resources() */
```

4.23.3 Member Function Documentation

4.23.3.1 __checkResourceKey1D() [1/2]

```
void Resources::__checkResourceKey1D (
            int resource_key,
            NoncombustionType noncombustion_type )  [private]
```

Helper method to check if given resource key (1D) is already in use.

Parameters

resource_key	The key associated with the given renewable resource.
noncombustion_type	The type of renewable resource being added to Resources.

```
114 {
115     if (this->resource_map_1D.count(resource_key) > 0) {
116         std::string error_str = "ERROR:  Resources::addResource(";
117 
118         switch (noncombustion_type) {
119             case (NoncombustionType :: HYDRO): {
120                 error_str += "HYDRO):  ";
121 
122                 break;
123             }
124 
125             default: {
126                 error_str += "UNDEFINED_TYPE):  ";
127 
128                 break;
129             }
130         }
131 
132         error_str += "resource key (1D) ";
133         error_str += std::to_string(resource_key);
134         error_str += " is already in use";
135 
136         #ifdef _WIN32
137             std::cout « error_str « std::endl;
138         #endif
139 
140         throw std::invalid_argument(error_str);
141     }
142 
143     return;
144 }   /*  __checkResourceKey1D() */
```

4.23.3.2 __checkResourceKey1D() [2/2]

```
void Resources::__checkResourceKey1D (
            int resource_key,
            RenewableType renewable_type )  [private]
```

Helper method to check if given resource key (1D) is already in use.

Parameters

resource_key	The key associated with the given renewable resource.
renewable_type	The type of renewable resource being added to Resources.

```
47 {
48     if (this->resource_map_1D.count(resource_key) > 0) {
49         std::string error_str = "ERROR:  Resources::addResource(";
50 
51         switch (renewable_type) {
52             case (RenewableType :: SOLAR): {
53                 error_str += "SOLAR):  ";
54 
55                 break;
56             }
57 
58             case (RenewableType :: TIDAL): {
59                 error_str += "TIDAL):  ";
60 
61                 break;
62             }
63 
64             case (RenewableType :: WIND): {
65                 error_str += "WIND):  ";
66 
67                 break;
68             }
69 
70             default: {
71                 error_str += "UNDEFINED_TYPE):  ";
72 
73                 break;
74             }
75         }
76 
77         error_str += "resource key (1D) ";
78         error_str += std::to_string(resource_key);
79         error_str += " is already in use";
80 
81         #ifdef _WIN32
82             std::cout « error_str « std::endl;
83         #endif
84 
85         throw std::invalid_argument(error_str);
86     }
87 
88     return;
89 }   /*  __checkResourceKey1D() */
```

4.23.3.3 __checkResourceKey2D()

```
void Resources::__checkResourceKey2D (
            int resource_key,
            RenewableType renewable_type )  [private]
```

Helper method to check if given resource key (2D) is already in use.

Parameters

resource_key	The key associated with the given renewable resource.

```
167 {
168     if (this->resource_map_2D.count(resource_key) > 0) {
169         std::string error_str = "ERROR:  Resources::addResource(";
170 
171         switch (renewable_type) {
172             case (RenewableType :: WAVE): {
173                 error_str += "WAVE):  ";
174 
175                 break;
176             }
177 
178             default: {
179                 error_str += "UNDEFINED_TYPE):  ";
180 
181                 break;
182             }
183         }
184 
```

```
185         error_str += "resource key (2D) ";
186         error_str += std::to_string(resource_key);
187         error_str += " is already in use";
188 
189         #ifdef _WIN32
190             std::cout << error_str << std::endl;
191         #endif
192 
193         throw std::invalid_argument(error_str);
194     }
195 
196     return;
197 }   /*  __checkResourceKey2D() */
```

4.23.3.4 __checkTimePoint()

```
void Resources::__checkTimePoint (
            double time_received_hrs,
            double time_expected_hrs,
            std::string path_2_resource_data,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.

Parameters

time_received_hrs	The point in time received from the given data.
time_expected_hrs	The point in time expected (this comes from the electrical load time series).
path_2_resource_data	The path (either relative or absolute) to the given resource time series.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
234 {
235     if (time_received_hrs != time_expected_hrs) {
236         std::string error_str = "ERROR:  Resources::addResource():  ";
237         error_str += "the given resource time series at ";
238         error_str += path_2_resource_data;
239         error_str += " does not align with the ";
240         error_str += "previously given electrical load time series at ";
241         error_str += electrical_load_ptr->path_2_electrical_load_time_series;
242 
243         #ifdef _WIN32
244             std::cout << error_str << std::endl;
245         #endif
246 
247         throw std::runtime_error(error_str);
248     }
249 
250     return;
251 }   /* __checkTimePoint() */
```

4.23.3.5 __readHydroResource()

```
void Resources::__readHydroResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a hydro resource time series into Resources.

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
resource_key	The key associated with the given renewable resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
323 {
324     //  1. init CSV reader, record path and type
325     io::CSVReader<2> CSV(path_2_resource_data);
326 
327     CSV.read_header(
328         io::ignore_extra_column,
329         "Time (since start of data) [hrs]",
330         "Hydro Inflow [m3/hr]"
331     );
332 
333     this->path_map_1D.insert(
334         std::pair<int, std::string>(resource_key, path_2_resource_data)
335     );
336 
337     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "HYDRO"));
338 
339     //  2. init map element
340     this->resource_map_1D.insert(
341         std::pair<int, std::vector<double»(resource_key, {})
342     );
343     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
344 
345 
346     //  3. read in resource data, check against time series (point-wise and length)
347     int n_points = 0;
348     double time_hrs = 0;
349     double time_expected_hrs = 0;
350     double hydro_resource_m3hr = 0;
351 
352     while (CSV.read_row(time_hrs, hydro_resource_m3hr)) {
353         if (n_points > electrical_load_ptr->n_points) {
354             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
355         }
356 
357         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
358         this->__checkTimePoint(
359             time_hrs,
360             time_expected_hrs,
361             path_2_resource_data,
362             electrical_load_ptr
363         );
364 
365         this->resource_map_1D[resource_key][n_points] = hydro_resource_m3hr;
366 
367         n_points++;
368     }
369 
370     //  4. check data length
371     if (n_points != electrical_load_ptr->n_points) {
372         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
373     }
374 
375     return;
376 }   /* __readHydroResource() */
```

4.23.3.6 __readSolarResource()

```
void Resources::__readSolarResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a solar resource time series into Resources.

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
resource_key	The key associated with the given renewable resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
406 {
407     //  1. init CSV reader, record path and type
408     io::CSVReader<2> CSV(path_2_resource_data);
409 
410     CSV.read_header(
411         io::ignore_extra_column,
412         "Time (since start of data) [hrs]",
413         "Solar GHI [kW/m2]"
414     );
415 
416     this->path_map_1D.insert(
417         std::pair<int, std::string>(resource_key, path_2_resource_data)
418     );
419 
420     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));
421 
422     //  2. init map element
423     this->resource_map_1D.insert(
424         std::pair<int, std::vector<double»(resource_key, {})
425     );
426     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
427 
428 
429     //  3. read in resource data, check against time series (point-wise and length)
430     int n_points = 0;
431     double time_hrs = 0;
432     double time_expected_hrs = 0;
433     double solar_resource_kWm2 = 0;
434 
435     while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
436         if (n_points > electrical_load_ptr->n_points) {
437             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
438         }
439 
440         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
441         this->__checkTimePoint(
442             time_hrs,
443             time_expected_hrs,
444             path_2_resource_data,
445             electrical_load_ptr
446         );
447 
448         this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
449 
450         n_points++;
451     }
452 
453     //  4. check data length
454     if (n_points != electrical_load_ptr->n_points) {
455         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
456     }
457 
458     return;
459 }   /* __readSolarResource() */
```

4.23.3.7 __readTidalResource()

```
void Resources::__readTidalResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a tidal resource time series into Resources.

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
resource_key	The key associated with the given renewable resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
489 {
490     //  1. init CSV reader, record path and type
491     io::CSVReader<2> CSV(path_2_resource_data);
492 
493     CSV.read_header(
494         io::ignore_extra_column,
495         "Time (since start of data) [hrs]",
496         "Tidal Speed (hub depth) [m/s]"
497     );
498 
499     this->path_map_1D.insert(
500         std::pair<int, std::string>(resource_key, path_2_resource_data)
501     );
502 
503     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));
504 
505     //  2. init map element
506     this->resource_map_1D.insert(
507         std::pair<int, std::vector<double»(resource_key, {})
508     );
509     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
510 
511 
512     //  3. read in resource data, check against time series (point-wise and length)
513     int n_points = 0;
514     double time_hrs = 0;
515     double time_expected_hrs = 0;
516     double tidal_resource_ms = 0;
517 
518     while (CSV.read_row(time_hrs, tidal_resource_ms)) {
519         if (n_points > electrical_load_ptr->n_points) {
520             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
521         }
522 
523         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
524         this->__checkTimePoint(
525             time_hrs,
526             time_expected_hrs,
527             path_2_resource_data,
528             electrical_load_ptr
529         );
530 
531         this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;
532 
533         n_points++;
534     }
535 
536     //  4. check data length
537     if (n_points != electrical_load_ptr->n_points) {
538         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
539     }
540 
541     return;
542 }   /* __readTidalResource() */
```

4.23.3.8 __readWaveResource()

```
void Resources::__readWaveResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a wave resource time series into Resources.

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
resource_key	The key associated with the given renewable resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
572 {
573     //  1. init CSV reader, record path and type
574     io::CSVReader<3> CSV(path_2_resource_data);
575 
576     CSV.read_header(
577         io::ignore_extra_column,
578         "Time (since start of data) [hrs]",
579         "Significant Wave Height [m]",
580         "Energy Period [s]"
581     );
582 
583     this->path_map_2D.insert(
584         std::pair<int, std::string>(resource_key, path_2_resource_data)
585     );
586 
587     this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));
588 
589     //  2. init map element
590     this->resource_map_2D.insert(
591         std::pair<int, std::vector<std::vector<double»>(resource_key, {})
592     );
593     this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});
594 
595 
596     //  3. read in resource data, check against time series (point-wise and length)
597     int n_points = 0;
598     double time_hrs = 0;
599     double time_expected_hrs = 0;
600     double significant_wave_height_m = 0;
601     double energy_period_s = 0;
602 
603     while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
604         if (n_points > electrical_load_ptr->n_points) {
605             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
606         }
607 
608         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
609         this->__checkTimePoint(
610             time_hrs,
611             time_expected_hrs,
612             path_2_resource_data,
613             electrical_load_ptr
614         );
615 
616         this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
617         this->resource_map_2D[resource_key][n_points][1] = energy_period_s;
618 
619         n_points++;
620     }
621 
622     //  4. check data length
623     if (n_points != electrical_load_ptr->n_points) {
624         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
625     }
626 
627     return;
628 }   /* __readWaveResource() */
```

4.23.3.9 __readWindResource()

```
void Resources::__readWindResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a wind resource time series into Resources.

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
resource_key	The key associated with the given renewable resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
658 {
659     //  1. init CSV reader, record path and type
660     io::CSVReader<2> CSV(path_2_resource_data);
661 
662     CSV.read_header(
663         io::ignore_extra_column,
664         "Time (since start of data) [hrs]",
665         "Wind Speed (hub height) [m/s]"
666     );
667 
668     this->path_map_1D.insert(
669         std::pair<int, std::string>(resource_key, path_2_resource_data)
670     );
671 
672     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));
673 
674     //  2. init map element
675     this->resource_map_1D.insert(
676         std::pair<int, std::vector<double»(resource_key, {})
677     );
678     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
679 
680 
681     //  3. read in resource data, check against time series (point-wise and length)
682     int n_points = 0;
683     double time_hrs = 0;
684     double time_expected_hrs = 0;
685     double wind_resource_ms = 0;
686 
687     while (CSV.read_row(time_hrs, wind_resource_ms)) {
688         if (n_points > electrical_load_ptr->n_points) {
689             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
690         }
691 
692         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
693         this->__checkTimePoint(
694             time_hrs,
695             time_expected_hrs,
696             path_2_resource_data,
697             electrical_load_ptr
698         );
699 
700         this->resource_map_1D[resource_key][n_points] = wind_resource_ms;
701 
702         n_points++;
703     }
704 
705     //  4. check data length
706     if (n_points != electrical_load_ptr->n_points) {
707         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
708     }
709 
710     return;
711 }   /* __readWindResource() */
```

4.23.3.10 __throwLengthError()

```
void Resources::__throwLengthError (
            std::string path_2_resource_data,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to throw data length error (if not the same as the given electrical load time series).

Parameters

path_2_resource_data	The path (either relative or absolute) to the given resource time series.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
278 {
279     std::string error_str = "ERROR:  Resources::addResource():  ";
280     error_str += "the given resource time series at ";
281     error_str += path_2_resource_data;
282     error_str += " is not the same length as the previously given electrical";
283     error_str += " load time series at ";
284     error_str += electrical_load_ptr->path_2_electrical_load_time_series;
285 
286     #ifdef _WIN32
287         std::cout « error_str « std::endl;
288     #endif
289 
290     throw std::runtime_error(error_str);
291 
292     return;
293 }   /* __throwLengthError() */
```

4.23.3.11 addResource() [1/2]

```
void Resources::addResource (
             NoncombustionType noncombustion_type,
             std::string path_2_resource_data,
             int resource_key,
             ElectricalLoad * electrical_load_ptr )
```

A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

Parameters

noncombustion_type	The type of renewable resource being added to Resources.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
769 {
770     switch (noncombustion_type) {
771         case (NoncombustionType :: HYDRO): {
772             this->__checkResourceKey1D(resource_key, noncombustion_type);
773 
774             this->__readHydroResource(
775                 path_2_resource_data,
776                 resource_key,
777                 electrical_load_ptr
778             );
779 
780             break;
781         }
782 
783         default: {
784             std::string error_str = "ERROR:  Resources :: addResource(:  ";
785             error_str += "noncombustion type ";
786             error_str += std::to_string(noncombustion_type);
787             error_str += " has no associated resource";
788 
789             #ifdef _WIN32
790                 std::cout « error_str « std::endl;
791             #endif
792 
793             throw std::runtime_error(error_str);
794 
795             break;
796         }
797     }
798 
799     return;
```

```
800 }   /* addResource() */
```

4.23.3.12 addResource() [2/2]

```
void Resources::addResource (
            RenewableType renewable_type,
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )
```

A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

Parameters

renewable_type	The type of renewable resource being added to Resources.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

```
837 {
838     switch (renewable_type) {
839         case (RenewableType :: SOLAR): {
840             this->__checkResourceKey1D(resource_key, renewable_type);
841 
842             this->__readSolarResource(
843                 path_2_resource_data,
844                 resource_key,
845                 electrical_load_ptr
846             );
847 
848             break;
849         }
850 
851         case (RenewableType :: TIDAL): {
852             this->__checkResourceKey1D(resource_key, renewable_type);
853 
854             this->__readTidalResource(
855                 path_2_resource_data,
856                 resource_key,
857                 electrical_load_ptr
858             );
859 
860             break;
861         }
862 
863         case (RenewableType :: WAVE): {
864             this->__checkResourceKey2D(resource_key, renewable_type);
865 
866             this->__readWaveResource(
867                 path_2_resource_data,
868                 resource_key,
869                 electrical_load_ptr
870             );
871 
872             break;
873         }
874 
875         case (RenewableType :: WIND): {
876             this->__checkResourceKey1D(resource_key, renewable_type);
877 
878             this->__readWindResource(
879                 path_2_resource_data,
880                 resource_key,
881                 electrical_load_ptr
882             );
```

```
883
884             break;
885         }
886
887         default: {
888             std::string error_str = "ERROR:  Resources :: addResource(:  ";
889             error_str += "renewable type ";
890             error_str += std::to_string(renewable_type);
891             error_str += " not recognized";
892
893             #ifdef _WIN32
894                 std::cout « error_str « std::endl;
895             #endif
896
897             throw std::runtime_error(error_str);
898
899             break;
900         }
901     }
902
903     return;
904 }   /* addResource() */
```

4.23.3.13 clear()

```
void Resources::clear (
              void  )
```

Method to clear all attributes of the Resources object.

```
918 {
919     this->resource_map_1D.clear();
920     this->string_map_1D.clear();
921     this->path_map_1D.clear();
922
923     this->resource_map_2D.clear();
924     this->string_map_2D.clear();
925     this->path_map_2D.clear();
926
927     return;
928 }   /* clear() */
```

4.23.4 Member Data Documentation

4.23.4.1 path_map_1D

```
std::map<int, std::string> Resources::path_map_1D
```

A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.

4.23.4.2 path_map_2D

```
std::map<int, std::string> Resources::path_map_2D
```

A map <int, string> of the paths (either relative or absolute) to given 2D renewable resource time series.

4.23.4.3 resource_map_1D

```
std::map<int, std::vector<double> > Resources::resource_map_1D
```

A map <int, vector<double>> of given 1D renewable resource time series.

4.23.4.4 resource_map_2D

```
std::map<int, std::vector<std::vector<double> > > Resources::resource_map_2D
```

A map <int, vector<vector<double>>> of given 2D renewable resource time series.

4.23.4.5 string_map_1D

```
std::map<int, std::string> Resources::string_map_1D
```

A map <int, string> of descriptors for the type of the given 1D renewable resource time series.

4.23.4.6 string_map_2D

```
std::map<int, std::string> Resources::string_map_2D
```

A map <int, string> of descriptors for the type of the given 2D renewable resource time series.

The documentation for this class was generated from the following files:

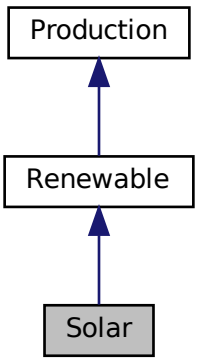
- header/Resources.h
- source/Resources.cpp

4.24 Solar Class Reference

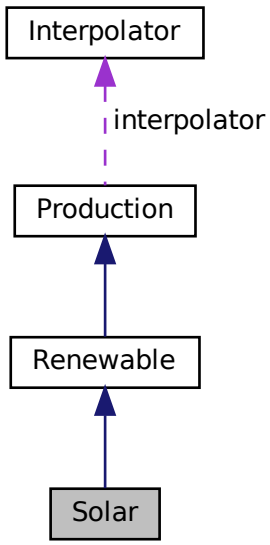
A derived class of the Renewable branch of Production which models solar production.

`#include <Solar.h>`

Inheritance diagram for Solar:



Collaboration diagram for Solar:



Public Member Functions

- Solar (void)

  Constructor (dummy) for the Solar class.
- Solar (int, double, SolarInputs, std::vector< double > *)

  Constructor (intended) for the Solar class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

  Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Solar (void)

  Destructor for the Solar class.

Public Attributes

- double derating

  The derating of the solar PV array (i.e., shadowing, soiling, etc.).

Private Member Functions

- void __checkInputs (SolarInputs)

  Helper method to check inputs to the Solar constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic solar PV array capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.
- void __writeSummary (std::string)

  Helper method to write summary results for Solar.
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

  Helper method to write time series results for Solar.

4.24.1 Detailed Description

A derived class of the Renewable branch of Production which models solar production.

4.24.2 Constructor & Destructor Documentation

4.24.2.1 Solar() [1/2]

```
Solar::Solar (
            void  )
```

Constructor (dummy) for the Solar class.

```
297 {
298     //...
299 
300     return;
301 }   /* Solar() */
```

4.24.2.2 Solar() [2/2]

```
Solar::Solar (
            int n_points,
            double n_years,
            SolarInputs solar_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Solar class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
solar_inputs	A structure of Solar constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
333   :
334 Renewable(
335     n_points,
336     n_years,
337     solar_inputs.renewable_inputs,
338     time_vec_hrs_ptr
339 )
340 {
341     //  1. check inputs
342     this->__checkInputs(solar_inputs);
343 
344     //  2. set attributes
345     this->type = RenewableType :: SOLAR;
346     this->type_str = "SOLAR";
347 
348     this->resource_key = solar_inputs.resource_key;
349 
350     this->derating = solar_inputs.derating;
351 
352     if (solar_inputs.capital_cost < 0) {
353         this->capital_cost = this->__getGenericCapitalCost();
354     }
355     else {
356         this->capital_cost = solar_inputs.capital_cost;
357     }
358 
359     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
360         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
361     }
362     else {
363         this->operation_maintenance_cost_kWh =
364             solar_inputs.operation_maintenance_cost_kWh;
365     }
366 
367     if (not this->is_sunk) {
368         this->capital_cost_vec[0] = this->capital_cost;
369     }
370 
```

```
371     //  3. construction print
372     if (this->print_flag) {
373         std::cout « "Solar object constructed at " « this « std::endl;
374     }
375
376     return;
377 }   /* Renewable() */
```

4.24.2.3 ~Solar()

```
Solar::~Solar (
            void  )
```

Destructor for the Solar class.

```
523 {
524     //  1. destruction print
525     if (this->print_flag) {
526         std::cout « "Solar object at " « this « " destroyed" « std::endl;
527     }
528
529     return;
530 }   /* ~Solar() */
```

4.24.3 Member Function Documentation

4.24.3.1 __checkInputs()

```
void Solar::__checkInputs (
            SolarInputs solar_inputs )  [private]
```

Helper method to check inputs to the Solar constructor.

```
37 {
38     //  1. check derating
39     if (
40         solar_inputs.derating < 0 or
41         solar_inputs.derating > 1
42     ) {
43         std::string error_str = "ERROR:  Solar():  ";
44         error_str += "SolarInputs::derating must be in the closed interval [0, 1]";
45
46         #ifdef _WIN32
47             std::cout « error_str « std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     return;
54 }   /* __checkInputs() */
```

4.24.3.2 __getGenericCapitalCost()

```
double Solar::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic solar PV array capital cost.

This model was obtained by way of surveying an assortment of published solar PV costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD].

Returns

A generic capital cost for the solar PV array [CAD].

```
76 {
77     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;
78
79     return capital_cost_per_kW * this->capacity_kW;
80 }   /* __getGenericCapitalCost() */
```

4.24.3.3 __getGenericOpMaintCost()

```
double Solar::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.

This model was obtained by way of surveying an assortment of published solar PV costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Returns

A generic operation and maintenance cost, per unit energy produced, for the solar PV array [CAD/kWh].

```
103 {
104     return 0.01;
105 }   /* __getGenericOpMaintCost() */
```

4.24.3.4 __writeSummary()

```
void Solar::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Solar.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Renewable.

```
123 {
124     //  1. create filestream
125     write_path += "summary_results.md";
126     std::ofstream ofs;
127     ofs.open(write_path, std::ofstream::out);
128 
129     //  2. write summary results (markdown)
130     ofs « "# ";
131     ofs « std::to_string(int(ceil(this->capacity_kW)));
132     ofs « " kW SOLAR Summary Results\n";
133     ofs « "\n--------\n\n";
134 
135     //  2.1. Production attributes
136     ofs « "## Production Attributes\n";
137     ofs « "\n";
138 
139     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
140     ofs « "\n";
141 
142     ofs « "Production Override: (N = 0 / Y = 1): "
143         « this->normalized_production_series_given « "  \n";
144     if (this->normalized_production_series_given) {
145         ofs « "Path to Normalized Production Time Series: "
146             « this->path_2_normalized_production_time_series « "  \n";
147     }
148     ofs « "\n";
149 
150     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
151     ofs « "Capital Cost: " « this->capital_cost « "  \n";
152     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
153         « " per kWh produced  \n";
154     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
155         « "  \n";
156     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
157         « "  \n";
158     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
159     ofs « "\n";
160 
161     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
162     ofs « "\n--------\n\n";
163 
164     //  2.2. Renewable attributes
165     ofs « "## Renewable Attributes\n";
166     ofs « "\n";
167 
168     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
169 
170     ofs « "\n--------\n\n";
171 
172     //  2.3. Solar attributes
173     ofs « "## Solar Attributes\n";
174     ofs « "\n";
175 
176     ofs « "Derating Factor: " « this->derating « "  \n";
177 
178     ofs « "\n--------\n\n";
179 
180     //  2.4. Solar Results
181     ofs « "## Results\n";
182     ofs « "\n";
183 
184     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
185     ofs « "\n";
186 
187     ofs « "Total Dispatch: " « this->total_dispatch_kWh
188         « " kWh  \n";
189 
190     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
191         « " per kWh dispatched  \n";
192     ofs « "\n";
193 
194     ofs « "Running Hours: " « this->running_hours « "  \n";
195     ofs « "Replacements: " « this->n_replacements « "  \n";
196 
197     ofs « "\n--------\n\n";
198 
199     ofs.close();
200     return;
201 }   /* __writeSummary() */
```

4.24.3.5 __writeTimeSeries()

```
void Solar::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Solar.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
max_lines	The maximum number of lines of output to write.

Reimplemented from Renewable.

```
239 {
240     //  1. create filestream
241     write_path += "time_series_results.csv";
242     std::ofstream ofs;
243     ofs.open(write_path, std::ofstream::out);
244 
245     //  2. write time series results (comma separated value)
246     ofs « "Time (since start of data) [hrs],";
247     ofs « "Solar Resource [kW/m2],";
248     ofs « "Production [kW],";
249     ofs « "Dispatch [kW],";
250     ofs « "Storage [kW],";
251     ofs « "Curtailment [kW],";
252     ofs « "Capital Cost (actual),";
253     ofs « "Operation and Maintenance Cost (actual),";
254     ofs « "\n";
255 
256     for (int i = 0; i < max_lines; i++) {
257         ofs « time_vec_hrs_ptr->at(i) « ",";
258 
259         if (not this->normalized_production_series_given) {
260             ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
261         }
262 
263         else {
264             ofs « "OVERRIDE" « ",";
265         }
266 
267         ofs « this->production_vec_kW[i] « ",";
268         ofs « this->dispatch_vec_kW[i] « ",";
269         ofs « this->storage_vec_kW[i] « ",";
270         ofs « this->curtailment_vec_kW[i] « ",";
271         ofs « this->capital_cost_vec[i] « ",";
272         ofs « this->operation_maintenance_cost_vec[i] « ",";
273         ofs « "\n";
274     }
275 
276     ofs.close();
277     return;
278 }   /* __writeTimeSeries() */
```

4.24.3.6 commit()

```
double Solar::commit (
            int timestep,
```

```
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Renewable.

```
495 {
496     //  1. invoke base class method
497     load_kW = Renewable :: commit(
498         timestep,
499         dt_hrs,
500         production_kW,
501         load_kW
502     );
503 
504 
505     //...
506 
507     return load_kW;
508 }   /* commit() */
```

4.24.3.7 computeProductionkW()

```
double Solar::computeProductionkW (
            int timestep,
            double dt_hrs,
            double solar_resource_kWm2 )  [virtual]
```

Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.

Ref: HOMER [2023f]

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
solar_resource_kWm2	Solar resource (i.e. irradiance) [kW/m2].

Returns

The production [kW] of the solar PV array.

Reimplemented from Renewable.

```
437 {
438     //  given production time series override
439     if (this->normalized_production_series_given) {
440         double production_kW = Production :: getProductionkW(timestep);
441 
442         return production_kW;
443     }
444 
445     // check if no resource
446     if (solar_resource_kWm2 <= 0) {
447         return 0;
448     }
449 
450     // compute production
451     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
452 
453     // cap production at capacity
454     if (production_kW > this->capacity_kW) {
455         production_kW = this->capacity_kW;
456     }
457 
458     return production_kW;
459 }   /* computeProductionkW() */
```

4.24.3.8 handleReplacement()

```
void Solar::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
395 {
396     //  1. reset attributes
397     //...
398 
399     //  2. invoke base class method
400     Renewable :: handleReplacement(timestep);
401 
402     return;
403 }   /* __handleReplacement() */
```

4.24.4 Member Data Documentation

4.24.4.1 derating

```
double Solar::derating
```

The derating of the solar PV array (i.e., shadowing, soiling, etc.).

The documentation for this class was generated from the following files:

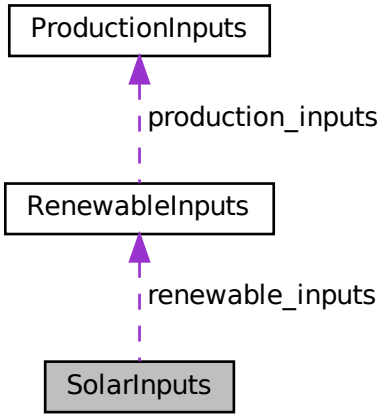
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

4.25 SolarInputs Struct Reference

A structure which bundles the necessary inputs for the Solar constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

`#include <Solar.h>`

Collaboration diagram for SolarInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double derating = 0.8

  The derating of the solar PV array (i.e., shadowing, soiling, etc.).

4.25.1 Detailed Description

A structure which bundles the necessary inputs for the Solar constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

4.25.2 Member Data Documentation

4.25.2.1 capital_cost

```
double SolarInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.25.2.2 derating

```
double SolarInputs::derating = 0.8
```

The derating of the solar PV array (i.e., shadowing, soiling, etc.).

4.25.2.3 operation_maintenance_cost_kWh

```
double SolarInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.25.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.25.2.5 resource_key

```
int SolarInputs::resource_key = 0
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

The documentation for this struct was generated from the following file:

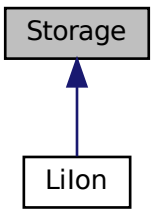
- header/Production/Renewable/Solar.h

4.26 Storage Class Reference

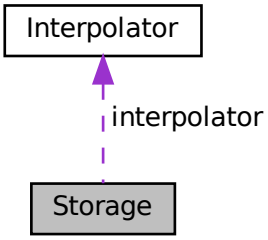
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

`#include <Storage.h>`

Inheritance diagram for Storage:



Collaboration diagram for Storage:



Public Member Functions

- Storage (void)

  Constructor (dummy) for the Storage class.
- Storage (int, double, StorageInputs)

  Constructor (intended) for the Storage class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeEconomics (std::vector< double > *)

  Helper method to compute key economic metrics for the Model run.
- virtual double getAvailablekW (double)
- virtual double getAcceptablekW (double)
- virtual void commitCharge (int, double, double)

- virtual double commitDischarge (int, double, double, double)
- void writeResults (std::string, std::vector< double > *, int, int=-1)

  Method which writes Storage results to an output directory.
- virtual ~Storage (void)

  Destructor for the Storage class.

Public Attributes

- StorageType type

  The type (StorageType) of the asset.
- Interpolator interpolator

  Interpolator component of Storage.
- bool print_flag

  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_depleted

  A boolean which indicates whether or not the asset is currently considered depleted.
- bool is_sunk

  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- int n_points

  The number of points in the modelling time series.
- int n_replacements

  The number of times the asset has been replaced.
- double n_years

  The number of years being modelled.
- double power_capacity_kW

  The rated power capacity [kW] of the asset.
- double energy_capacity_kWh

  The rated energy capacity [kWh] of the asset.
- double charge_kWh

  The energy [kWh] stored in the asset.
- double power_kW

  The power [kW] currently being charged/discharged by the asset.
- double nominal_inflation_annual

  The nominal, annual inflation rate to use in computing model economics.
- double nominal_discount_annual

  The nominal, annual discount rate to use in computing model economics.
- double real_discount_annual

  The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double capital_cost

  The capital cost of the asset (undefined currency).
- double operation_maintenance_cost_kWh

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged.
- double net_present_cost

  The net present cost of this asset.
- double total_discharge_kWh

  The total energy discharged [kWh] over the Model run.
- double levellized_cost_of_energy_kWh

  The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only discharge.

- std::string type_str

  A string describing the type of the asset.
- std::vector< double > charge_vec_kWh

  A vector of the charge state [kWh] at each point in the modelling time series.
- std::vector< double > charging_power_vec_kW

  A vector of the charging power [kW] at each point in the modelling time series.
- std::vector< double > discharging_power_vec_kW

  A vector of the discharging power [kW] at each point in the modelling time series.
- std::vector< double > capital_cost_vec

  A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > operation_maintenance_cost_vec

  A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- void __checkInputs (int, double, StorageInputs)

  Helper method to check inputs to the Storage constructor.
- double __computeRealDiscountAnnual (double, double)

  Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

4.26.1 Detailed Description

The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 Storage() [1/2]

```
Storage::Storage (
            void  )
```

Constructor (dummy) for the Storage class.

```
151 {
152     return;
153 }   /* Storage() */
```

4.26.2.2 Storage() [2/2]

```
Storage::Storage (
            int n_points,
            double n_years,
            StorageInputs storage_inputs )
```

Constructor (intended) for the Storage class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
storage_inputs	A structure of Storage constructor inputs.

```
182 {
183     //  1. check inputs
184     this->__checkInputs(n_points, n_years, storage_inputs);
185 
186     //  2. set attributes
187     this->print_flag = storage_inputs.print_flag;
188     this->is_depleted = false;
189     this->is_sunk = storage_inputs.is_sunk;
190 
191     this->n_points = n_points;
192     this->n_replacements = 0;
193 
194     this->n_years = n_years;
195 
196     this->power_capacity_kW = storage_inputs.power_capacity_kW;
197     this->energy_capacity_kWh = storage_inputs.energy_capacity_kWh;
198 
199     this->charge_kWh = 0;
200     this->power_kW = 0;
201 
202     this->nominal_inflation_annual = storage_inputs.nominal_inflation_annual;
203     this->nominal_discount_annual = storage_inputs.nominal_discount_annual;
204 
205     this->real_discount_annual = this->__computeRealDiscountAnnual(
206         storage_inputs.nominal_inflation_annual,
207         storage_inputs.nominal_discount_annual
208     );
209 
210     this->capital_cost = 0;
211     this->operation_maintenance_cost_kWh = 0;
212     this->net_present_cost = 0;
213     this->total_discharge_kWh = 0;
214     this->levellized_cost_of_energy_kWh = 0;
215 
216     this->charge_vec_kWh.resize(this->n_points, 0);
217     this->charging_power_vec_kW.resize(this->n_points, 0);
218     this->discharging_power_vec_kW.resize(this->n_points, 0);
219 
220     this->capital_cost_vec.resize(this->n_points, 0);
221     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
222 
223     //  3. construction print
224     if (this->print_flag) {
225         std::cout « "Storage object constructed at " « this « std::endl;
226     }
227 
228     return;
229 }   /* Storage() */
```

4.26.2.3 ~Storage()

```
Storage::~Storage (
            void  )  [virtual]
```

Destructor for the Storage class.

```
414 {
415     //  1. destruction print
416     if (this->print_flag) {
417         std::cout « "Storage object at " « this « " destroyed" « std::endl;
418     }
419 
420     return;
421 }   /* ~Storage() */
```

4.26.3 Member Function Documentation

4.26.3.1 __checkInputs()

```
void Storage::__checkInputs (
            int n_points,
            double n_years,
            StorageInputs storage_inputs )  [private]
```

Helper method to check inputs to the Storage constructor.

Parameters

n_points	The number of points in the modelling time series.
storage_inputs	A structure of Storage constructor inputs.

```
45 {
46     //  1. check n_points
47     if (n_points <= 0) {
48         std::string error_str = "ERROR:  Storage():  n_points must be > 0";
49 
50         #ifdef _WIN32
51             std::cout << error_str << std::endl;
52         #endif
53 
54         throw std::invalid_argument(error_str);
55     }
56 
57     //  2. check n_years
58     if (n_years <= 0) {
59         std::string error_str = "ERROR:  Storage():  n_years must be > 0";
60 
61         #ifdef _WIN32
62             std::cout << error_str << std::endl;
63         #endif
64 
65         throw std::invalid_argument(error_str);
66     }
67 
68     //  3. check power_capacity_kW
69     if (storage_inputs.power_capacity_kW <= 0) {
70         std::string error_str = "ERROR:  Storage():  ";
71         error_str += "StorageInputs::power_capacity_kW must be > 0";
72 
73         #ifdef _WIN32
74             std::cout << error_str << std::endl;
75         #endif
76 
77         throw std::invalid_argument(error_str);
78     }
79 
80     //  4. check energy_capacity_kWh
81     if (storage_inputs.energy_capacity_kWh <= 0) {
82         std::string error_str = "ERROR:  Storage():  ";
83         error_str += "StorageInputs::energy_capacity_kWh must be > 0";
84 
85         #ifdef _WIN32
86             std::cout << error_str << std::endl;
87         #endif
88 
89         throw std::invalid_argument(error_str);
90     }
91 
92     return;
93 }   /* __checkInputs() */
```

4.26.3.2 __computeRealDiscountAnnual()

```
double Storage::__computeRealDiscountAnnual (
            double nominal_inflation_annual,
            double nominal_discount_annual )  [private]
```

Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

Ref: HOMER [2023h]
Ref: HOMER [2023b]

Parameters

nominal_inflation_annual	The nominal, annual inflation rate to use in computing model economics.
nominal_discount_annual	The nominal, annual discount rate to use in computing model economics.

Returns

The real, annual discount rate to use in computing model economics.

```
127 {
128     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
129     real_discount_annual /= 1 + nominal_inflation_annual;
130 
131     return real_discount_annual;
132 }   /* __computeRealDiscountAnnual() */
```

4.26.3.3 __writeSummary()

```
virtual void Storage::__writeSummary (
            std::string  )  [inline], [private], [virtual]
```

Reimplemented in Lilon.

```
79 {return;}
```

4.26.3.4 __writeTimeSeries()

```
virtual void Storage::__writeTimeSeries (
            std::string ,
            std::vector< double > * ,
            int  = -1 )  [inline], [private], [virtual]
```

Reimplemented in Lilon.

```
80 {return;}
```

4.26.3.5 commitCharge()

```
virtual void Storage::commitCharge (
            int ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Lilon.

```
134 {return;}
```

4.26.3.6 commitDischarge()

```
virtual double Storage::commitDischarge (
            int ,
            double ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in LiIon.

```
135 {return 0;}
```

4.26.3.7 computeEconomics()

```
void Storage::computeEconomics (
            std::vector< double > * time_vec_hrs_ptr )
```

Helper method to compute key economic metrics for the Model run.

Ref: HOMER [2023b]
Ref: HOMER [2023g]
Ref: HOMER [2023i]
Ref: HOMER [2023a]

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

1. compute levellized cost of energy (per unit discharged)

```
282 {
283     //  1. compute net present cost
284     double t_hrs = 0;
285     double real_discount_scalar = 0;
286 
287     for (int i = 0; i < this->n_points; i++) {
288         t_hrs = time_vec_hrs_ptr->at(i);
289 
290         real_discount_scalar = 1.0 / pow(
291             1 + this->real_discount_annual,
292             t_hrs / 8760
293         );
294 
295         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
296 
297         this->net_present_cost +=
298             real_discount_scalar * this->operation_maintenance_cost_vec[i];
299     }
300 
302     //     assuming 8,760 hours per year
303     if (this->total_discharge_kWh <= 0) {
304         this->levellized_cost_of_energy_kWh = this->net_present_cost;
305     }
306 
307     else {
308         double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
309 
310         double capital_recovery_factor =
311             (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
312             (pow(1 + this->real_discount_annual, n_years) - 1);
313 
314         double total_annualized_cost = capital_recovery_factor *
315             this->net_present_cost;
```

```
316
317         this->levellized_cost_of_energy_kWh =
318             (n_years * total_annualized_cost) /
319             this->total_discharge_kWh;
320     }
321
322     return;
323 }   /* computeEconomics() */
```

4.26.3.8 getAcceptablekW()

```
virtual double Storage::getAcceptablekW (
              double  )  [inline], [virtual]
```

Reimplemented in Lilon.

```
132 {return 0;}
```

4.26.3.9 getAvailablekW()

```
virtual double Storage::getAvailablekW (
              double  )  [inline], [virtual]
```

Reimplemented in Lilon.

```
131 {return 0;}
```

4.26.3.10 handleReplacement()

```
void Storage::handleReplacement (
              int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented in Lilon.

```
247 {
248     //  1. reset attributes
249     this->charge_kWh = 0;
250     this->power_kW = 0;
251
252     //  2. log replacement
253     this->n_replacements++;
254
255     //  3. incur capital cost in timestep
256     this->capital_cost_vec[timestep] = this->capital_cost;
257
258     return;
259 }   /* __handleReplacement() */
```

4.26.3.11 writeResults()

```
void Storage::writeResults (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            int storage_index,
            int max_lines = -1 )
```

Method which writes Storage results to an output directory.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
storage_index	An integer which corresponds to the index of the Storage asset in the Model.
max_lines	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```
360 {
361     //  1. handle sentinel
362     if (max_lines < 0) {
363         max_lines = this->n_points;
364     }
365 
366     //  2. create subdirectories
367     write_path += "Storage/";
368     if (not std::filesystem::is_directory(write_path)) {
369         std::filesystem::create_directory(write_path);
370     }
371 
372     write_path += this->type_str;
373     write_path += "_";
374     write_path += std::to_string(int(ceil(this->power_capacity_kW)));
375     write_path += "kW_";
376     write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
377     write_path += "kWh_idx";
378     write_path += std::to_string(storage_index);
379     write_path += "/";
380     std::filesystem::create_directory(write_path);
381 
382     //  3. write summary
383     this->__writeSummary(write_path);
384 
385     //  4. write time series
386     if (max_lines > this->n_points) {
387         max_lines = this->n_points;
388     }
389 
390     if (max_lines > 0) {
391         this->__writeTimeSeries(
392             write_path,
393             time_vec_hrs_ptr,
394             max_lines
395         );
396     }
397 
398     return;
399 }   /* writeResults() */
```

4.26.4 Member Data Documentation

4.26.4.1 capital_cost

```
double Storage::capital_cost
```

The capital cost of the asset (undefined currency).

4.26.4.2 capital_cost_vec

```
std::vector<double> Storage::capital_cost_vec
```

A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

4.26.4.3 charge_kWh

```
double Storage::charge_kWh
```

The energy [kWh] stored in the asset.

4.26.4.4 charge_vec_kWh

```
std::vector<double> Storage::charge_vec_kWh
```

A vector of the charge state [kWh] at each point in the modelling time series.

4.26.4.5 charging_power_vec_kW

```
std::vector<double> Storage::charging_power_vec_kW
```

A vector of the charging power [kW] at each point in the modelling time series.

4.26.4.6 discharging_power_vec_kW

```
std::vector<double> Storage::discharging_power_vec_kW
```

A vector of the discharging power [kW] at each point in the modelling time series.

4.26.4.7 energy_capacity_kWh

```
double Storage::energy_capacity_kWh
```

The rated energy capacity [kWh] of the asset.

4.26.4.8 interpolator

```
Interpolator Storage::interpolator
```

Interpolator component of Storage.

4.26.4.9 is_depleted

```
bool Storage::is_depleted
```

A boolean which indicates whether or not the asset is currently considered depleted.

4.26.4.10 is_sunk

```
bool Storage::is_sunk
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

4.26.4.11 levellized_cost_of_energy_kWh

```
double Storage::levellized_cost_of_energy_kWh
```

The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only discharge.

4.26.4.12 n_points

```
int Storage::n_points
```

The number of points in the modelling time series.

4.26.4.13 n_replacements

```
int Storage::n_replacements
```

The number of times the asset has been replaced.

4.26.4.14 n_years

```
double Storage::n_years
```

The number of years being modelled.

4.26.4.15 net_present_cost

```
double Storage::net_present_cost
```

The net present cost of this asset.

4.26.4.16 nominal_discount_annual

```
double Storage::nominal_discount_annual
```

The nominal, annual discount rate to use in computing model economics.

4.26.4.17 nominal_inflation_annual

```
double Storage::nominal_inflation_annual
```

The nominal, annual inflation rate to use in computing model economics.

4.26.4.18 operation_maintenance_cost_kWh

```
double Storage::operation_maintenance_cost_kWh
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged.

4.26.4.19 operation_maintenance_cost_vec

```
std::vector<double> Storage::operation_maintenance_cost_vec
```

A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

4.26.4.20 power_capacity_kW

```
double Storage::power_capacity_kW
```

The rated power capacity [kW] of the asset.

4.26.4.21 power_kW

```
double Storage::power_kW
```

The power [kW] currently being charged/discharged by the asset.

4.26.4.22 print_flag

```
bool Storage::print_flag
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.26.4.23 real_discount_annual

```
double Storage::real_discount_annual
```

The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.

4.26.4.24 total_discharge_kWh

```
double Storage::total_discharge_kWh
```

The total energy discharged [kWh] over the Model run.

4.26.4.25 type

```
StorageType Storage::type
```

The type (StorageType) of the asset.

4.26.4.26 type_str

```
std::string Storage::type_str
```

A string describing the type of the asset.

The documentation for this class was generated from the following files:

- header/Storage/Storage.h
- source/Storage/Storage.cpp

4.27 StorageInputs Struct Reference

A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input.

```
#include <Storage.h>
```

Public Attributes

- bool print_flag = false

  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_sunk = false

  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double power_capacity_kW = 100

  The rated power capacity [kW] of the asset.
- double energy_capacity_kWh = 1000

  The rated energy capacity [kWh] of the asset.
- double nominal_inflation_annual = 0.02

  The nominal, annual inflation rate to use in computing model economics.
- double nominal_discount_annual = 0.04

  The nominal, annual discount rate to use in computing model economics.

4.27.1 Detailed Description

A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input.

4.27.2 Member Data Documentation

4.27.2.1 energy_capacity_kWh

```
double StorageInputs::energy_capacity_kWh = 1000
```

The rated energy capacity [kWh] of the asset.

4.27.2.2 is_sunk

```
bool StorageInputs::is_sunk = false
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

4.27.2.3 nominal_discount_annual

```
double StorageInputs::nominal_discount_annual = 0.04
```

The nominal, annual discount rate to use in computing model economics.

4.27.2.4 nominal_inflation_annual

```
double StorageInputs::nominal_inflation_annual = 0.02
```

The nominal, annual inflation rate to use in computing model economics.

4.27.2.5 power_capacity_kW

```
double StorageInputs::power_capacity_kW = 100
```

The rated power capacity [kW] of the asset.

4.27.2.6 print_flag

```
bool StorageInputs::print_flag = false
```

A flag which indicates whether or not object construct/destruction should be verbose.

The documentation for this struct was generated from the following file:

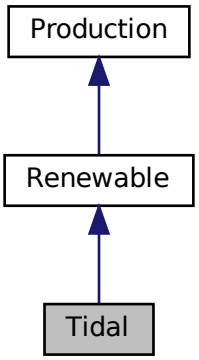
- header/Storage/Storage.h

4.28 Tidal Class Reference

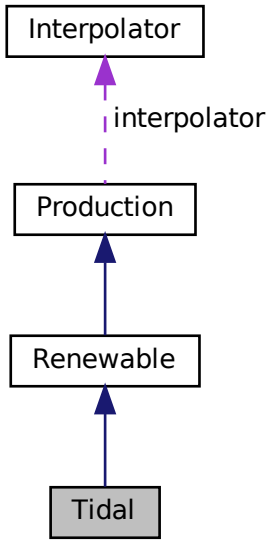
A derived class of the Renewable branch of Production which models tidal production.

```
#include <Tidal.h>
```

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- Tidal (void)

  Constructor (dummy) for the Tidal class.
- Tidal (int, double, TidalInputs, std::vector< double > *)

  Constructor (intended) for the Tidal class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

  Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Tidal (void)

  Destructor for the Tidal class.

Public Attributes

- double design_speed_ms

  The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.
- TidalPowerProductionModel power_model

  The tidal power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (TidalInputs)

  Helper method to check inputs to the Tidal constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic tidal turbine capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeCubicProductionkW (int, double, double)

  Helper method to compute tidal turbine production under a cubic production model.
- double __computeExponentialProductionkW (int, double, double)

  Helper method to compute tidal turbine production under an exponential production model.
- double __computeLookupProductionkW (int, double, double)

  Helper method to compute tidal turbine production by way of looking up using given power curve data.
- void __writeSummary (std::string)

  Helper method to write summary results for Tidal.
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

  Helper method to write time series results for Tidal.

4.28.1 Detailed Description

A derived class of the Renewable branch of Production which models tidal production.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 Tidal() [1/2]

```
Tidal::Tidal (
            void  )
```

Constructor (dummy) for the Tidal class.

```
456 {
457     return;
458 }   /* Tidal() */
```

4.28.2.2 Tidal() [2/2]

```
Tidal::Tidal (
            int n_points,
            double n_years,
            TidalInputs tidal_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Tidal class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
tidal_inputs	A structure of Tidal constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
490   :
491 Renewable(
492     n_points,
493     n_years,
494     tidal_inputs.renewable_inputs,
495     time_vec_hrs_ptr
496 )
497 {
498     //  1. check inputs
499     this->__checkInputs(tidal_inputs);
500 
501     //  2. set attributes
502     this->type = RenewableType :: TIDAL;
503     this->type_str = "TIDAL";
504 
505     this->resource_key = tidal_inputs.resource_key;
506 
507     this->design_speed_ms = tidal_inputs.design_speed_ms;
508 
509     this->power_model = tidal_inputs.power_model;
510 
511     switch (this->power_model) {
512         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
513             this->power_model_string = "CUBIC";
514 
515             break;
516         }
517 
518         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
519             this->power_model_string = "EXPONENTIAL";
520 
521             break;
522         }
523 
524         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
525             this->power_model_string = "LOOKUP";
526 
527             break;
528         }
529 
530         default: {
531             std::string error_str = "ERROR:  Tidal():  ";
532             error_str += "power production model ";
533             error_str += std::to_string(this->power_model);
534             error_str += " not recognized";
535 
536             #ifdef _WIN32
537                 std::cout « error_str « std::endl;
538             #endif
539 
540             throw std::runtime_error(error_str);
541 
542             break;
543         }
544     }
545 
546     if (tidal_inputs.capital_cost < 0) {
547         this->capital_cost = this->__getGenericCapitalCost();
548     }
549     else {
550         this->capital_cost = tidal_inputs.capital_cost;
551     }
552 
553     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
554         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
555     }
556     else {
557         this->operation_maintenance_cost_kWh =
558             tidal_inputs.operation_maintenance_cost_kWh;
559     }
560 
561     if (not this->is_sunk) {
562         this->capital_cost_vec[0] = this->capital_cost;
563     }
564 
565     //  3. construction print
```

```
566     if (this->print_flag) {
567         std::cout « "Tidal object constructed at " « this « std::endl;
568     }
569
570     return;
571 }   /* Renewable() */
```

4.28.2.3 ~Tidal()

```
Tidal::~Tidal (
            void  )
```

Destructor for the Tidal class.

```
758 {
759     //  1. destruction print
760     if (this->print_flag) {
761         std::cout « "Tidal object at " « this « " destroyed" « std::endl;
762     }
763
764     return;
765 }   /* ~Tidal() */
```

4.28.3 Member Function Documentation

4.28.3.1 __checkInputs()

```
void Tidal::__checkInputs (
            TidalInputs tidal_inputs )  [private]
```

Helper method to check inputs to the Tidal constructor.

Ref: Bir et al. [2011]
Ref: Lewis et al. [2021]

```
40 {
41     //  1. check design_speed_ms
42     if (tidal_inputs.design_speed_ms <= 0) {
43         std::string error_str = "ERROR:  Tidal():  ";
44         error_str += "TidalInputs::design_speed_ms must be > 0";
45
46         #ifdef _WIN32
47             std::cout « error_str « std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     else if (tidal_inputs.design_speed_ms < 2) {
54         std::string warning_str = "WARNING:  Tidal():  ";
55         warning_str += "Setting TidalInputs::design_speed_ms to less than 2 m/s may be ";
56         warning_str += "technically unrealistic";
57
58         std::cout « warning_str « std::endl;
59     }
60
61     return;
62 }   /* __checkInputs() */
```

4.28.3.2 __computeCubicProductionkW()

```
double Tidal::__computeCubicProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production under a cubic production model.

Ref: Buckham et al. [2023]
Ref: Bir et al. [2011]
Ref: Lewis et al. [2021]
Ref: Whitby and Ugalde-Loo [2013]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
tidal_resource_ms	The available tidal stream resource [m/s].

Returns

The production [kW] of the tidal turbine, under a cubic model.

```
152 {
153     double production = 0;
154 
155     if (
156         tidal_resource_ms < 0.15 * this->design_speed_ms or
157         tidal_resource_ms > 1.25 * this->design_speed_ms
158     ){
159         production = 0;
160     }
161 
162     else if (
163         0.15 * this->design_speed_ms <= tidal_resource_ms and
164         tidal_resource_ms <= this->design_speed_ms
165     ) {
166         production = (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
167     }
168 
169     else {
170         production = 1;
171     }
172 
173     return production * this->capacity_kW;
174 }   /* __computeCubicProductionkW() */
```

4.28.3.3 __computeExponentialProductionkW()

```
double Tidal::__computeExponentialProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production under an exponential production model.

Ref: Truelove et al. [2019]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
tidal_resource_ms	The available tidal stream resource [m/s].

Returns

The production [kW] of the tidal turbine, under an exponential model.

```
208 {
209     double production = 0;
210 
211     double turbine_speed =
212         (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
213 
214     if (turbine_speed < -0.71 or turbine_speed > 0.65) {
215         production = 0;
216     }
217 
218     else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
219         production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
220     }
221 
222     else {
223         production = 1;
224     }
225 
226     return production * this->capacity_kW;
227 }   /* __computeExponentialProductionkW() */
```

4.28.3.4 __computeLookupProductionkW()

```
double Tidal::__computeLookupProductionkW (
             int timestep,
             double dt_hrs,
             double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production by way of looking up using given power curve data.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
tidal_resource_ms	The available tidal stream resource [m/s].

Returns

The interpolated production [kW] of the tidal tubrine.

```
259 {
260     // *** WORK IN PROGRESS *** //
261 
262     return 0;
263 }   /* __computeLookupProductionkW() */
```

4.28.3.5 __getGenericCapitalCost()

```
double Tidal::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic tidal turbine capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: MacDougall [2019]

Returns

A generic capital cost for the tidal turbine [CAD].

```
84 {
85     double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;
86 
87     return capital_cost_per_kW * this->capacity_kW;
88 }   /* __getGenericCapitalCost() */
```

4.28.3.6 __getGenericOpMaintCost()

```
double Tidal::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: MacDougall [2019]

Returns

A generic operation and maintenance cost, per unit energy produced, for the tidal turbine [CAD/kWh].

```
111 {
112     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
113 
114     return operation_maintenance_cost_kWh;
115 }   /* __getGenericOpMaintCost() */
```

4.28.3.7 __writeSummary()

```
void Tidal::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Tidal.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Renewable.

```
281 {
282     //  1. create filestream
283     write_path += "summary_results.md";
284     std::ofstream ofs;
285     ofs.open(write_path, std::ofstream::out);
286 
287     //  2. write summary results (markdown)
288     ofs « "# ";
289     ofs « std::to_string(int(ceil(this->capacity_kW)));
290     ofs « " kW TIDAL Summary Results\n";
291     ofs « "\n--------\n\n";
292 
293     //  2.1. Production attributes
294     ofs « "## Production Attributes\n";
295     ofs « "\n";
296 
297     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
298     ofs « "\n";
299 
300     ofs « "Production Override: (N = 0 / Y = 1): "
301         « this->normalized_production_series_given « "  \n";
302     if (this->normalized_production_series_given) {
303         ofs « "Path to Normalized Production Time Series: "
304             « this->path_2_normalized_production_time_series « "  \n";
305     }
306     ofs « "\n";
307 
308     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
309     ofs « "Capital Cost: " « this->capital_cost « "  \n";
310     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
311         « " per kWh produced  \n";
312     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
313         « "  \n";
314     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
315         « "  \n";
316     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
317     ofs « "\n";
318 
319     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
320     ofs « "\n--------\n\n";
321 
322     //  2.2. Renewable attributes
323     ofs « "## Renewable Attributes\n";
324     ofs « "\n";
325 
326     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
327 
328     ofs « "\n--------\n\n";
329 
330     //  2.3. Tidal attributes
331     ofs « "## Tidal Attributes\n";
332     ofs « "\n";
333 
334     ofs « "Power Production Model: " « this->power_model_string « "  \n";
335     ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";
336 
337     ofs « "\n--------\n\n";
338 
339     //  2.4. Tidal Results
340     ofs « "## Results\n";
341     ofs « "\n";
342 
343     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
344     ofs « "\n";
345 
346     ofs « "Total Dispatch: " « this->total_dispatch_kWh
347         « " kWh  \n";
348 
349     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
350         « " per kWh dispatched  \n";
351     ofs « "\n";
352 
353     ofs « "Running Hours: " « this->running_hours « "  \n";
354     ofs « "Replacements: " « this->n_replacements « "  \n";
355 
356     ofs « "\n--------\n\n";
357 
```

```
358     ofs.close();
359
360     return;
361 }   /* __writeSummary() */
```

4.28.3.8 __writeTimeSeries()

```
void Tidal::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Tidal.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
max_lines	The maximum number of lines of output to write.

Reimplemented from Renewable.

```
399 {
400     //  1. create filestream
401     write_path += "time_series_results.csv";
402     std::ofstream ofs;
403     ofs.open(write_path, std::ofstream::out);
404
405     //  2. write time series results (comma separated value)
406     ofs « "Time (since start of data) [hrs],";
407     ofs « "Tidal Resource [m/s],";
408     ofs « "Production [kW],";
409     ofs « "Dispatch [kW],";
410     ofs « "Storage [kW],";
411     ofs « "Curtailment [kW],";
412     ofs « "Capital Cost (actual),";
413     ofs « "Operation and Maintenance Cost (actual),";
414     ofs « "\n";
415
416     for (int i = 0; i < max_lines; i++) {
417         ofs « time_vec_hrs_ptr->at(i) « ",";
418
419         if (not this->normalized_production_series_given) {
420             ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
421         }
422
423         else {
424             ofs « "OVERRIDE" « ",";
425         }
426
427         ofs « this->production_vec_kW[i] « ",";
428         ofs « this->dispatch_vec_kW[i] « ",";
429         ofs « this->storage_vec_kW[i] « ",";
430         ofs « this->curtailment_vec_kW[i] « ",";
431         ofs « this->capital_cost_vec[i] « ",";
432         ofs « this->operation_maintenance_cost_vec[i] « ",";
433         ofs « "\n";
434     }
435
436     return;
437 }   /* __writeTimeSeries() */
```

4.28.3.9 commit()

```
double Tidal::commit (
              int timestep,
              double dt_hrs,
              double production_kW,
              double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Renewable.

```
730 {
731     //  1. invoke base class method
732     load_kW = Renewable :: commit(
733         timestep,
734         dt_hrs,
735         production_kW,
736         load_kW
737     );
738 
739 
740     //...
741 
742     return load_kW;
743 }   /* commit() */
```

4.28.3.10 computeProductionkW()

```
double Tidal::computeProductionkW (
              int timestep,
              double dt_hrs,
              double tidal_resource_ms )  [virtual]
```

Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
tidal_resource_ms	Tidal resource (i.e. tidal stream speed) [m/s].

Returns

The production [kW] of the tidal turbine.

Reimplemented from Renewable.

```
629 {
630     //  given production time series override
631     if (this->normalized_production_series_given) {
632         double production_kW = Production :: getProductionkW(timestep);
633 
634         return production_kW;
635     }
636 
637     // check if no resource
638     if (tidal_resource_ms <= 0) {
639         return 0;
640     }
641 
642     // compute production
643     double production_kW = 0;
644 
645     switch (this->power_model) {
646         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
647             production_kW = this->__computeCubicProductionkW(
648                 timestep,
649                 dt_hrs,
650                 tidal_resource_ms
651             );
652 
653             break;
654         }
655 
656 
657         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
658             production_kW = this->__computeExponentialProductionkW(
659                 timestep,
660                 dt_hrs,
661                 tidal_resource_ms
662             );
663 
664             break;
665         }
666 
667         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
668             production_kW = this->__computeLookupProductionkW(
669                 timestep,
670                 dt_hrs,
671                 tidal_resource_ms
672             );
673 
674             break;
675         }
676 
677         default: {
678             std::string error_str = "ERROR:  Tidal::computeProductionkW():  ";
679             error_str += "power model ";
680             error_str += std::to_string(this->power_model);
681             error_str += " not recognized";
682 
683             #ifdef _WIN32
684                 std::cout « error_str « std::endl;
685             #endif
686 
687             throw std::runtime_error(error_str);
688 
689             break;
690         }
691     }
692 
693     return production_kW;
694 }   /* computeProductionkW() */
```

4.28.3.11 handleReplacement()

```
void Tidal::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
589 {
590     //  1. reset attributes
591     //...
592 
593     //  2. invoke base class method
594     Renewable :: handleReplacement(timestep);
595 
596     return;
597 }   /* __handleReplacement() */
```

4.28.4 Member Data Documentation

4.28.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.

4.28.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

4.28.4.3 power_model_string

```
std::string Tidal::power_model_string
```

A string describing the active power production model.

The documentation for this class was generated from the following files:

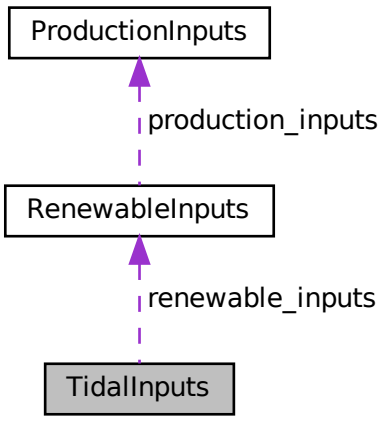
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.29 TidalInputs Struct Reference

A structure which bundles the necessary inputs for the Tidal constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

`#include <Tidal.h>`

Collaboration diagram for TidalInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double design_speed_ms = 3

  The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.
- TidalPowerProductionModel power_model = TidalPowerProductionModel :: TIDAL_POWER_CUBIC

  The tidal power production model to be applied.

4.29.1 Detailed Description

A structure which bundles the necessary inputs for the Tidal constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

4.29.2 Member Data Documentation

4.29.2.1 capital_cost

```
double TidalInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.29.2.2 design_speed_ms

```
double TidalInputs::design_speed_ms = 3
```

The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.

4.29.2.3 operation_maintenance_cost_kWh

```
double TidalInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.29.2.4 power_model

```
TidalPowerProductionModel TidalInputs::power_model = TidalPowerProductionModel ::  TIDAL_POWER_CUBIC
```

The tidal power production model to be applied.

4.29.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.29.2.6 resource_key

```
int TidalInputs::resource_key = 0
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

The documentation for this struct was generated from the following file:

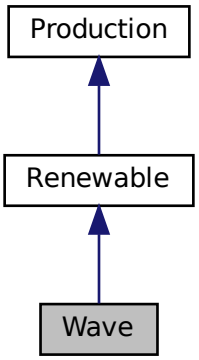
- header/Production/Renewable/Tidal.h

4.30 Wave Class Reference

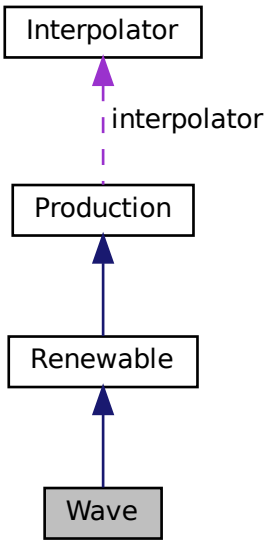
A derived class of the Renewable branch of Production which models wave production.

```
#include <Wave.h>
```

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

- Wave (void)

  Constructor (dummy) for the Wave class.
- Wave (int, double, WaveInputs, std::vector< double > *)

  Constructor (intended) for the Wave class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double, double)

  Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Wave (void)

  Destructor for the Wave class.

Public Attributes

- double design_significant_wave_height_m

  The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- double design_energy_period_s

  The energy period [s] at which the wave energy converter achieves its rated capacity.
- WavePowerProductionModel power_model

  The wave power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (WaveInputs)

  Helper method to check inputs to the Wave constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic wave energy converter capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeGaussianProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production under a Gaussian production model.
- double __computeParaboloidProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production under a paraboloid production model.
- double __computeLookupProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production by way of looking up using given performance matrix.
- void __writeSummary (std::string)

  Helper method to write summary results for Wave.
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

  Helper method to write time series results for Wave.

4.30.1 Detailed Description

A derived class of the Renewable branch of Production which models wave production.

4.30.2 Constructor & Destructor Documentation

4.30.2.1 Wave() [1/2]

```
Wave::Wave (
            void  )
```

Constructor (dummy) for the Wave class.

```
518 {
519     return;
520 }   /* Wave() */
```

4.30.2.2 Wave() [2/2]

```
Wave::Wave (
            int n_points,
            double n_years,
            WaveInputs wave_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Wave class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
wave_inputs	A structure of Wave constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
552   :
553 Renewable(
554     n_points,
555     n_years,
556     wave_inputs.renewable_inputs,
557     time_vec_hrs_ptr
558 )
559 {
560     //  1. check inputs
561     this->__checkInputs(wave_inputs);
562 
563     //  2. set attributes
564     this->type = RenewableType :: WAVE;
565     this->type_str = "WAVE";
566 
567     this->resource_key = wave_inputs.resource_key;
568 
569     this->design_significant_wave_height_m =
570         wave_inputs.design_significant_wave_height_m;
571     this->design_energy_period_s = wave_inputs.design_energy_period_s;
572 
573     this->power_model = wave_inputs.power_model;
574 
575     switch (this->power_model) {
576         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
577             this->power_model_string = "GAUSSIAN";
578 
579             break;
580         }
581 
582         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
583             this->power_model_string = "PARABOLOID";
584 
585             break;
586         }
587 
588         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
589             this->power_model_string = "LOOKUP";
590 
591             this->interpolator.addData2D(
592                 0,
593                 wave_inputs.path_2_normalized_performance_matrix
594             );
595 
596             break;
597         }
598 
599         default: {
600             std::string error_str = "ERROR:  Wave():  ";
601             error_str += "power production model ";
602             error_str += std::to_string(this->power_model);
603             error_str += " not recognized";
604 
605             #ifdef _WIN32
606                 std::cout « error_str « std::endl;
607             #endif
608 
609             throw std::runtime_error(error_str);
610 
611             break;
612         }
613     }
614 
615     if (wave_inputs.capital_cost < 0) {
616         this->capital_cost = this->__getGenericCapitalCost();
617     }
618     else {
619         this->capital_cost = wave_inputs.capital_cost;
620     }
621 
622     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
623         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
624     }
625     else {
626         this->operation_maintenance_cost_kWh =
627             wave_inputs.operation_maintenance_cost_kWh;
```

```
628     }
629
630     if (not this->is_sunk) {
631         this->capital_cost_vec[0] = this->capital_cost;
632     }
633
634     //  3. construction print
635     if (this->print_flag) {
636         std::cout « "Wave object constructed at " « this « std::endl;
637     }
638
639     return;
640 }   /* Renewable() */
```

4.30.2.3 ~Wave()

```
Wave::~Wave (
            void  )
```

Destructor for the Wave class.

```
833 {
834     //  1. destruction print
835     if (this->print_flag) {
836         std::cout « "Wave object at " « this « " destroyed" « std::endl;
837     }
838
839     return;
840 }   /* ~Wave() */
```

4.30.3 Member Function Documentation

4.30.3.1 __checkInputs()

```
void Wave::__checkInputs (
            WaveInputs wave_inputs )  [private]
```

Helper method to check inputs to the Wave constructor.

Parameters

wave_inputs	A structure of Wave constructor inputs.

```
39 {
40     //  1. check design_significant_wave_height_m
41     if (wave_inputs.design_significant_wave_height_m <= 0) {
42         std::string error_str = "ERROR:  Wave():  ";
43         error_str += "WaveInputs::design_significant_wave_height_m must be > 0";
44
45         #ifdef _WIN32
46             std::cout « error_str « std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     //  2. check design_energy_period_s
53     if (wave_inputs.design_energy_period_s <= 0) {
54         std::string error_str = "ERROR:  Wave():  ";
55         error_str += "WaveInputs::design_energy_period_s must be > 0";
56
57         #ifdef _WIN32
```

```
58              std::cout « error_str « std::endl;
59          #endif
60
61          throw std::invalid_argument(error_str);
62      }
63
64      //  3. if WAVE_POWER_LOOKUP, check that path is given
65      if (
66          wave_inputs.power_model == WavePowerProductionModel :: WAVE_POWER_LOOKUP and
67          wave_inputs.path_2_normalized_performance_matrix.empty()
68      ) {
69          std::string error_str = "ERROR:  Wave()  power model was set to ";
70          error_str += "WavePowerProductionModel::WAVE_POWER_LOOKUP, but no path to a ";
71          error_str += "normalized performance matrix was given";
72
73          #ifdef _WIN32
74              std::cout « error_str « std::endl;
75          #endif
76
77          throw std::invalid_argument(error_str);
78      }
79
80      return;
81  }   /* __checkInputs() */
```

4.30.3.2 __computeGaussianProductionkW()

```
double Wave::__computeGaussianProductionkW (
              int timestep,
              double dt_hrs,
              double significant_wave_height_m,
              double energy_period_s )  [private]
```

Helper method to compute wave energy converter production under a Gaussian production model.

Ref: Truelove et al. [2019]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under an exponential model.

```
176 {
177     double H_s_nondim =
178         (significant_wave_height_m - this->design_significant_wave_height_m) /
179         this->design_significant_wave_height_m;
180
181     double T_e_nondim =
182         (energy_period_s - this->design_energy_period_s) /
183         this->design_energy_period_s;
184
185     double production = exp(
186         -2.25119 * pow(T_e_nondim, 2) +
187         3.44570 * T_e_nondim * H_s_nondim -
188         4.01508 * pow(H_s_nondim, 2)
189     );
190
191     return production * this->capacity_kW;
```

```
192 }   /* __computeGaussianProductionkW() */
```

4.30.3.3 __computeLookupProductionkW()

```
double Wave::__computeLookupProductionkW (
            int timestep,
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [private]
```

Helper method to compute wave energy converter production by way of looking up using given performance matrix.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	The energy period [s] in the vicinity of the wave energy converter

Returns

The interpolated production [kW] of the wave energy converter.

```
293 {
294     double prod = this->interpolator.interp2D(
295         0,
296         significant_wave_height_m,
297         energy_period_s
298     );
299 
300     return prod * this->capacity_kW;
301 }   /* __computeLookupProductionkW() */
```

4.30.3.4 __computeParaboloidProductionkW()

```
double Wave::__computeParaboloidProductionkW (
            int timestep,
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [private]
```

Helper method to compute wave energy converter production under a paraboloid production model.

Ref: Robertson et al. [2021]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under a paraboloid model.

```
233 {
234     // first, check for idealized wave breaking (deep water)
235     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
236         return 0;
237     }
238 
239     // otherwise, apply generic quadratic performance model
240     // (with outputs bounded to [0, 1])
241     double production =
242         0.289 * significant_wave_height_m -
243         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
244         0.0169 * energy_period_s;
245 
246     if (production < 0) {
247         production = 0;
248     }
249 
250     else if (production > 1) {
251         production = 1;
252     }
253 
254     return production * this->capacity_kW;
255 }   /* __computeParaboloidProductionkW() */
```

4.30.3.5 __getGenericCapitalCost()

```
double Wave::__getGenericCapitalCost (
            void  )  [private]
```

Helper method to generate a generic wave energy converter capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: MacDougall [2019]

Returns

A generic capital cost for the wave energy converter [CAD].

```
103 {
104     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
105 
106     return capital_cost_per_kW * this->capacity_kW;
107 }   /* __getGenericCapitalCost() */
```

4.30.3.6 __getGenericOpMaintCost()

```
double Wave::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: MacDougall [2019]

Returns

A generic operation and maintenance cost, per unit energy produced, for the wave energy converter [CAD/k↩ Wh].

```
131 {
132     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
133 
134     return operation_maintenance_cost_kWh;
135 }   /* __getGenericOpMaintCost() */
```

4.30.3.7 __writeSummary()

```
void Wave::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Wave.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Renewable.

```
319 {
320     //  1. create filestream
321     write_path += "summary_results.md";
322     std::ofstream ofs;
323     ofs.open(write_path, std::ofstream::out);
324 
325     //  2. write summary results (markdown)
326     ofs « "# ";
327     ofs « std::to_string(int(ceil(this->capacity_kW)));
328     ofs « " kW WAVE Summary Results\n";
329     ofs « "\n--------\n\n";
330 
331     //  2.1. Production attributes
332     ofs « "## Production Attributes\n";
333     ofs « "\n";
334 
335     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
336     ofs « "\n";
337 
338     ofs « "Production Override: (N = 0 / Y = 1): "
339         « this->normalized_production_series_given « "  \n";
340     if (this->normalized_production_series_given) {
341         ofs « "Path to Normalized Production Time Series: "
342             « this->path_2_normalized_production_time_series « "  \n";
343     }
344     ofs « "\n";
345 
346     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
347     ofs « "Capital Cost: " « this->capital_cost « "  \n";
348     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
349         « " per kWh produced  \n";
350     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
351         « "  \n";
352     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
353         « "  \n";
354     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
355     ofs « "\n";
356 
357     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
358     ofs « "\n--------\n\n";
359 
360     //  2.2. Renewable attributes
361     ofs « "## Renewable Attributes\n";
362     ofs « "\n";
363 
364     ofs « "Resource Key (2D): " « this->resource_key « "  \n";
365 
366     ofs « "\n--------\n\n";
367 
368     //  2.3. Wave attributes
369     ofs « "## Wave Attributes\n";
370     ofs « "\n";
371 
372     ofs « "Power Production Model: " « this->power_model_string « "  \n";
373     switch (this->power_model) {
374         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
375             ofs « "Design Significant Wave Height: "
376                 « this->design_significant_wave_height_m « " m  \n";
377 
378             ofs « "Design Energy Period: " « this->design_energy_period_s « " s  \n";
379 
380             break;
381         }
382 
383         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
```

```
384             ofs « "Normalized Performance Matrix: "
385                 « this->interpolator.path_map_2D[0] « "  \n";
386
387             break;
388         }
389
390         default: {
391             // write nothing!
392
393             break;
394         }
395     }
396
397     ofs « "\n--------\n\n";
398
399     //  2.4. Wave Results
400     ofs « "## Results\n";
401     ofs « "\n";
402
403     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
404     ofs « "\n";
405
406     ofs « "Total Dispatch: " « this->total_dispatch_kWh
407         « " kWh  \n";
408
409     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
410         « " per kWh dispatched  \n";
411     ofs « "\n";
412
413     ofs « "Running Hours: " « this->running_hours « "  \n";
414     ofs « "Replacements: " « this->n_replacements « "  \n";
415
416     ofs « "\n--------\n\n";
417
418     ofs.close();
419
420     return;
421 }   /* __writeSummary() */
```

4.30.3.8 __writeTimeSeries()

```
void Wave::__writeTimeSeries (
              std::string write_path,
              std::vector< double > * time_vec_hrs_ptr,
              std::map< int, std::vector< double >> * resource_map_1D_ptr,
              std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
              int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Wave.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
max_lines	The maximum number of lines of output to write.

Reimplemented from Renewable.

```
459 {
460     //  1. create filestream
461     write_path += "time_series_results.csv";
462     std::ofstream ofs;
463     ofs.open(write_path, std::ofstream::out);
464
465     //  2. write time series results (comma separated value)
```

```
466     ofs « "Time (since start of data) [hrs],";
467     ofs « "Significant Wave Height [m],";
468     ofs « "Energy Period [s],";
469     ofs « "Production [kW],";
470     ofs « "Dispatch [kW],";
471     ofs « "Storage [kW],";
472     ofs « "Curtailment [kW],";
473     ofs « "Capital Cost (actual),";
474     ofs « "Operation and Maintenance Cost (actual),";
475     ofs « "\n";
476 
477     for (int i = 0; i < max_lines; i++) {
478         ofs « time_vec_hrs_ptr->at(i) « ",";
479 
480         if (not this->normalized_production_series_given) {
481             ofs « resource_map_2D_ptr->at(this->resource_key)[i][0] « ",";
482             ofs « resource_map_2D_ptr->at(this->resource_key)[i][1] « ",";
483         }
484 
485         else {
486             ofs « "OVERRIDE" « ",";
487             ofs « "OVERRIDE" « ",";
488         }
489 
490         ofs « this->production_vec_kW[i] « ",";
491         ofs « this->dispatch_vec_kW[i] « ",";
492         ofs « this->storage_vec_kW[i] « ",";
493         ofs « this->curtailment_vec_kW[i] « ",";
494         ofs « this->capital_cost_vec[i] « ",";
495         ofs « this->operation_maintenance_cost_vec[i] « ",";
496         ofs « "\n";
497     }
498 
499     return;
500 }   /* __writeTimeSeries() */
```

4.30.3.9 commit()

```
double Wave::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Renewable.

```
805 {
806     //  1. invoke base class method
807     load_kW = Renewable :: commit(
808         timestep,
809         dt_hrs,
810         production_kW,
```

```
811         load_kW
812     );
813
814
815     //...
816
817     return load_kW;
818 }   /* commit() */
```

4.30.3.10 computeProductionkW()

```
double Wave::computeProductionkW (
             int timestep,
             double dt_hrs,
             double significant_wave_height_m,
             double energy_period_s )  [virtual]
```

Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
signficiant_wave_height_m	The significant wave height (wave statistic) [m].
energy_period_s	The energy period (wave statistic) [s].

Returns

The production [kW] of the wave turbine.

Reimplemented from Renewable.

```
702 {
703     //  given production time series override
704     if (this->normalized_production_series_given) {
705         double production_kW = Production :: getProductionkW(timestep);
706
707         return production_kW;
708     }
709
710     // check if no resource
711     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
712         return 0;
713     }
714
715     // compute production
716     double production_kW = 0;
717
718     switch (this->power_model) {
719         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
720             production_kW = this->__computeParaboloidProductionkW(
721                 timestep,
722                 dt_hrs,
723                 significant_wave_height_m,
724                 energy_period_s
725             );
726
727             break;
728         }
729
730         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
731             production_kW = this->__computeGaussianProductionkW(
732                 timestep,
733                 dt_hrs,
```

```
734                 significant_wave_height_m,
735                 energy_period_s
736             );
737 
738             break;
739         }
740 
741         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
742             production_kW = this->__computeLookupProductionkW(
743                 timestep,
744                 dt_hrs,
745                 significant_wave_height_m,
746                 energy_period_s
747             );
748 
749             break;
750         }
751 
752         default: {
753             std::string error_str = "ERROR:  Wave::computeProductionkW():  ";
754             error_str += "power model ";
755             error_str += std::to_string(this->power_model);
756             error_str += " not recognized";
757 
758             #ifdef _WIN32
759                 std::cout « error_str « std::endl;
760             #endif
761 
762             throw std::runtime_error(error_str);
763 
764             break;
765         }
766     }
767 
768     return production_kW;
769 }   /* computeProductionkW() */
```

4.30.3.11 handleReplacement()

```
void Wave::handleReplacement (
              int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
658 {
659     //  1. reset attributes
660     //...
661 
662     //  2. invoke base class method
663     Renewable :: handleReplacement(timestep);
664 
665     return;
666 }   /* __handleReplacement() */
```

4.30.4 Member Data Documentation

4.30.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

The energy period [s] at which the wave energy converter achieves its rated capacity.

4.30.4.2 design_significant_wave_height_m

```
double Wave::design_significant_wave_height_m
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

4.30.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

4.30.4.4 power_model_string

```
std::string Wave::power_model_string
```

A string describing the active power production model.

The documentation for this class was generated from the following files:

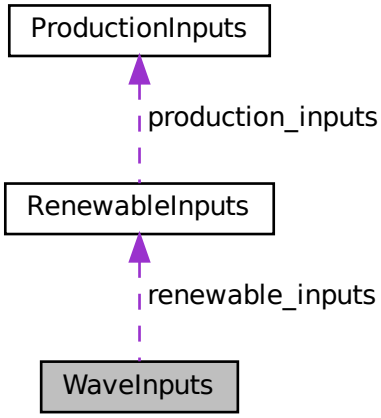
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

4.31 WaveInputs Struct Reference

A structure which bundles the necessary inputs for the Wave constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

`#include <Wave.h>`

Collaboration diagram for WaveInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double design_significant_wave_height_m = 3

  The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- double design_energy_period_s = 10

  The energy period [s] at which the wave energy converter achieves its rated capacity.
- WavePowerProductionModel power_model = WavePowerProductionModel :: WAVE_POWER_PARABOLOID

  The wave power production model to be applied.
- std::string path_2_normalized_performance_matrix = ""

  A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.1 Detailed Description

A structure which bundles the necessary inputs for the Wave constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

4.31.2 Member Data Documentation

4.31.2.1 capital_cost

```
double WaveInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.31.2.2 design_energy_period_s

```
double WaveInputs::design_energy_period_s = 10
```

The energy period [s] at which the wave energy converter achieves its rated capacity.

4.31.2.3 design_significant_wave_height_m

```
double WaveInputs::design_significant_wave_height_m = 3
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

4.31.2.4 operation_maintenance_cost_kWh

```
double WaveInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.31.2.5 path_2_normalized_performance_matrix

```
std::string WaveInputs::path_2_normalized_performance_matrix = ""
```

A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.2.6 power_model

```
WavePowerProductionModel WaveInputs::power_model = WavePowerProductionModel ::  WAVE_POWER_PARABOLOID
```

The wave power production model to be applied.

4.31.2.7 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.31.2.8 resource_key

```
int WaveInputs::resource_key = 0
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

The documentation for this struct was generated from the following file:

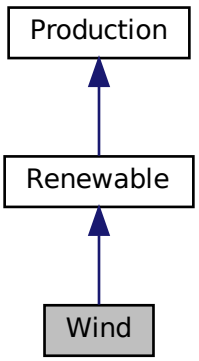
- header/Production/Renewable/Wave.h

4.32 Wind Class Reference

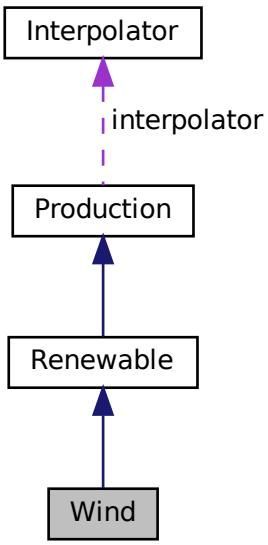
A derived class of the Renewable branch of Production which models wind production.

`#include <Wind.h>`

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- Wind (void)

  Constructor (dummy) for the Wind class.
- Wind (int, double, WindInputs, std::vector< double > *)

  Constructor (intended) for the Wind class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

  Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ~Wind (void)

  Destructor for the Wind class.

Public Attributes

- double design_speed_ms

  The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- WindPowerProductionModel power_model

  The wind power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (WindInputs)

  Helper method to check inputs to the Wind constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic wind turbine capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeCubicProductionkW (int, double, double)

  Helper method to compute wind turbine production under a cubic production model.
- double __computeExponentialProductionkW (int, double, double)

  Helper method to compute wind turbine production under an exponential production model.
- double __computeLookupProductionkW (int, double, double)

  Helper method to compute wind turbine production by way of looking up using given power curve data.
- void __writeSummary (std::string)

  Helper method to write summary results for Wind.
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

  Helper method to write time series results for Wind.

4.32.1 Detailed Description

A derived class of the Renewable branch of Production which models wind production.

4.32.2 Constructor & Destructor Documentation

4.32.2.1 Wind() [1/2]

```
Wind::Wind (
            void  )
```

Constructor (dummy) for the Wind class.

```
476 {
477     return;
478 }   /* Wind() */
```

4.32.2.2 Wind() [2/2]

```
Wind::Wind (
            int n_points,
            double n_years,
            WindInputs wind_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Wind class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
wind_inputs	A structure of Wind constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
510   :
511 Renewable(
512     n_points,
513     n_years,
514     wind_inputs.renewable_inputs,
515     time_vec_hrs_ptr
516 )
517 {
518     //  1. check inputs
519     this->__checkInputs(wind_inputs);
520 
521     //  2. set attributes
522     this->type = RenewableType :: WIND;
523     this->type_str = "WIND";
524 
525     this->resource_key = wind_inputs.resource_key;
526 
527     this->design_speed_ms = wind_inputs.design_speed_ms;
528 
529     this->power_model = wind_inputs.power_model;
530 
531     switch (this->power_model) {
532         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
533             this->power_model_string = "CUBIC";
534 
535             break;
536         }
537 
538         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
539             this->power_model_string = "EXPONENTIAL";
540 
```

```
541             break;
542         }
543 
544         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
545             this->power_model_string = "LOOKUP";
546 
547             break;
548         }
549 
550         default: {
551             std::string error_str = "ERROR:  Wind():  ";
552             error_str += "power production model ";
553             error_str += std::to_string(this->power_model);
554             error_str += " not recognized";
555 
556             #ifdef _WIN32
557                 std::cout << error_str << std::endl;
558             #endif
559 
560             throw std::runtime_error(error_str);
561 
562             break;
563         }
564     }
565 
566     if (wind_inputs.capital_cost < 0) {
567         this->capital_cost = this->__getGenericCapitalCost();
568     }
569     else {
570         this->capital_cost = wind_inputs.capital_cost;
571     }
572 
573     if (wind_inputs.operation_maintenance_cost_kWh < 0) {
574         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
575     }
576     else {
577         this->operation_maintenance_cost_kWh =
578             wind_inputs.operation_maintenance_cost_kWh;
579     }
580 
581     if (not this->is_sunk) {
582         this->capital_cost_vec[0] = this->capital_cost;
583     }
584 
585     //  3. construction print
586     if (this->print_flag) {
587         std::cout << "Wind object constructed at " << this << std::endl;
588     }
589 
590     return;
591 }   /* Renewable() */
```

4.32.2.3 ~Wind()

```
Wind::~Wind (
            void  )
```

Destructor for the Wind class.

```
777 {
778     //  1. destruction print
779     if (this->print_flag) {
780         std::cout << "Wind object at " << this << " destroyed" << std::endl;
781     }
782 
783     return;
784 }   /* ~Wind() */
```

4.32.3 Member Function Documentation

4.32.3.1 __checkInputs()

```
void Wind::__checkInputs (
            WindInputs wind_inputs )  [private]
```

Helper method to check inputs to the Wind constructor.

Ref: Zafar [2018]

Parameters

wind_inputs	A structure of Wind constructor inputs.

```
41 {
42     //  1. check design_speed_ms
43     if (wind_inputs.design_speed_ms <= 0) {
44         std::string error_str = "ERROR:  Wind():  ";
45         error_str += "WindInputs::design_speed_ms must be > 0";
46 
47         #ifdef _WIN32
48             std::cout « error_str « std::endl;
49         #endif
50 
51         throw std::invalid_argument(error_str);
52     }
53 
54     else if (wind_inputs.design_speed_ms < 12) {
55         std::string warning_str = "WARNING:  Wind():  ";
56         warning_str += "Setting WindInputs::design_speed_ms to less than 12 m/s may be ";
57         warning_str += "technically unrealistic";
58 
59         std::cout « warning_str « std::endl;
60     }
61 
62     return;
63 }   /* __checkInputs() */
```

4.32.3.2 __computeCubicProductionkW()

```
double Wind::__computeCubicProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms )  [private]
```

Helper method to compute wind turbine production under a cubic production model.

Ref: Milan et al. [2010]
Ref: Zafar [2018]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

The production [kW] of the wind turbine, under an exponential model.

```
151 {
152     double production = 0;
153 
154     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
155         this->design_speed_ms;
156 
157     if (turbine_speed < -0.7857 or turbine_speed > 0.7857) {
158         production = 0;
159     }
160 
161     else if (turbine_speed >= -0.7857 and turbine_speed <= 0) {
162         production = (1 / pow(this->design_speed_ms, 3)) * pow(wind_resource_ms, 3);
163     }
164 
165     else {
166         production = 1;
167     }
168 
169     return production * this->capacity_kW;
170 }   /* __computeCubicProductionkW() */
```

4.32.3.3 __computeExponentialProductionkW()

```
double Wind::__computeExponentialProductionkW (
              int timestep,
              double dt_hrs,
              double wind_resource_ms )  [private]
```

Helper method to compute wind turbine production under an exponential production model.

Ref: Truelove et al. [2019]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

The production [kW] of the wind turbine, under an exponential model.

```
204 {
205     double production = 0;
206 
207     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
208         this->design_speed_ms;
209 
210     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
211         production = 0;
212     }
213 
214     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
215         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
216     }
217 
218     else {
219         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
220     }
221 
222     return production * this->capacity_kW;
223 }   /* __computeExponentialProductionkW() */
```

4.32.3.4 __computeLookupProductionkW()

```
double Wind::__computeLookupProductionkW (
              int timestep,
              double dt_hrs,
              double wind_resource_ms )  [private]
```

Helper method to compute wind turbine production by way of looking up using given power curve data.

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

The interpolated production [kW] of the wind turbine.

```
255 {
256     // *** WORK IN PROGRESS *** //
257
258     return 0;
259 }   /* __computeLookupProductionkW() */
```

4.32.3.5 __getGenericCapitalCost()

```
double Wind::__getGenericCapitalCost (
              void  )  [private]
```

Helper method to generate a generic wind turbine capital cost.

This model was obtained by way of surveying an assortment of published wind turbine costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD].

Returns

A generic capital cost for the wind turbine [CAD].

```
85 {
86     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
87
88     return capital_cost_per_kW * this->capacity_kW;
89 }   /* __getGenericCapitalCost() */
```

4.32.3.6 __getGenericOpMaintCost()

```
double Wind::__getGenericOpMaintCost (
            void  )  [private]
```

Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

This model was obtained by way of surveying an assortment of published wind turbine costs, and then constructing a best fit model. Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Returns

A generic operation and maintenance cost, per unit energy produced, for the wind turbine [CAD/kWh].

```
112 {
113     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
114 
115     return operation_maintenance_cost_kWh;
116 }   /* __getGenericOpMaintCost() */
```

4.32.3.7 __writeSummary()

```
void Wind::__writeSummary (
            std::string write_path )  [private], [virtual]
```

Helper method to write summary results for Wind.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.

Reimplemented from Renewable.

```
277 {
278     //  1. create filestream
279     write_path += "summary_results.md";
280     std::ofstream ofs;
281     ofs.open(write_path, std::ofstream::out);
282 
283     //  2. write summary results (markdown)
284     ofs « "# ";
285     ofs « std::to_string(int(ceil(this->capacity_kW)));
286     ofs « " kW WIND Summary Results\n";
287     ofs « "\n--------\n\n";
288 
289 
290     //  2.1. Production attributes
291     ofs « "## Production Attributes\n";
292     ofs « "\n";
293 
294     ofs « "Capacity: " « this->capacity_kW « " kW  \n";
295     ofs « "\n";
296 
297     ofs « "Production Override: (N = 0 / Y = 1): "
298         « this->normalized_production_series_given « "  \n";
299     if (this->normalized_production_series_given) {
300         ofs « "Path to Normalized Production Time Series: "
301             « this->path_2_normalized_production_time_series « "  \n";
302     }
303     ofs « "\n";
304 
305     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
306     ofs « "Capital Cost: " « this->capital_cost « "  \n";
```

```
307     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
308         « " per kWh produced  \n";
309     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
310         « "  \n";
311     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
312         « "  \n";
313     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
314     ofs « "\n";
315 
316     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
317     ofs « "\n--------\n\n";
318 
319     //  2.2. Renewable attributes
320     ofs « "## Renewable Attributes\n";
321     ofs « "\n";
322 
323     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
324 
325     ofs « "\n--------\n\n";
326 
327     //  2.3. Wind attributes
328     ofs « "## Wind Attributes\n";
329     ofs « "\n";
330 
331     ofs « "Power Production Model: " « this->power_model_string « "  \n";
332     switch (this->power_model) {
333         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
334             ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";
335 
336             break;
337         }
338 
339         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
340             ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";
341 
342             break;
343         }
344 
345         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
346             //...
347 
348             break;
349         }
350 
351         default: {
352             // write nothing!
353 
354             break;
355         }
356     }
357 
358     ofs « "\n--------\n\n";
359 
360     //  2.4. Wind Results
361     ofs « "## Results\n";
362     ofs « "\n";
363 
364     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
365     ofs « "\n";
366 
367     ofs « "Total Dispatch: " « this->total_dispatch_kWh
368         « " kWh  \n";
369 
370     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
371         « " per kWh dispatched  \n";
372     ofs « "\n";
373 
374     ofs « "Running Hours: " « this->running_hours « "  \n";
375     ofs « "Replacements: " « this->n_replacements « "  \n";
376 
377     ofs « "\n--------\n\n";
378 
379     ofs.close();
380 
381     return;
382 }   /* __writeSummary() */
```

4.32.3.8 __writeTimeSeries()

```
void Wind::__writeTimeSeries (
            std::string write_path,
```

```
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Wind.

Parameters

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
max_lines	The maximum number of lines of output to write.

Reimplemented from Renewable.

```
420 {
421     //  1. create filestream
422     write_path += "time_series_results.csv";
423     std::ofstream ofs;
424     ofs.open(write_path, std::ofstream::out);
425 
426     //  2. write time series results (comma separated value)
427     ofs « "Time (since start of data) [hrs],";
428     ofs « "Wind Resource [m/s],";
429     ofs « "Production [kW],";
430     ofs « "Dispatch [kW],";
431     ofs « "Storage [kW],";
432     ofs « "Curtailment [kW],";
433     ofs « "Capital Cost (actual),";
434     ofs « "Operation and Maintenance Cost (actual),";
435     ofs « "\n";
436 
437     for (int i = 0; i < max_lines; i++) {
438         ofs « time_vec_hrs_ptr->at(i) « ",";
439 
440         if (not this->normalized_production_series_given) {
441             ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
442         }
443 
444         else {
445             ofs « "OVERRIDE" « ",";
446         }
447 
448         ofs « this->production_vec_kW[i] « ",";
449         ofs « this->dispatch_vec_kW[i] « ",";
450         ofs « this->storage_vec_kW[i] « ",";
451         ofs « this->curtailment_vec_kW[i] « ",";
452         ofs « this->capital_cost_vec[i] « ",";
453         ofs « this->operation_maintenance_cost_vec[i] « ",";
454         ofs « "\n";
455     }
456 
457     return;
458 }   /* __writeTimeSeries() */
```

4.32.3.9 commit()

```
double Wind::commit (
            int timestep,
            double dt_hrs,
            double production_kW,
            double load_kW )  [virtual]
```

Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
load_kW	The load [kW] passed to the asset in this timestep.

Returns

The load [kW] remaining after the dispatch is deducted from it.

Reimplemented from Renewable.

```
749 {
750     //  1. invoke base class method
751     load_kW = Renewable :: commit(
752         timestep,
753         dt_hrs,
754         production_kW,
755         load_kW
756     );
757 
758 
759     //...
760 
761     return load_kW;
762 }   /* commit() */
```

4.32.3.10 computeProductionkW()

```
double Wind::computeProductionkW (
              int timestep,
              double dt_hrs,
              double wind_resource_ms )  [virtual]
```

Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
wind_resource_ms	Wind resource (i.e. wind speed) [m/s].

Returns

The production [kW] of the wind turbine.

Reimplemented from Renewable.

```
649 {
650     //  given production time series override
651     if (this->normalized_production_series_given) {
652         double production_kW = Production :: getProductionkW(timestep);
653 
654         return production_kW;
655     }
656 
```

```
657     // check if no resource
658     if (wind_resource_ms <= 0) {
659         return 0;
660     }
661 
662     // compute production
663     double production_kW = 0;
664 
665     switch (this->power_model) {
666         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
667             production_kW = this->__computeCubicProductionkW(
668                 timestep,
669                 dt_hrs,
670                 wind_resource_ms
671             );
672 
673             break;
674         }
675 
676         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
677             production_kW = this->__computeExponentialProductionkW(
678                 timestep,
679                 dt_hrs,
680                 wind_resource_ms
681             );
682 
683             break;
684         }
685 
686         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
687             production_kW = this->__computeLookupProductionkW(
688                 timestep,
689                 dt_hrs,
690                 wind_resource_ms
691             );
692 
693             break;
694         }
695 
696         default: {
697             std::string error_str = "ERROR:  Wind::computeProductionkW():  ";
698             error_str += "power model ";
699             error_str += std::to_string(this->power_model);
700             error_str += " not recognized";
701 
702             #ifdef _WIN32
703                 std::cout « error_str « std::endl;
704             #endif
705 
706             throw std::runtime_error(error_str);
707 
708             break;
709         }
710     }
711 
712     return production_kW;
713 }   /* computeProductionkW() */
```

4.32.3.11 handleReplacement()

```
void Wind::handleReplacement (
            int timestep )  [virtual]
```

Method to handle asset replacement and capital cost incursion, if applicable.

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
609 {
610     //  1. reset attributes
611     //...
```

```
612
613     //  2. invoke base class method
614     Renewable :: handleReplacement(timestep);
615
616     return;
617 }   /* __handleReplacement() */
```

4.32.4 Member Data Documentation

4.32.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

The wind speed [m/s] at which the wind turbine achieves its rated capacity.

4.32.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

4.32.4.3 power_model_string

```
std::string Wind::power_model_string
```

A string describing the active power production model.

The documentation for this class was generated from the following files:

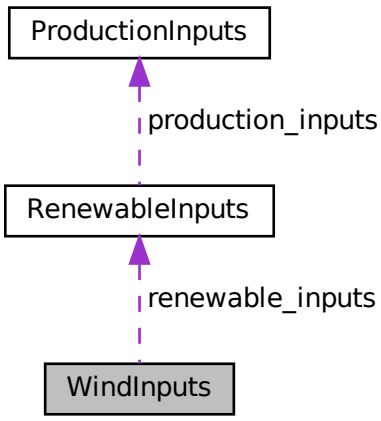
- header/Production/Renewable/Wind.h
- source/Production/Renewable/Wind.cpp

4.33 WindInputs Struct Reference

A structure which bundles the necessary inputs for the Wind constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

`#include <Wind.h>`

Collaboration diagram for WindInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

  The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double operation_maintenance_cost_kWh = -1

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double design_speed_ms = 14

  The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- WindPowerProductionModel power_model = WindPowerProductionModel :: WIND_POWER_CUBIC

  The wind power production model to be applied.

4.33.1 Detailed Description

A structure which bundles the necessary inputs for the Wind constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.

4.33.2 Member Data Documentation

4.33.2.1 capital_cost

```
double WindInputs::capital_cost = -1
```

The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].

4.33.2.2 design_speed_ms

```
double WindInputs::design_speed_ms = 14
```

The wind speed [m/s] at which the wind turbine achieves its rated capacity.

4.33.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].

4.33.2.4 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel ::  WIND_POWER_CUBIC
```

The wind power production model to be applied.

4.33.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.33.2.6 resource_key

```
int WindInputs::resource_key = 0
```

A key used to index into the Resources object, to associate this asset with the appropriate resource time series.

The documentation for this struct was generated from the following file:

- header/Production/Renewable/Wind.h

Chapter 5

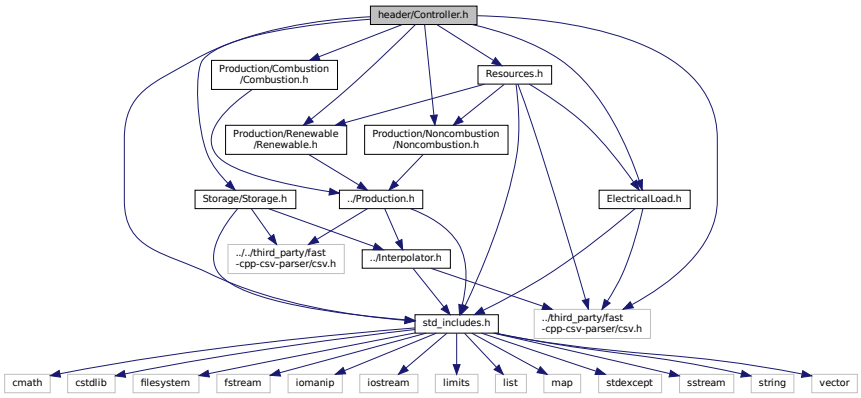
File Documentation

5.1 header/Controller.h File Reference

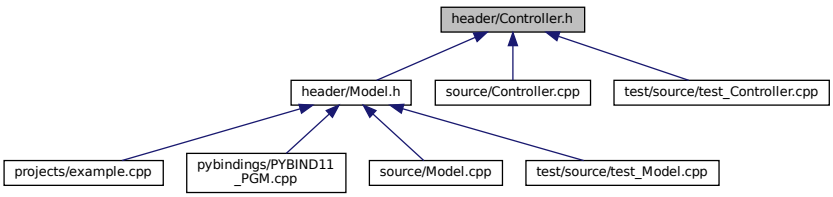
Header file for the Controller class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Combustion.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"
#include "Storage/Storage.h"
```

Include dependency graph for Controller.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Controller

  A class which contains a various dispatch control logic. Intended to serve as a component class of Model.

Enumerations

- enum ControlMode { LOAD_FOLLOWING , CYCLE_CHARGING , N_CONTROL_MODES }

  An enumeration of the types of control modes supported by PGMcpp.

5.1.1 Detailed Description

Header file for the Controller class.

5.1.2 Enumeration Type Documentation

5.1.2.1 ControlMode

```
enum ControlMode
```

An enumeration of the types of control modes supported by PGMcpp.

Enumerator

LOAD_FOLLOWING	Load following control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
CYCLE_CHARGING	Cycle charging control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
N_CONTROL_MODES	A simple hack to get the number of elements in ControlMode.

```
44                  {
45     LOAD_FOLLOWING,
46     CYCLE_CHARGING,
47     N_CONTROL_MODES
48 };
```

5.2 header/doxygen_cite.h File Reference

Header file which simply cites the doxygen tool.

5.2.1 Detailed Description

Header file which simply cites the doxygen tool.

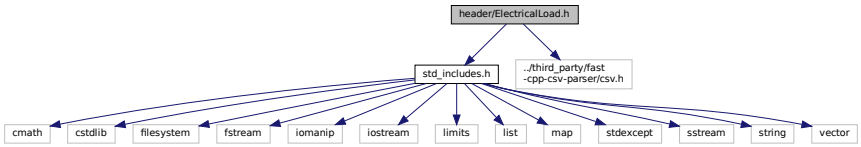
Ref: van Heesch. [2023]

5.3 header/ElectricalLoad.h File Reference

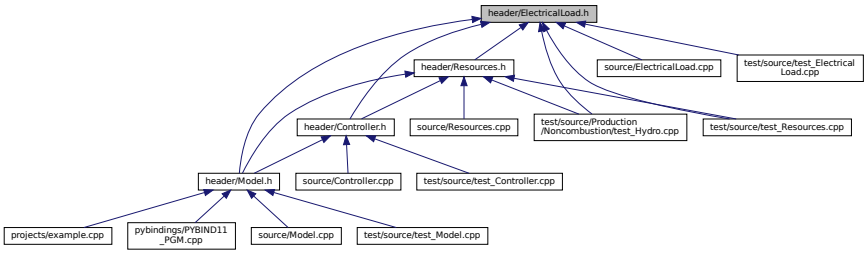
Header file for the ElectricalLoad class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
```

Include dependency graph for ElectricalLoad.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ElectricalLoad

  A class which contains time and electrical load data. Intended to serve as a component class of Model.

5.3.1 Detailed Description

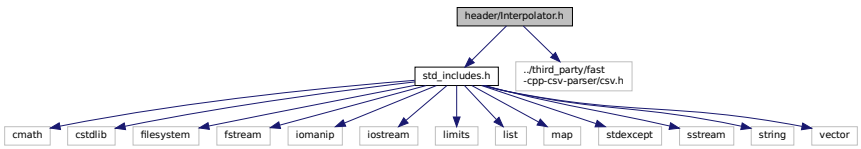
Header file for the ElectricalLoad class.

5.4 header/Interpolator.h File Reference

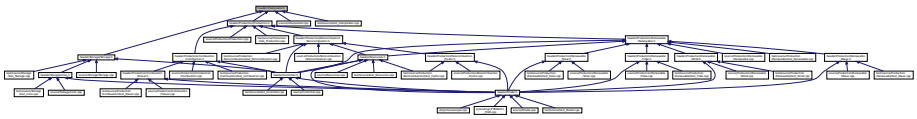
Header file for the Interpolator class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
```

Include dependency graph for Interpolator.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct InterpolatorStruct1D

  A struct which holds two parallel vectors for use in 1D interpolation.
- struct InterpolatorStruct2D

  A struct which holds two parallel vectors and a matrix for use in 2D interpolation.
- class Interpolator

  A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies.

5.4.1 Detailed Description

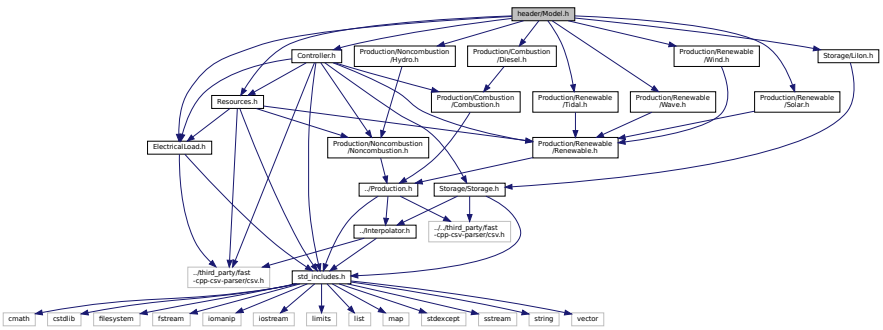
Header file for the Interpolator class.

5.5 header/Model.h File Reference

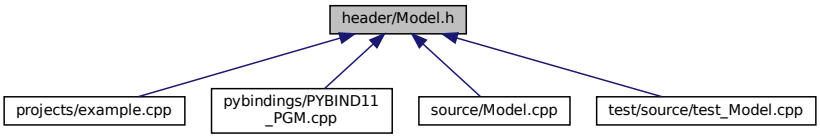
Header file for the Model class.

```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Noncombustion/Hydro.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
#include "Production/Renewable/Wave.h"
#include "Production/Renewable/Wind.h"
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct ModelInputs

  A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).
- class Model

  A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

5.5.1 Detailed Description

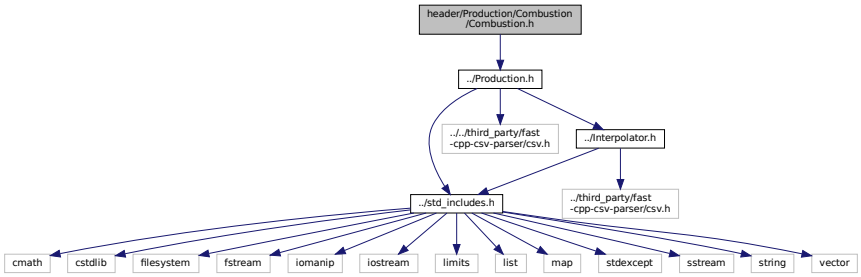
Header file for the Model class.

5.6 header/Production/Combustion/Combustion.h File Reference

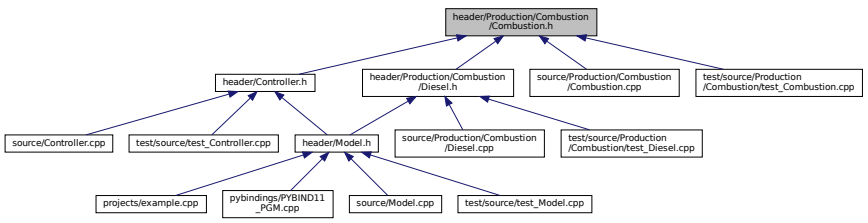
Header file for the Combustion class.

```
#include "../Production.h"
```

Include dependency graph for Combustion.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct CombustionInputs

  A structure which bundles the necessary inputs for the Combustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.
- struct Emissions

  A structure which bundles the emitted masses of various emissions chemistries.
- class Combustion

  The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

Enumerations

- enum CombustionType { DIESEL , N_COMBUSTION_TYPES }

  An enumeration of the types of Combustion asset supported by PGMcpp.
- enum FuelMode { FUEL_MODE_LINEAR , FUEL_MODE_LOOKUP , N_FUEL_MODES }

  An enumeration of the fuel modes for the Combustion asset which are supported by PGMcpp.

5.6.1 Detailed Description

Header file for the Combustion class.

Header file for the Noncombustion class.

5.6.2 Enumeration Type Documentation

5.6.2.1 CombustionType

```
enum CombustionType
```

An enumeration of the types of Combustion asset supported by PGMcpp.

Enumerator

DIESEL	A diesel generator.
N_COMBUSTION_TYPES	A simple hack to get the number of elements in CombustionType.

```
33                  {
34     DIESEL,
35     N_COMBUSTION_TYPES
36 };
```

5.6.2.2 FuelMode

```
enum FuelMode
```

An enumeration of the fuel modes for the Combustion asset which are supported by PGMcpp.

Enumerator

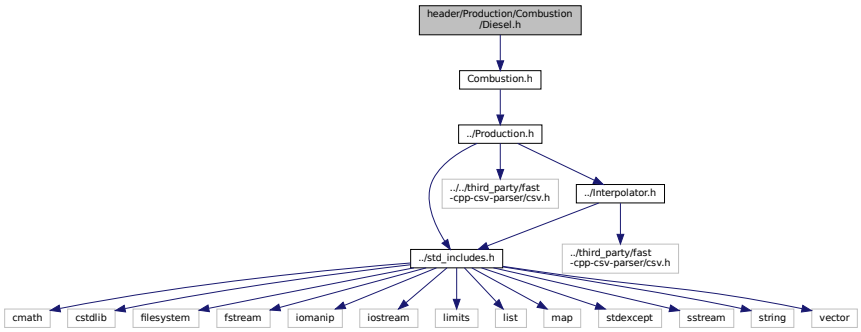
FUEL_MODE_LINEAR	A linearized fuel curve model (i.e., HOMER-like model)
FUEL_MODE_LOOKUP	Interpolating over a given fuel lookup table.
N_FUEL_MODES	A simple hack to get the number of elements in FuelMode.

```
46              {
47     FUEL_MODE_LINEAR,
48     FUEL_MODE_LOOKUP,
49     N_FUEL_MODES
50 };
```

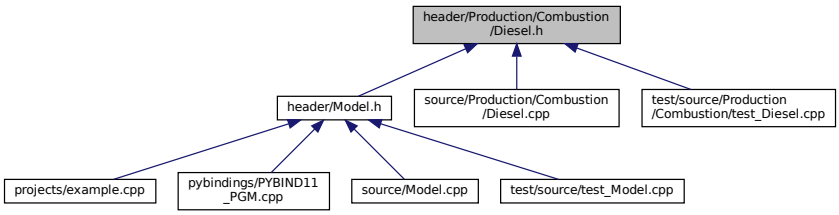
5.7 header/Production/Combustion/Diesel.h File Reference

Header file for the Diesel class.

`#include "Combustion.h"`
Include dependency graph for Diesel.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct DieselInputs

  A structure which bundles the necessary inputs for the Diesel constructor. Provides default values for every necessary input. Note that this structure encapsulates CombustionInputs.
- class Diesel

  A derived class of the Combustion branch of Production which models production using a diesel generator.

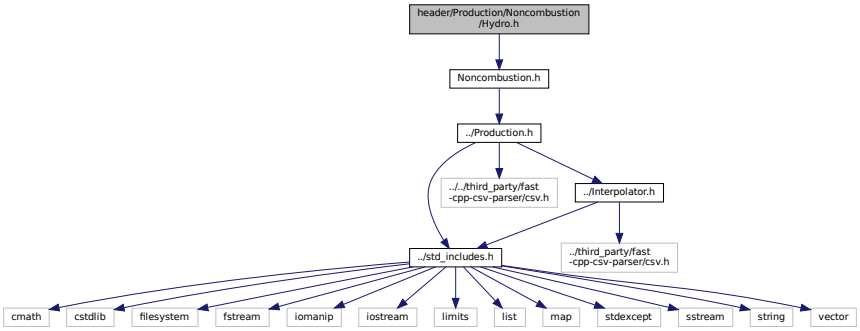
5.7.1 Detailed Description

Header file for the Diesel class.

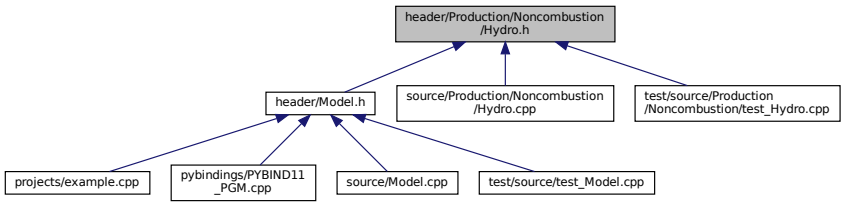
5.8 header/Production/Noncombustion/Hydro.h File Reference

Header file for the Hydro class.

```
#include "Noncombustion.h"
```
Include dependency graph for Hydro.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct HydroInputs

  A structure which bundles the necessary inputs for the Hydro constructor. Provides default values for every necessary input. Note that this structure encapsulates NoncombustionInputs.
- class Hydro

  A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not).

Enumerations

- enum HydroTurbineType { HYDRO_TURBINE_PELTON , HYDRO_TURBINE_FRANCIS , HYDRO_TURBINE_KAPLAN , N_HYDRO_TURBINES }

  An enumeration of the types of hydroelectric turbine supported by PGMcpp.
- enum HydroInterpKeys { GENERATOR_EFFICIENCY_INTERP_KEY , TURBINE_EFFICIENCY_INTERP_KEY , FLOW_TO_POWER_INTERP_KEY , N_HYDRO_INTERP_KEYS }

  An enumeration of the Interpolator keys used by the Hydro asset.

5.8.1 Detailed Description

Header file for the Hydro class.

5.8.2 Enumeration Type Documentation

5.8.2.1 HydroInterpKeys

```
enum HydroInterpKeys
```

An enumeration of the Interpolator keys used by the Hydro asset.

Enumerator

GENERATOR_EFFICIENCY_INTERP_KEY	The key for generator efficiency interpolation.
TURBINE_EFFICIENCY_INTERP_KEY	The key for turbine efficiency interpolation.
FLOW_TO_POWER_INTERP_KEY	The key for flow to power interpolation.
N_HYDRO_INTERP_KEYS	A simple hack to get the number of elements in HydroInterpKeys.

```
47                      {
48     GENERATOR_EFFICIENCY_INTERP_KEY,
49     TURBINE_EFFICIENCY_INTERP_KEY,
50     FLOW_TO_POWER_INTERP_KEY,
51     N_HYDRO_INTERP_KEYS
52 };
```

5.8.2.2 HydroTurbineType

```
enum HydroTurbineType
```

An enumeration of the types of hydroelectric turbine supported by PGMcpp.

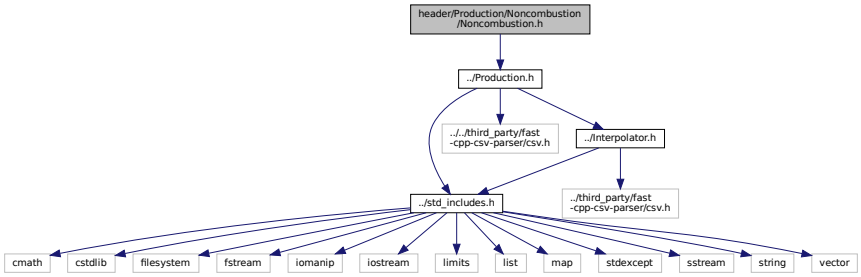
Enumerator

HYDRO_TURBINE_PELTON	A Pelton turbine (impluse)
HYDRO_TURBINE_FRANCIS	A Francis turbine (reaction)
HYDRO_TURBINE_KAPLAN	A Kaplan turbine (reaction)
N_HYDRO_TURBINES	A simple hack to get the number of elements in HydroTurbineType.

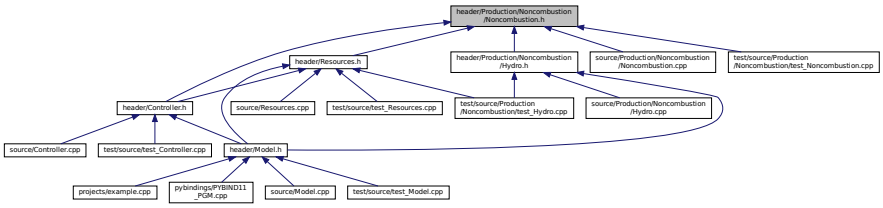
```
33                      {
34     HYDRO_TURBINE_PELTON,
35     HYDRO_TURBINE_FRANCIS,
36     HYDRO_TURBINE_KAPLAN,
37     N_HYDRO_TURBINES
38 };
```

5.9 header/Production/Noncombustion/Noncombustion.h File Reference

`#include "../Production.h"`
Include dependency graph for Noncombustion.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct NoncombustionInputs

  A structure which bundles the necessary inputs for the Noncombustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.
- class Noncombustion

  The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

Enumerations

- enum NoncombustionType { HYDRO , N_NONCOMBUSTION_TYPES }

  An enumeration of the types of Noncombustion asset supported by PGMcpp.

5.9.1 Enumeration Type Documentation

5.9.1.1 NoncombustionType

```
enum NoncombustionType
```

An enumeration of the types of Noncombustion asset supported by PGMcpp.

Enumerator

HYDRO	A hydroelectric generator (either with reservoir or not)
N_NONCOMBUSTION_TYPES	A simple hack to get the number of elements in NoncombustionType.

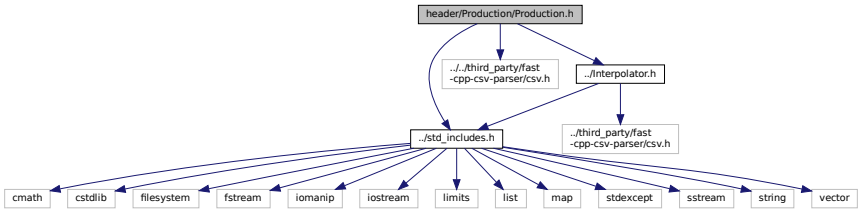
```
33                          {
34     HYDRO,
35     N_NONCOMBUSTION_TYPES
36 };
```

5.10 header/Production/Production.h File Reference

Header file for the Production class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
#include "../Interpolator.h"
```

Include dependency graph for Production.h:



This graph shows which files directly or indirectly include this file:

Classes

- struct ProductionInputs

  A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input.
- class Production

  The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

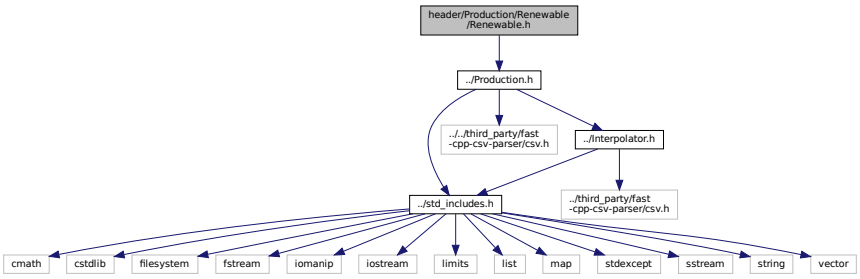
5.10.1 Detailed Description

Header file for the Production class.

5.11 header/Production/Renewable/Renewable.h File Reference

Header file for the Renewable class.

```
#include "../Production.h"
```
Include dependency graph for Renewable.h:



This graph shows which files directly or indirectly include this file:

Classes

- struct RenewableInputs

  A structure which bundles the necessary inputs for the Renewable constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs.
- class Renewable

  The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

Enumerations

- enum RenewableType {
  SOLAR , TIDAL , WAVE , WIND ,
  N_RENEWABLE_TYPES }

  An enumeration of the types of Renewable asset supported by PGMcpp.

5.11.1 Detailed Description

Header file for the Renewable class.

5.11.2 Enumeration Type Documentation

5.11.2.1 RenewableType

```
enum RenewableType
```

An enumeration of the types of Renewable asset supported by PGMcpp.

Enumerator

SOLAR	A solar photovoltaic (PV) array.
TIDAL	A tidal stream turbine (or tidal energy converter, TEC)
WAVE	A wave energy converter (WEC)
WIND	A wind turbine.
N_RENEWABLE_TYPES	A simple hack to get the number of elements in RenewableType.

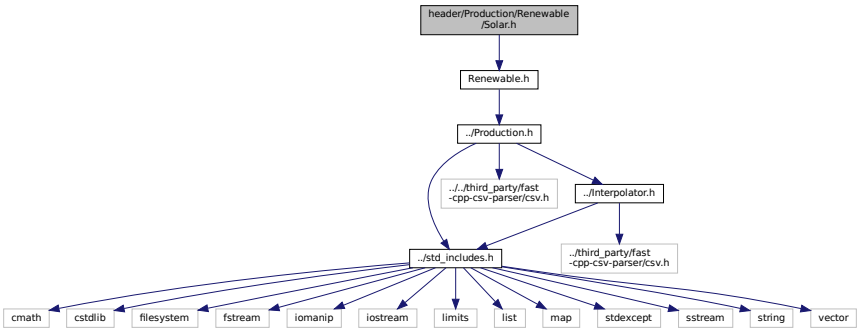
```
33                  {
34     SOLAR,
35     TIDAL,
36     WAVE,
37     WIND,
38     N_RENEWABLE_TYPES
39 };
```

5.12 header/Production/Renewable/Solar.h File Reference

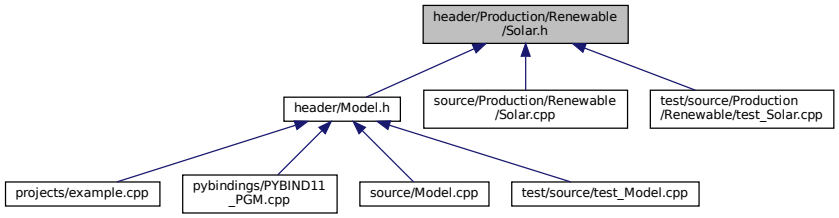
Header file for the Solar class.

```
#include "Renewable.h"
```

Include dependency graph for Solar.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct SolarInputs

  A structure which bundles the necessary inputs for the Solar constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.
- class Solar

  A derived class of the Renewable branch of Production which models solar production.

5.12.1 Detailed Description

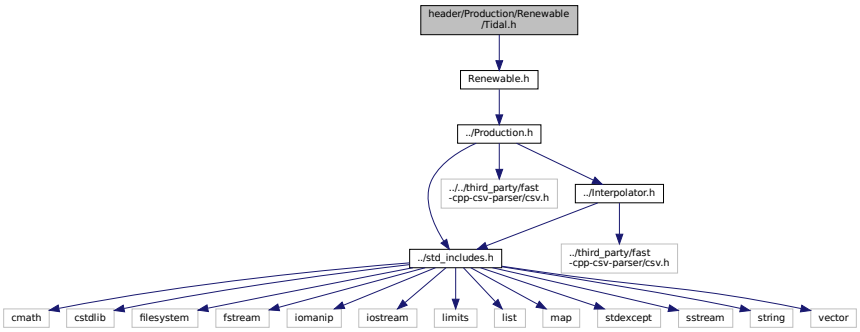
Header file for the Solar class.

5.13 header/Production/Renewable/Tidal.h File Reference

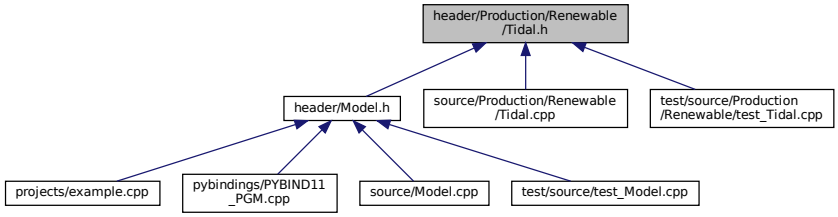
Header file for the Tidal class.

```
#include "Renewable.h"
```

Include dependency graph for Tidal.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct TidalInputs

  A structure which bundles the necessary inputs for the Tidal constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.
- class Tidal

  A derived class of the Renewable branch of Production which models tidal production.

Enumerations

- enum TidalPowerProductionModel { TIDAL_POWER_CUBIC , TIDAL_POWER_EXPONENTIAL , TIDAL_POWER_LOOKUP , N_TIDAL_POWER_PRODUCTION_MODELS }

5.13.1 Detailed Description

Header file for the Tidal class.

5.13.2 Enumeration Type Documentation

5.13.2.1 TidalPowerProductionModel

```
enum TidalPowerProductionModel
```

Enumerator

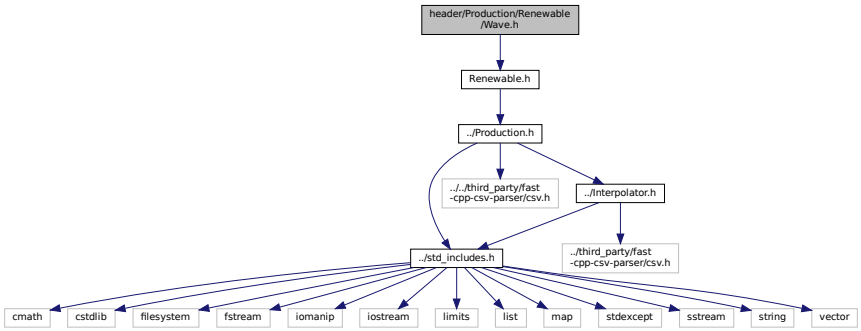
TIDAL_POWER_CUBIC	A cubic power production model.
TIDAL_POWER_EXPONENTIAL	An exponential power production model.
TIDAL_POWER_LOOKUP	Lookup from a given set of power curve data.
N_TIDAL_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in TidalPowerProductionModel.

```
34                                {
35     TIDAL_POWER_CUBIC,
36     TIDAL_POWER_EXPONENTIAL,
37     TIDAL_POWER_LOOKUP,
38     N_TIDAL_POWER_PRODUCTION_MODELS
39 };
```

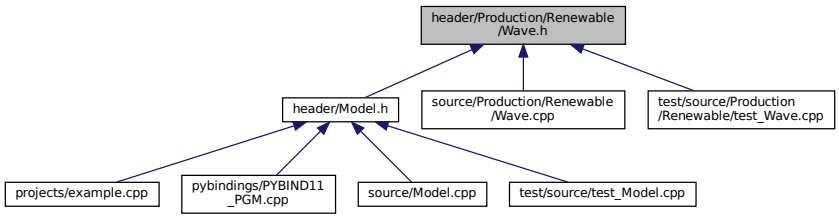
5.14 header/Production/Renewable/Wave.h File Reference

Header file for the Wave class.

`#include "Renewable.h"`
Include dependency graph for Wave.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct WaveInputs

  A structure which bundles the necessary inputs for the Wave constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.
- class Wave

  A derived class of the Renewable branch of Production which models wave production.

Enumerations

- enum WavePowerProductionModel { WAVE_POWER_GAUSSIAN , WAVE_POWER_PARABOLOID , WAVE_POWER_LOOKUP , N_WAVE_POWER_PRODUCTION_MODELS }

5.14.1 Detailed Description

Header file for the Wave class.

5.14.2 Enumeration Type Documentation

5.14.2.1 WavePowerProductionModel

```
enum WavePowerProductionModel
```

Enumerator

WAVE_POWER_GAUSSIAN	A Gaussian power production model.
WAVE_POWER_PARABOLOID	A paraboloid power production model.
WAVE_POWER_LOOKUP	Lookup from a given performance matrix.
N_WAVE_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in WavePowerProductionModel.

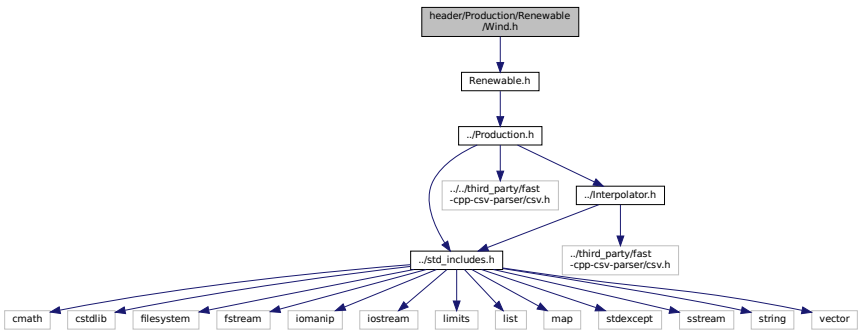
```
34                              {
35     WAVE_POWER_GAUSSIAN,
36     WAVE_POWER_PARABOLOID,
37     WAVE_POWER_LOOKUP,
38     N_WAVE_POWER_PRODUCTION_MODELS
39 };
```

5.15 header/Production/Renewable/Wind.h File Reference

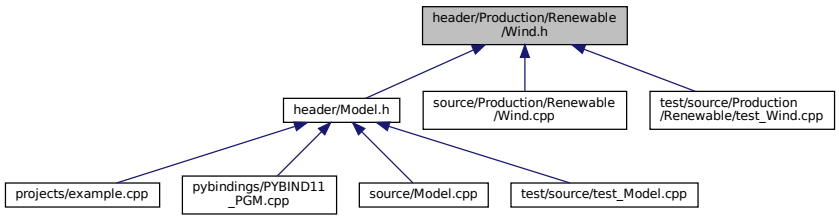
Header file for the Wind class.

```
#include "Renewable.h"
```

Include dependency graph for Wind.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct WindInputs

  A structure which bundles the necessary inputs for the Wind constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.
- class Wind

  A derived class of the Renewable branch of Production which models wind production.

Enumerations

- enum WindPowerProductionModel { WIND_POWER_CUBIC , WIND_POWER_EXPONENTIAL , WIND_POWER_LOOKUP , N_WIND_POWER_PRODUCTION_MODELS }

5.15.1 Detailed Description

Header file for the Wind class.

5.15.2 Enumeration Type Documentation

5.15.2.1 WindPowerProductionModel

```
enum WindPowerProductionModel
```

Enumerator

WIND_POWER_CUBIC	A cubic power production model.
WIND_POWER_EXPONENTIAL	An exponential power production model.
WIND_POWER_LOOKUP	Lookup from a given set of power curve data.
N_WIND_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in WindPowerProductionModel.

```
34                              {
35     WIND_POWER_CUBIC,
```

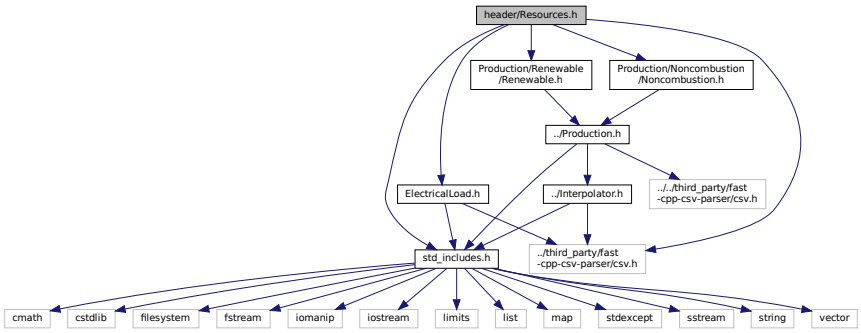
```
36     WIND_POWER_EXPONENTIAL,
37     WIND_POWER_LOOKUP,
38     N_WIND_POWER_PRODUCTION_MODELS
39 };
```

5.16 header/Resources.h File Reference

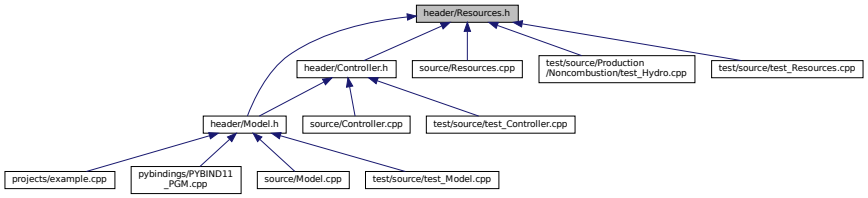
Header file for the Resources class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"
```

Include dependency graph for Resources.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Resources

  A class which contains renewable resource data. Intended to serve as a component class of Model.

5.16.1 Detailed Description

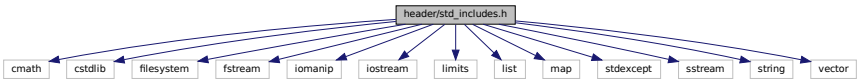
Header file for the Resources class.

5.17 header/std_includes.h File Reference

Header file which simply batches together some standard includes.

```
#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <stdexcept>
#include <sstream>
#include <string>
#include <vector>
```

Include dependency graph for std_includes.h:



This graph shows which files directly or indirectly include this file:

5.17.1 Detailed Description

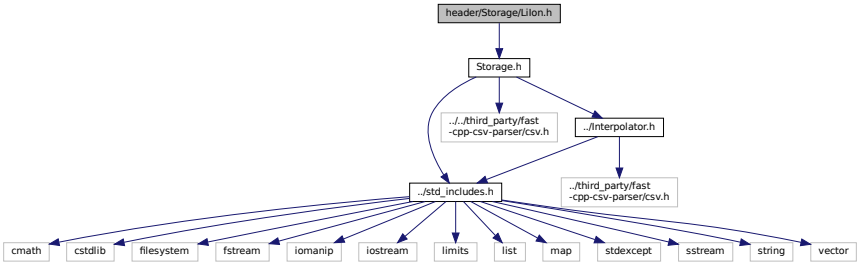
Header file which simply batches together some standard includes.

5.18 header/Storage/Lilon.h File Reference

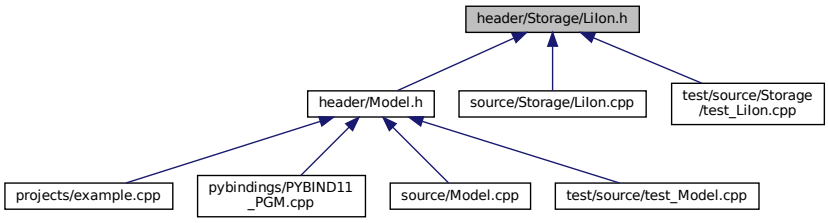
Header file for the Lilon class.

```
#include "Storage.h"
```

Include dependency graph for Lilon.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct LiIonInputs

  A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.
- class LiIon

  A derived class of Storage which models energy storage by way of lithium-ion batteries.

5.18.1 Detailed Description

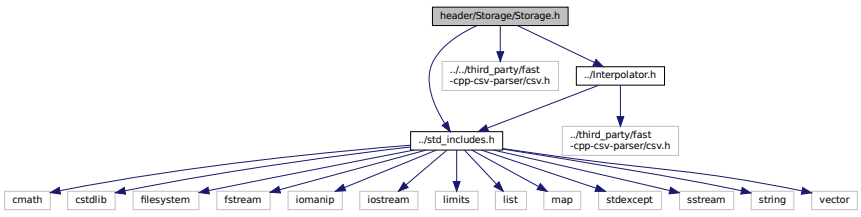
Header file for the LiIon class.

5.19 header/Storage/Storage.h File Reference

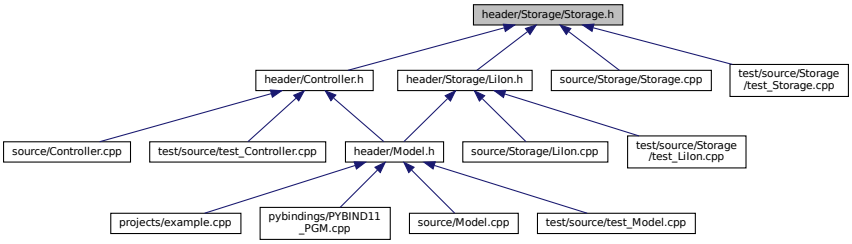
Header file for the Storage class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
#include "../Interpolator.h"
```

Include dependency graph for Storage.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct StorageInputs

  A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input.
- class Storage

  The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

Enumerations

- enum StorageType { LIION , N_STORAGE_TYPES }

  An enumeration of the types of Storage asset supported by PGMcpp.

5.19.1 Detailed Description

Header file for the Storage class.

5.19.2 Enumeration Type Documentation

5.19.2.1 StorageType

```
enum StorageType
```

An enumeration of the types of Storage asset supported by PGMcpp.

Enumerator

LIION	A system of lithium ion batteries.
N_STORAGE_TYPES	A simple hack to get the number of elements in StorageType.

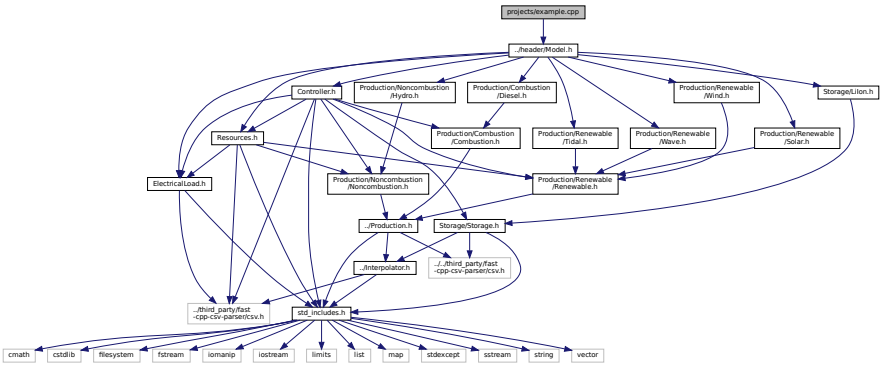
```
36                  {
37     LIION,
```

```
38     N_STORAGE_TYPES
39 };
```

5.20 projects/example.cpp File Reference

```
#include "../header/Model.h"
```

Include dependency graph for example.cpp:



Functions

- int main (int argc, char **argv)

5.20.1 Function Documentation

5.20.1.1 main()

```
int main (
            int argc,
            char ** argv )
26 {
27     /*
28      *  1. construct Model object
29      *
30      *  This block constructs a Model object, which is the central container for the
31      *  entire microgrid model.
32      *
33      *  The fist argument that must be provided to the Model constructor is a valid
34      *  path (either relative or absolute) to a time series of electrical load data.
35      *  For an example of the expected format, see
36      *
37      *  data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv
38      *
39      *  Note that the length of the given electrical load time series defines the
40      *  modelled project life (so if you want to model n years of microgrid operation,
41      *  then you must pass a path to n years worth of electrical load data). In addition,
42      *  the given electrical load time series defines which points in time are modelled.
43      *  As such, all subsequent time series data which is passed in must (1) be of the
44      *  same length as the electrical load time series, and (2) provide data for the
45      *  same set of points in time. Of course, the electrical load time series can be
46      *  of arbitrary length, and it need not be a uniform time series.
47      *
48      *  The second argument that one can provide is the desired disptach control mode.
```

```
49      *  If nothing is given here, then the model will default to simple load following
50      *  control. However, one can stipulate which control mode to use by altering the
51      *  control_mode attribute of the ModelInputs structure. In this case, the
52      *  cycle charging control mode is being set.
53      */
54
55     std::string path_2_electrical_load_time_series =
56         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
57
58     ModelInputs model_inputs;
59
60     model_inputs.path_2_electrical_load_time_series =
61         path_2_electrical_load_time_series;
62
63     model_inputs.control_mode = ControlMode :: CYCLE_CHARGING;
64
65     Model model(model_inputs);
66
67
68
69     /*
70      *  2. add Diesel objects to Model
71      *
72      *  This block defines and adds a set of diesel generators to the Model object.
73      *
74      *  In this example, a single DieselInputs structure is used to define and add
75      *  three diesel generators to the model.
76      *
77      *  The first diesel generator is defined as a 300 kW generator (which shows an
78      *  example of how to access and alter an encapsulated attribute of DieselInputs).
79      *  In addition, the diesel generator is taken to be a sunk cost (and so no capital
80      *  cost is incurred in the first time step; the opposite is true for non-sunk
81      *  assets).
82      *
83      *  The last two diesel generators are defined as 150 kW each. Likewise, they are
84      *  also sunk assets (since the same DieselInputs structure is being re-used without
85      *  overwriting the is_sunk attribute).
86      *
87      *  For more details on the various attributes of DieselInputs, refer to the
88      *  PGMcpp manual. For instance, note that no economic inputs are given; in this
89      *  example, the default values apply.
90      */
91
92     DieselInputs diesel_inputs;
93
94     //  2.1. add 1 x 300 kW diesel generator (since mean load is ~250 kW)
95     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 300;
96     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
97
98     model.addDiesel(diesel_inputs);
99
100     //  2.2. add 2 x 150 kW diesel generators (since max load is 500 kW)
101     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
102
103     model.addDiesel(diesel_inputs);
104     model.addDiesel(diesel_inputs);
105
106
107
108     /*
109      *  3. add renewable resources to Model
110      *
111      *  This block adds a set of renewable resource time series to the Model object.
112      *
113      *  The first resource added is a solar resource time series, which gives
114      *  horizontal irradiance [kW/m2] at each point in time. Again, remember that all
115      *  given time series must align with the electrical load time series (i.e., same
116      *  length, same points). For an example of the expected format, see
117      *
118      *  data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv
119      *
120      *  Finally, note the declaration of a solar resource key. This variable will be
121      *  re-used later to associate a solar PV array object with this particular solar
122      *  resource. This method of key association between resource and asset allows for
123      *  greater flexibility in modelling production assets that are exposed to different
124      *  renewable resources (due to being geographically separated, etc.).
125      *
126      *  The second resource added is a tidal resource time series, which gives tidal
127      *  stream speed [m/s] at each point in time. For an example of the expected format,
128      *  see
129      *
130      *  data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv
131      *
132      *  Again, note the tidal resource key.
133      *
134      *  The third resource added is a wave resource time series, which gives significant
135      *  wave height [m] and energy period [s] at each point in time. For an example of
```

```
136      *  the expected format, see
137      *
138      *  data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv
139      *
140      *  Again, note the wave resource key.
141      *
142      *  The fourth resource added is a wind resource time series, which gives wind speed
143      *  [m/s] at each point in time. For an example of the expected format, see
144      *
145      *  data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv
146      *
147      *  Again, note the wind resource key.
148      *
149      *  The fifth resource added is a hydro resource time series, which gives inflow
150      *  rate [m3/hr] at each point in time. For an example of the expected format, see
151      *
152      *  data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv
153      *
154      *  Again, note the hydro resource key.
155      */
156
157     //  3.1. add solar resource time series
158     int solar_resource_key = 0;
159     std::string path_2_solar_resource_data =
160         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
161
162     model.addResource(
163         RenewableType :: SOLAR,
164         path_2_solar_resource_data,
165         solar_resource_key
166     );
167
168     //  3.2. add tidal resource time series
169     int tidal_resource_key = 1;
170     std::string path_2_tidal_resource_data =
171         "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
172
173     model.addResource(
174         RenewableType :: TIDAL,
175         path_2_tidal_resource_data,
176         tidal_resource_key
177     );
178
179     //  3.3. add wave resource time series
180     int wave_resource_key = 2;
181     std::string path_2_wave_resource_data =
182         "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
183
184     model.addResource(
185         RenewableType :: WAVE,
186         path_2_wave_resource_data,
187         wave_resource_key
188     );
189
190     //  3.4. add wind resource time series
191     int wind_resource_key = 3;
192     std::string path_2_wind_resource_data =
193         "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
194
195     model.addResource(
196         RenewableType :: WIND,
197         path_2_wind_resource_data,
198         wind_resource_key
199     );
200
201     //  3.5. add hydro resource time series
202     int hydro_resource_key = 4;
203     std::string path_2_hydro_resource_data =
204         "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
205
206     model.addResource(
207         NoncombustionType :: HYDRO,
208         path_2_hydro_resource_data,
209         hydro_resource_key
210     );
211
212
213
214     /*
215      *  4. add Hydro object to Model
216      *
217      *  This block defines and adds a hydroelectric asset to the Model object.
218      *
219      *  In this example, a 300 kW hydroelectric station with a 10,000 m3 reservoir
220      *  is defined. The initial reservoir state is set to 50% (so half full), and the
221      *  hydroelectric asset is taken to be a sunk asset (so no capital cost incurred
222      *  in the first time step). Note the association with the previously given hydro
```

```
223      *  resource series by way of the hydro resource key.
224      *
225      *  For more details on the various attributes of HydroInputs, refer to the
226      *  PGMcpp manual. For instance, note that no economic inputs are given; in this
227      *  example, the default values apply.
228      */
229
230     HydroInputs hydro_inputs;
231     hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
232     hydro_inputs.reservoir_capacity_m3 = 10000;
233     hydro_inputs.init_reservoir_state = 0.5;
234     hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
235     hydro_inputs.resource_key = hydro_resource_key;
236
237     model.addHydro(hydro_inputs);
238
239
240
241     /*
242      *  5. add Renewable objects to Model
243      *
244      *  This block defines and adds a set of renewable production assets to the Model
245      *  object.
246      *
247      *  The first block defines and adds a solar PV array to the Model object. In this
248      *  example, the installed solar capacity is set to 250 kW. Note the association
249      *  with the previously given solar resource series by way of the solar resource
250      *  key. Also, note that this asset is not taken as sunk (as the is_sunk attribute
251      *  of the SolarInputs structure is unchanged and thus defaults to true). As such,
252      *  this asset will incur a capital cost in the first time step.
253      *
254      *  For more details on the various attributes of SolarInputs, refer to the PGMcpp
255      *  manual. For instance, note that no economic inputs are given; in this
256      *  example, the default values apply.
257      *
258      *  The second block defines and adds a tidal turbine to the Model object. In this
259      *  example, the installed tidal capacity is set to 120 kW. In addition, the design
260      *  speed of the asset (i.e., the speed at which the rated capacity is achieved) is
261      *  set to 2.5 m/s. Note the association with the previously given tidal resource
262      *  series by way of the tidal resource key.
263      *
264      *  For more details on the various attributes of TidalInputs, refer to the PGMcpp
265      *  manual. For instance, note that no economic inputs are given; in this
266      *  example, the default values apply.
267      *
268      *  The third block defines and adds a wind turbine to the Model object. In this
269      *  example, the installed wind capacity is set to 150 kW. In addition, the design
270      *  speed of the asset is not given, and so will default to 8 m/s. Note the
271      *  association with the previously given tidal resource series by way of the wind
272      *  resource key.
273      *
274      *  For more details on the various attributes of WindInputs, refer to the PGMcpp
275      *  manual. For instance, note that no economic inputs are given; in this
276      *  example, the default values apply.
277      *
278      *  The fourth block defines and adds a wave energy converter to the Model object.
279      *  In this example, the installed wave capacity is set to 100 kW. Note the
280      *  association with the previously given wave resource series by way of the wave
281      *  resource key.
282      *
283      *  For more details on the various attributes of WaveInputs, refer to the PGMcpp
284      *  manual. For instance, note that no economic inputs are given; in this
285      *  example, the default values apply.
286      */
287
288     //  5.1. add 1 x 250 kW solar PV array
289     SolarInputs solar_inputs;
290
291     solar_inputs.renewable_inputs.production_inputs.capacity_kW = 250;
292     solar_inputs.resource_key = solar_resource_key;
293
294     model.addSolar(solar_inputs);
295
296     //  5.2. add 1 x 120 kW tidal turbine
297     TidalInputs tidal_inputs;
298
299     tidal_inputs.renewable_inputs.production_inputs.capacity_kW = 120;
300     tidal_inputs.design_speed_ms = 2.5;
301     tidal_inputs.resource_key = tidal_resource_key;
302
303     model.addTidal(tidal_inputs);
304
305     //  5.3. add 1 x 150 kW wind turbine
306     WindInputs wind_inputs;
307
308     wind_inputs.renewable_inputs.production_inputs.capacity_kW = 150;
309     wind_inputs.resource_key = wind_resource_key;
```

```
310
311     model.addWind(wind_inputs);
312
313     //  5.4. add 1 x 100 kW wave energy converter
314     WaveInputs wave_inputs;
315
316     wave_inputs.renewable_inputs.production_inputs.capacity_kW = 100;
317     wave_inputs.resource_key = wave_resource_key;
318
319     model.addWave(wave_inputs);
320
321
322
323     /*
324      *  6. add LiIon object to Model
325      *
326      *  This block defines and adds a lithium ion battery energy storage system to the
327      *  Model object.
328      *
329      *  In this example, a battery energy storage system with a 500 kW power capacity
330      *  and a 1050 kWh energy capacity (which represents about four hours of mean load
331      *  autonomy) is defined.
332      *
333      *  For more details on the various attributes of LiIonInputs, refer to the PGMcpp
334      *  manual. For instance, note that no economic inputs are given; in this
335      *  example, the default values apply.
336      */
337
338     //  6.1. add 1 x (500 kW, ) lithium ion battery energy storage system
339     LiIonInputs liion_inputs;
340
341     liion_inputs.storage_inputs.power_capacity_kW = 500;
342     liion_inputs.storage_inputs.energy_capacity_kWh = 1050;
343
344     model.addLiIon(liion_inputs);
345
346
347
348     /*
349      *  7. run and write results
350      *
351      *  This block runs the model and then writes results to the given output path
352      *  (either relative or absolute). Note that the writeResults() will create the
353      *  last directory on the given path, but not any in-between directories, so be
354      *  sure those exist before calling out to this method.
355      */
356
357     model.run();
358
359     model.writeResults("projects/example_cpp");
360
361     return 0;
362 }   /* main() */
```

5.21 pybindings/PYBIND11_PGM.cpp File Reference

Bindings file for PGMcpp.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include "../header/Model.h"
#include "snippets/PYBIND11_Controller.cpp"
#include "snippets/PYBIND11_ElectricalLoad.cpp"
#include "snippets/PYBIND11_Interpolator.cpp"
#include "snippets/PYBIND11_Model.cpp"
#include "snippets/PYBIND11_Resources.cpp"
#include "snippets/Production/PYBIND11_Production.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"
#include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
#include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
#include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
#include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
```

```
#include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
#include "snippets/Storage/PYBIND11_Storage.cpp"
#include "snippets/Storage/PYBIND11_LiIon.cpp"
```

Include dependency graph for PYBIND11_PGM.cpp:

Functions

- PYBIND11_MODULE (PGMcpp, m)

5.21.1 Detailed Description

Bindings file for PGMcpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for PGMcpp. Only public attributes/methods are bound!

5.21.2 Function Documentation

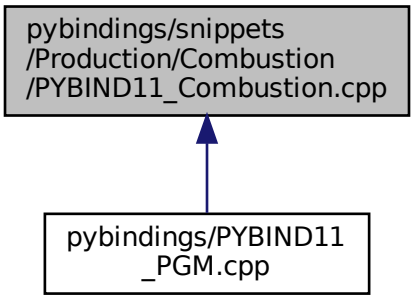
5.21.2.1 PYBIND11_MODULE()

```
PYBIND11_MODULE (
             PGMcpp ,
             m   )
31                           {
32
33     #include "snippets/PYBIND11_Controller.cpp"
34     #include "snippets/PYBIND11_ElectricalLoad.cpp"
35     #include "snippets/PYBIND11_Interpolator.cpp"
36     #include "snippets/PYBIND11_Model.cpp"
37     #include "snippets/PYBIND11_Resources.cpp"
38
39     #include "snippets/Production/PYBIND11_Production.cpp"
40
41     #include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
42     #include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"
43
44     #include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
45     #include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
46
47     #include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
48     #include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
49     #include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
50     #include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
51     #include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
52
53     #include "snippets/Storage/PYBIND11_Storage.cpp"
54     #include "snippets/Storage/PYBIND11_LiIon.cpp"
55
56 }   /* PYBIND11_MODULE() */
```

5.22 pybindings/snippets/Production/Combustion/PYBIND11_↩ Combustion.cpp File Reference

Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- CombustionType::DIESEL value ("N_COMBUSTION_TYPES", CombustionType::N_COMBUSTION_↩ TYPES)
- FuelMode::FUEL_MODE_LINEAR value ("FUEL_MODE_LOOKUP", FuelMode::FUEL_MODE_LOOKUP) .value("N_FUEL_MODES"
- &CombustionInputs::production_inputs def_readwrite ("fuel_mode", &CombustionInputs::fuel_mode) .def_↩ readwrite("nominal_fuel_escalation_annual"

Variables

- &CombustionInputs::production_inputs &CombustionInputs::nominal_fuel_escalation_annual def_↩ readwrite("path_2_fuel_interp_data", &CombustionInputs::path_2_fuel_interp_data) .def(pybind11 &Emissions::CO2_kg def_readwrite ("CO_kg", &Emissions::CO_kg) .def_readwrite("NOx_kg"

5.22.1 Detailed Description

Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Combustion class. Only public attributes/methods are bound!

5.22.2 Function Documentation

5.22.2.1 def_readwrite()

```
& CombustionInputs::production_inputs def_readwrite (
             "fuel_mode" ,
             &CombustionInputs::fuel_mode  )
```

5.22.2.2 value() [1/2]

```
FuelMode::FUEL_MODE_LINEAR value (
             "FUEL_MODE_LOOKUP" ,
             FuelMode::FUEL_MODE_LOOKUP  )
```

5.22.2.3 value() [2/2]

```
CombustionType::DIESEL value (
             "N_COMBUSTION_TYPES" ,
             CombustionType::N_COMBUSTION_TYPES  )
```

5.22.3 Variable Documentation

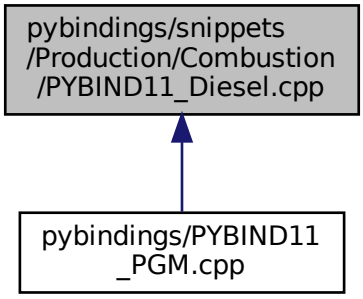
5.22.3.1 def_readwrite

```
&StorageInputs::print_flag &StorageInputs::power_capacity_kW &StorageInputs::nominal_inflation_annual
def_readwrite (
             "CO_kg" ,
             &Emissions::CO_kg  )
```

5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference

Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &DieselInputs::combustion_inputs def_readwrite ("replace_running_hrs", &DieselInputs::replace_running_↩ hrs) .def_readwrite("capital_cost"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost def_readwrite ("operation_maintenance_↩ cost_kWh", &DieselInputs::operation_maintenance_cost_kWh) .def_readwrite("fuel_cost_L"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L def_readwrite ("minimum_load_ratio", &DieselInputs::minimum_load_ratio) .def_readwrite("minimum_runtime_hrs"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hr def_readwrite ("linear_fuel_slope_LkWh", &DieselInputs::linear_fuel_slope_LkWh) .def_readwrite("linear_↩ fuel_intercept_LkWh"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hr &DieselInputs::linear_fuel_intercept_LkWh def_readwrite ("CO2_emissions_intensity_kgL", &DieselInputs↩ ::CO2_emissions_intensity_kgL) .def_readwrite("CO_emissions_intensity_kgL"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hr &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO_emissions_intensity_kgL def_readwrite ("NOx_emissions_intensity_kgL", &DieselInputs::NOx_emissions_intensity_kgL) .def_readwrite("SOx_↩ emissions_intensity_kgL"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hr &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO_emissions_intensity_kgL &DieselInputs::SOx_emissions_intens def_readwrite ("CH4_emissions_intensity_kgL", &DieselInputs::CH4_emissions_intensity_kgL) .def_↩ readwrite("PM_emissions_intensity_kgL"
- &DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hr &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO_emissions_intensity_kgL &DieselInputs::SOx_emissions_intens &DieselInputs::PM_emissions_intensity_kgL def (pybind11::init())
- &Diesel::minimum_load_ratio def_readwrite ("minimum_runtime_hrs", &Diesel::minimum_runtime_hrs) .def_readwrite("time_since_last_start_hrs"

5.23.1 Detailed Description

Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Diesel class. Only public attributes/methods are bound!

5.23.2 Function Documentation

5.23.2.1 def()

```
&InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D:
&InterpolatorStruct2D::z_matrix def (
            pybind11::init()  )
```

5.23.2.2 def_readwrite() [1/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
& DieselInputs::SOx_emissions_intensity_kgL def_readwrite (
            "CH4_emissions_intensity_kgL" ,
            &DieselInputs::CH4_emissions_intensity_kgL  )
```

5.23.2.3 def_readwrite() [2/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh def_readwrite (
            "CO2_emissions_intensity_kgL" ,
            &DieselInputs::CO2_emissions_intensity_kgL  )
```

5.23.2.4 def_readwrite() [3/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs def_readwrite (
            "linear_fuel_slope_LkWh" ,
            &DieselInputs::linear_fuel_slope_LkWh  )
```

5.23.2.5 def_readwrite() [4/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L
def_readwrite (
            "minimum_load_ratio" ,
            &DieselInputs::minimum_load_ratio  )
```

5.23.2.6 def_readwrite() [5/8]

```
& Diesel::minimum_load_ratio def_readwrite (
              "minimum_runtime_hrs" ,
              &Diesel::minimum_runtime_hrs  )
```

5.23.2.7 def_readwrite() [6/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
def_readwrite (
              "NOx_emissions_intensity_kgL" ,
              &DieselInputs::NOx_emissions_intensity_kgL  )
```

5.23.2.8 def_readwrite() [7/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost def_readwrite (
              "operation_maintenance_cost_kWh" ,
              &DieselInputs::operation_maintenance_cost_kWh  )
```

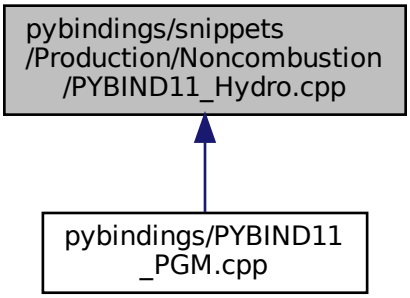
5.23.2.9 def_readwrite() [8/8]

```
& DieselInputs::combustion_inputs def_readwrite (
              "replace_running_hrs" ,
              &DieselInputs::replace_running_hrs  )
```

5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_↩ Hydro.cpp File Reference

Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- HydroTurbineType::HYDRO_TURBINE_PELTON value ("HYDRO_TURBINE_FRANCIS", HydroTurbine↩ Type::HYDRO_TURBINE_FRANCIS) .value("HYDRO_TURBINE_KAPLAN"
- HydroTurbineType::HYDRO_TURBINE_PELTON HydroTurbineType::HYDRO_TURBINE_KAPLAN value ("N_HYDRO_TURBINES", HydroTurbineType::N_HYDRO_TURBINES)
- &HydroInputs::noncombustion_inputs def_readwrite ("resource_key", &HydroInputs::resource_key) .def_↩ readwrite("capital_cost"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost def_readwrite ("operation_maintenance↩ _cost_kWh", &HydroInputs::operation_maintenance_cost_kWh) .def_readwrite("fluid_density_kgm3"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 def_readwrite ("net_head_m", &HydroInputs::net_head_m) .def_readwrite("reservoir_capacity_m3"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 &HydroInputs::reservoir_capacity_m3 def_readwrite ("init_reservoir_state", &HydroInputs::init_reservoir↩ _state) .def_readwrite("turbine_type"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 &HydroInputs::reservoir_capacity_m3 &HydroInputs::turbine_type def (pybind11::init())
- &Hydro::turbine_type def_readwrite ("fluid_density_kgm3", &Hydro::fluid_density_kgm3) .def_readwrite("net↩ _head_m"
- &Hydro::turbine_type &Hydro::net_head_m def_readwrite ("reservoir_capacity_m3", &Hydro::reservoir_↩ capacity_m3) .def_readwrite("init_reservoir_state"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state def_readwrite ("stored_volume_↩ m3", &Hydro::stored_volume_m3) .def_readwrite("minimum_power_kW"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state &Hydro::minimum_power_kW def_readwrite ("minimum_flow_m3hr", &Hydro::minimum_flow_m3hr) .def_readwrite("maximum_flow_m3hr"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state &Hydro::minimum_power_kW &Hydro::maximum_flow_m3hr def_readwrite ("turbine_flow_vec_m3hr", &Hydro::turbine_flow_vec_m3hr) .def_readwrite("spill_rate_vec_m3hr"

5.24.1 Detailed Description

Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Hydro class. Only public attributes/methods are bound!

5.24.2 Function Documentation

5.24.2.1 def()

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 & HydroInputs::turbine_type def (
            pybind11::init()  )
```

5.24.2.2 def_readwrite() [1/9]

```
& Hydro::turbine_type def_readwrite (
              "fluid_density_kgm3" ,
              &Hydro::fluid_density_kgm3  )
```

5.24.2.3 def_readwrite() [2/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 def_readwrite (
              "init_reservoir_state" ,
              &HydroInputs::init_reservoir_state  )
```

5.24.2.4 def_readwrite() [3/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
def_readwrite (
              "minimum_flow_m3hr" ,
              &Hydro::minimum_flow_m3hr  )
```

5.24.2.5 def_readwrite() [4/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
def_readwrite (
              "net_head_m" ,
              &HydroInputs::net_head_m  )
```

5.24.2.6 def_readwrite() [5/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost def_readwrite (
              "operation_maintenance_cost_kWh" ,
              &HydroInputs::operation_maintenance_cost_kWh  )
```

5.24.2.7 def_readwrite() [6/9]

```
& Hydro::turbine_type & Hydro::net_head_m def_readwrite (
              "reservoir_capacity_m3" ,
              &Hydro::reservoir_capacity_m3  )
```

5.24.2.8 def_readwrite() [7/9]

```
& HydroInputs::noncombustion_inputs def_readwrite (
             "resource_key" ,
             &HydroInputs::resource_key  )
```

5.24.2.9 def_readwrite() [8/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state def_readwrite (
             "stored_volume_m3" ,
             &Hydro::stored_volume_m3  )
```

5.24.2.10 def_readwrite() [9/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
& Hydro::maximum_flow_m3hr def_readwrite (
             "turbine_flow_vec_m3hr" ,
             &Hydro::turbine_flow_vec_m3hr  )
```

5.24.2.11 value() [1/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON value (
             "HYDRO_TURBINE_FRANCIS" ,
             HydroTurbineType::HYDRO_TURBINE_FRANCIS  )
```

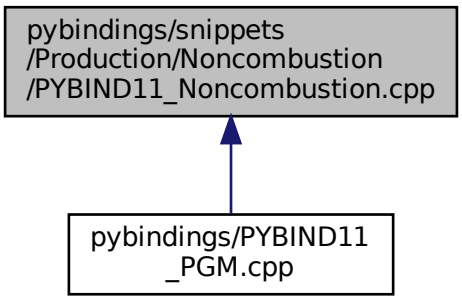
5.24.2.12 value() [2/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON HydroTurbineType::HYDRO_TURBINE_KAPLAN value (
             "N_HYDRO_TURBINES" ,
             HydroTurbineType::N_HYDRO_TURBINES  )
```

5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_↩ Noncombustion.cpp File Reference

Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- NoncombustionType::HYDRO value ("N_NONCOMBUSTION_TYPES", NoncombustionType::N_↩ NONCOMBUSTION_TYPES)
- &NoncombustionInputs::production_inputs def (pybind11::init())

5.25.1 Detailed Description

Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Noncombustion class. Only public attributes/methods are bound!

5.25.2 Function Documentation

5.25.2.1 def()

```
& NoncombustionInputs::production_inputs def (
            pybind11::init()  )
```

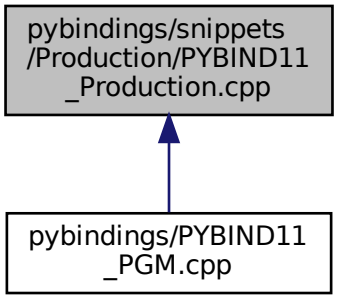
5.25.2.2 value()

```
NoncombustionType::HYDRO value (
            "N_NONCOMBUSTION_TYPES" ,
            NoncombustionType::N_NONCOMBUSTION_TYPES  )
```

5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference

Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &ProductionInputs::print_flag def_readwrite ("is_sunk", &ProductionInputs::is_sunk) .def_readwrite("capacity↩ _kW"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW def_readwrite ("nominal_inflation_annual", &ProductionInputs::nominal_inflation_annual) .def_readwrite("nominal_discount_annual"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual def_readwrite ("replace_running_hrs", &ProductionInputs::replace_running_hrs) .def_readwrite("path_2_↩ normalized_production_time_series"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual &ProductionInputs::path_2_normalized_production_time_series def (pybind11::init())
- &Production::interpolator def_readwrite ("print_flag", &Production::print_flag) .def_readwrite("is_running"
- &Production::interpolator &Production::is_running def_readwrite ("is_sunk", &Production::is_sunk) .def_↩ readwrite("normalized_production_series_given"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given def_readwrite ("n_points", &Production::n_points) .def_readwrite("n_starts"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts def_readwrite ("n_replacements", &Production::n_replacements) .def_readwrite("n_↩ years"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years def_readwrite ("running_hours", &Production::running_hours) .def_readwrite("replace_running_hrs"

- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs def_readwrite ("capacity_↩ kW", &Production::capacity_kW) .def_readwrite("nominal_inflation_annual"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual def_readwrite ("nominal_discount_annual", &Production::nominal_discount_annual) .def_readwrite("real_↩ discount_annual"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual def_readwrite ("capital_cost", &Production::capital_cost) .def_↩ readwrite("operation_maintenance_cost_kWh"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh def_readwrite ("net_↩ present_cost", &Production::net_present_cost) .def_readwrite("total_dispatch_kWh"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh def_readwrite ("levellized_cost_of_energy_kWh", &Production::levellized_cost_of_energy_kWh) .def_↩ readwrite("type_str"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str def_readwrite ("path_2_normalized_production_time_series", &Production::path_2_↩ normalized_production_time_series) .def_readwrite("is_running_vec"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec def_readwrite ("normalized_production_vec", &Production↩ ::normalized_production_vec) .def_readwrite("production_vec_kW"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW def_readwrite ("dispatch_vec_kW", &Production::dispatch_vec_kW) .def_readwrite("storage_vec_kW"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW &Production::storage_vec_kW def_readwrite ("curtailment_vec_kW", &Production::curtailment_vec_kW) .def_readwrite("capital_cost_vec"

5.26.1 Detailed Description

Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Production class. Only public attributes/methods are bound!

5.26.2 Function Documentation

5.26.2.1 def()

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
& ProductionInputs::path_2_normalized_production_time_series def (
            pybind11::init()  )
```

5.26.2.2 def_readwrite() [1/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs def_readwrite (
            "capacity_kW" ,
            &Production::capacity_kW  )
```

5.26.2.3 def_readwrite() [2/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual def_readwrite (
            "capital_cost" ,
            &Production::capital_cost  )
```

5.26.2.4 def_readwrite() [3/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW & Production::storage_vec_
def_readwrite (
            "curtailment_vec_kW" ,
            &Production::curtailment_vec_kW  )
```

5.26.2.5 def_readwrite() [4/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW def_↩
readwrite (
            "dispatch_vec_kW" ,
            &Production::dispatch_vec_kW  )
```

5.26.2.6 def_readwrite() [5/17]

```
& Production::interpolator & Production::is_running def_readwrite (
             "is_sunk" ,
             &Production::is_sunk  )
```

5.26.2.7 def_readwrite() [6/17]

```
& ProductionInputs::print_flag def_readwrite (
             "is_sunk" ,
             &ProductionInputs::is_sunk  )
```

5.26.2.8 def_readwrite() [7/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
def_readwrite (
             "levellized_cost_of_energy_kWh" ,
             &Production::levellized_cost_of_energy_kWh  )
```

5.26.2.9 def_readwrite() [8/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
def_readwrite (
             "n_points" ,
             &Production::n_points  )
```

5.26.2.10 def_readwrite() [9/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts def_readwrite (
             "n_replacements" ,
             &Production::n_replacements  )
```

5.26.2.11 def_readwrite() [10/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh def_readwrite
(
            "net_present_cost" ,
            &Production::net_present_cost  )
```

5.26.2.12 def_readwrite() [11/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
def_readwrite (
            "nominal_discount_annual" ,
            &Production::nominal_discount_annual  )
```

5.26.2.13 def_readwrite() [12/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW def_readwrite (
            "nominal_inflation_annual" ,
            &ProductionInputs::nominal_inflation_annual  )
```

5.26.2.14 def_readwrite() [13/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec def_readwrite (
            "normalized_production_vec" ,
            &Production::normalized_production_vec  )
```

5.26.2.15 def_readwrite() [14/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str def_readwrite (
            "path_2_normalized_production_time_series" ,
            &Production::path_2_normalized_production_time_series  )
```

5.26.2.16 def_readwrite() [15/17]

```
& Production::interpolator def_readwrite (
              "print_flag" ,
              &Production::print_flag  )
```

5.26.2.17 def_readwrite() [16/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
def_readwrite (
              "replace_running_hrs" ,
              &ProductionInputs::replace_running_hrs  )
```

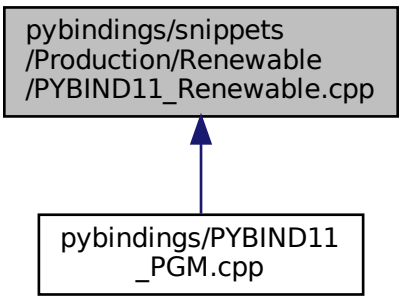
5.26.2.18 def_readwrite() [17/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years def_readwrite (
              "running_hours" ,
              &Production::running_hours  )
```

5.27 pybindings/snippets/Production/Renewable/PYBIND11_↩ Renewable.cpp File Reference

Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- RenewableType::SOLAR value ("TIDAL", RenewableType::TIDAL) .value("WAVE"
- RenewableType::SOLAR RenewableType::WAVE value ("WIND", RenewableType::WIND) .value("N_↩ RENEWABLE_TYPES"
- &RenewableInputs::production_inputs def (pybind11::init())

5.27.1 Detailed Description

Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Renewable class. Only public attributes/methods are bound!

5.27.2 Function Documentation

5.27.2.1 def()

```
& RenewableInputs::production_inputs def (
            pybind11::init()  )
```

5.27.2.2 value() [1/2]

```
RenewableType::SOLAR value (
            "TIDAL" ,
            RenewableType::TIDAL  )
```

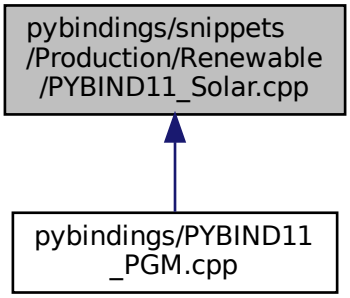
5.27.2.3 value() [2/2]

```
RenewableType::SOLAR RenewableType::WAVE value (
            "WIND" ,
            RenewableType::WIND  )
```

5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference

Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &SolarInputs::renewable_inputs def_readwrite ("resource_key", &SolarInputs::resource_key) .def_↩ readwrite("capital_cost"
- &SolarInputs::renewable_inputs &SolarInputs::capital_cost def_readwrite ("operation_maintenance_cost_↩ kWh", &SolarInputs::operation_maintenance_cost_kWh) .def_readwrite("derating"
- &SolarInputs::renewable_inputs &SolarInputs::capital_cost &SolarInputs::derating def (pybind11::init())

5.28.1 Detailed Description

Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Solar class. Only public attributes/methods are bound!

5.28.2 Function Documentation

5.28.2.1 def()

```
& SolarInputs::renewable_inputs & SolarInputs::capital_cost & SolarInputs::derating def (
            pybind11::init()  )
```

5.28.2.2 def_readwrite() [1/2]

```
& SolarInputs::renewable_inputs & SolarInputs::capital_cost def_readwrite (
              "operation_maintenance_cost_kWh" ,
              &SolarInputs::operation_maintenance_cost_kWh  )
```

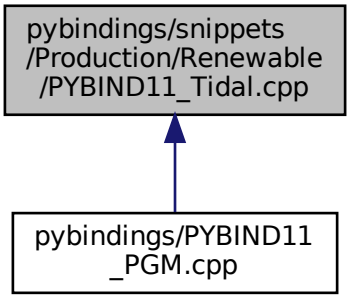
5.28.2.3 def_readwrite() [2/2]

```
& SolarInputs::renewable_inputs def_readwrite (
              "resource_key" ,
              &SolarInputs::resource_key  )
```

5.29 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference

Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- TidalPowerProductionModel::TIDAL_POWER_CUBIC value ("TIDAL_POWER_EXPONENTIAL", Tidal↩ PowerProductionModel::TIDAL_POWER_EXPONENTIAL) .value("TIDAL_POWER_LOOKUP"
- TidalPowerProductionModel::TIDAL_POWER_CUBIC TidalPowerProductionModel::TIDAL_POWER_LOOKUP value ("N_TIDAL_POWER_PRODUCTION_MODELS", TidalPowerProductionModel::N_TIDAL_POWER_↩ PRODUCTION_MODELS)
- &TidalInputs::renewable_inputs def_readwrite ("resource_key", &TidalInputs::resource_key) .def_↩ readwrite("capital_cost"
- &TidalInputs::renewable_inputs &TidalInputs::capital_cost def_readwrite ("operation_maintenance_cost_k↩ Wh", &TidalInputs::operation_maintenance_cost_kWh) .def_readwrite("design_speed_ms"

Variables

- &TidalInputs::renewable_inputs &TidalInputs::capital_cost &TidalInputs::design_speed_ms def_readwrite("power_model", &TidalInputs::power_model) .def(pybind11 &Tidal::design_speed_ms def_readwrite ("power_model", &Tidal::power_model) .def_readwrite("power_model_string"

5.29.1 Detailed Description

Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Tidal class. Only public attributes/methods are bound!

5.29.2 Function Documentation

5.29.2.1 def_readwrite() [1/2]

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost def_readwrite (
            "operation_maintenance_cost_kWh" ,
            &TidalInputs::operation_maintenance_cost_kWh  )
```

5.29.2.2 def_readwrite() [2/2]

```
& TidalInputs::renewable_inputs def_readwrite (
            "resource_key" ,
            &TidalInputs::resource_key  )
```

5.29.2.3 value() [1/2]

```
TidalPowerProductionModel::TIDAL_POWER_CUBIC TidalPowerProductionModel::TIDAL_POWER_LOOKUP
value (
            "N_TIDAL_POWER_PRODUCTION_MODELS" ,
            TidalPowerProductionModel::N_TIDAL_POWER_PRODUCTION_MODELS  )
```

5.29.2.4 value() [2/2]

```
TidalPowerProductionModel::TIDAL_POWER_CUBIC value (
            "TIDAL_POWER_EXPONENTIAL" ,
            TidalPowerProductionModel::TIDAL_POWER_EXPONENTIAL  )
```

5.29.3 Variable Documentation

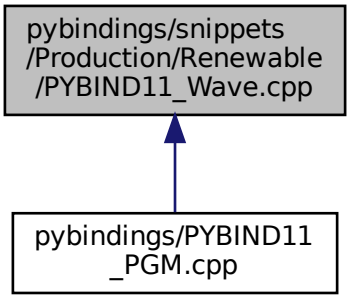
5.29.3.1 def_readwrite

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost & TidalInputs::design_speed_ms
def_readwrite ("power_model", &TidalInputs::power_model) .def(pybind11 & Tidal::design_speed_ms
def_readwrite("power_model", &Tidal::power_model) .def_readwrite("power_model_string" (
            "power_model" ,
            &Tidal::power_model  )
```

5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference

Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- WavePowerProductionModel::WAVE_POWER_GAUSSIAN value ("WAVE_POWER_PARABOLOID", WavePowerProductionModel::WAVE_POWER_PARABOLOID) .value("WAVE_POWER_LOOKUP"
- WavePowerProductionModel::WAVE_POWER_GAUSSIAN WavePowerProductionModel::WAVE_POWER_LOOKUP value ("N_WAVE_POWER_PRODUCTION_MODELS", WavePowerProductionModel::N_WAVE_POWER↩ _PRODUCTION_MODELS)
- &WaveInputs::renewable_inputs def_readwrite ("resource_key", &WaveInputs::resource_key) .def_↩ readwrite("capital_cost"
- &WaveInputs::renewable_inputs &WaveInputs::capital_cost def_readwrite ("operation_maintenance_cost_↩ kWh", &WaveInputs::operation_maintenance_cost_kWh) .def_readwrite("design_significant_wave_height↩ _m"
- &WaveInputs::renewable_inputs &WaveInputs::capital_cost &WaveInputs::design_significant_wave_height_m def_readwrite ("design_energy_period_s", &WaveInputs::design_energy_period_s) .def_readwrite("power↩ _model"

Variables

- &WaveInputs::renewable_inputs &WaveInputs::capital_cost &WaveInputs::design_significant_wave_height_m &WaveInputs::power_model def_readwrite("path_2_normalized_performance_matrix", &WaveInputs↩ ::path_2_normalized_performance_matrix) .def(pybind11 &Wave::design_significant_wave_height_m def_readwrite ("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power_model"

5.30.1 Detailed Description

Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Wave class. Only public attributes/methods are bound!

5.30.2 Function Documentation

5.30.2.1 def_readwrite() [1/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
def_readwrite (
             "design_energy_period_s" ,
             &WaveInputs::design_energy_period_s  )
```

5.30.2.2 def_readwrite() [2/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost def_readwrite (
             "operation_maintenance_cost_kWh" ,
             &WaveInputs::operation_maintenance_cost_kWh  )
```

5.30.2.3 def_readwrite() [3/3]

```
& WaveInputs::renewable_inputs def_readwrite (
             "resource_key" ,
             &WaveInputs::resource_key  )
```

5.30.2.4 value() [1/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN WavePowerProductionModel::WAVE_POWER_LOOKUP
value (
            "N_WAVE_POWER_PRODUCTION_MODELS" ,
            WavePowerProductionModel::N_WAVE_POWER_PRODUCTION_MODELS  )
```

5.30.2.5 value() [2/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN value (
            "WAVE_POWER_PARABOLOID" ,
            WavePowerProductionModel::WAVE_POWER_PARABOLOID  )
```

5.30.3 Variable Documentation

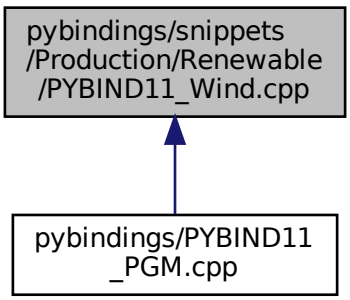
5.30.3.1 def_readwrite

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
& WaveInputs::power_model def_readwrite ( "path_2_normalized_performance_matrix", &Wave↩
Inputs::path_2_normalized_performance_matrix ) .def(pybind11 & Wave::design_significant_wave_height_m
def_readwrite("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power↩
_model" (
            "design_energy_period_s" ,
            &Wave::design_energy_period_s  )
```

5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference

Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- WindPowerProductionModel::WIND_POWER_EXPONENTIAL value ("WIND_POWER_LOOKUP", Wind↩ PowerProductionModel::WIND_POWER_LOOKUP) .value("N_WIND_POWER_PRODUCTION_MODELS"
- &WindInputs::renewable_inputs def_readwrite ("resource_key", &WindInputs::resource_key) .def_↩ readwrite("capital_cost"
- &WindInputs::renewable_inputs &WindInputs::capital_cost def_readwrite ("operation_maintenance_cost_↩ kWh", &WindInputs::operation_maintenance_cost_kWh) .def_readwrite("design_speed_ms"

Variables

- &WindInputs::renewable_inputs &WindInputs::capital_cost &WindInputs::design_speed_ms def_↩ readwrite("power_model", &WindInputs::power_model) .def(pybind11 &Wind::design_speed_ms def_readwrite ("power_model", &Wind::power_model) .def_readwrite("power_model_string"

5.31.1 Detailed Description

Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Wind class. Only public attributes/methods are bound!

5.31.2 Function Documentation

5.31.2.1 def_readwrite() [1/2]

```
& WindInputs::renewable_inputs & WindInputs::capital_cost def_readwrite (
             "operation_maintenance_cost_kWh" ,
             &WindInputs::operation_maintenance_cost_kWh  )
```

5.31.2.2 def_readwrite() [2/2]

```
& WindInputs::renewable_inputs def_readwrite (
             "resource_key" ,
             &WindInputs::resource_key  )
```

5.31.2.3 value()

```
WindPowerProductionModel::WIND_POWER_EXPONENTIAL value (
             "WIND_POWER_LOOKUP" ,
             WindPowerProductionModel::WIND_POWER_LOOKUP  )
```

5.31.3 Variable Documentation

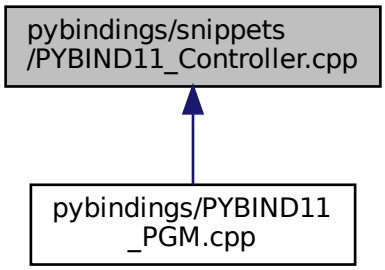
5.31.3.1 def_readwrite

```
& WindInputs::renewable_inputs & WindInputs::capital_cost & WindInputs::design_speed_ms def↩
_readwrite ("power_model", &WindInputs::power_model) .def(pybind11 & Wind::design_speed_ms
def_readwrite("power_model", &Wind::power_model) .def_readwrite("power_model_string" (
             "power_model" ,
             &Wind::power_model  )
```

5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference

Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- ControlMode::LOAD_FOLLOWING value ("CYCLE_CHARGING", ControlMode::CYCLE_CHARGING) .value("N_CONTROL_MODES"
- &Controller::control_mode def_readwrite ("control_string", &Controller::control_string) .def_readwrite("net↩ _load_vec_kW"
- &Controller::control_mode &Controller::net_load_vec_kW def_readwrite ("missed_load_vec_kW", &Controller↩ ::missed_load_vec_kW) .def_readwrite("combustion_map"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map def (pybind11↩ ::init<>()) .def("setControlMode"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode def ("init", &Controller::init) .def("applyDispatchControl"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode &Controller::applyDispatchControl def ("clear", &Controller::clear)

5.32.1 Detailed Description

Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Controller class. Only public attributes/methods are bound!

5.32.2 Function Documentation

5.32.2.1 def() [1/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
& Controller::applyDispatchControl def (
            "clear" ,
            &Controller::clear  )
```

5.32.2.2 def() [2/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
def (
            "init" ,
            &Controller::init  )
```

5.32.2.3 def() [3/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map def (
            pybind11::init<> () )
```

5.32.2.4 def_readwrite() [1/2]

```
& Controller::control_mode def_readwrite (
            "control_string" ,
            &Controller::control_string  )
```

5.32.2.5 def_readwrite() [2/2]

```
& Controller::control_mode & Controller::net_load_vec_kW def_readwrite (
             "missed_load_vec_kW" ,
             &Controller::missed_load_vec_kW  )
```

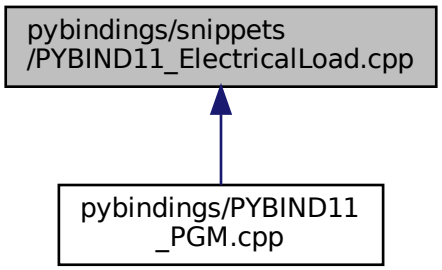
5.32.2.6 value()

```
ControlMode::LOAD_FOLLOWING value (
             "CYCLE_CHARGING" ,
             ControlMode::CYCLE_CHARGING  )
```

5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference

Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &ElectricalLoad::n_points def_readwrite ("n_years", &ElectricalLoad::n_years) .def_readwrite("min_load_↩ kW"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW def_readwrite ("mean_load_kW", &Electrical↩ Load::mean_load_kW) .def_readwrite("max_load_kW"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW &ElectricalLoad::max_load_kW def_readwrite ("path_2_electrical_load_time_series", &ElectricalLoad::path_2_electrical_load_time_series) .def_↩ readwrite("time_vec_hrs"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW &ElectricalLoad::max_load_kW &ElectricalLoad::time_vec_hrs def_readwrite ("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs) .def_readwrite("load_vec_kW"

5.33.1 Detailed Description

Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the ElectricalLoad class. Only public attributes/methods are bound!

5.33.2 Function Documentation

5.33.2.1 def_readwrite() [1/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW & ElectricalLoad::max_load_kW & ElectricalLoad::time_
def_readwrite (
             "dt_vec_hrs" ,
             &ElectricalLoad::dt_vec_hrs  )
```

5.33.2.2 def_readwrite() [2/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW def_readwrite (
             "mean_load_kW" ,
             &ElectricalLoad::mean_load_kW  )
```

5.33.2.3 def_readwrite() [3/4]

```
& ElectricalLoad::n_points def_readwrite (
             "n_years" ,
             &ElectricalLoad::n_years  )
```

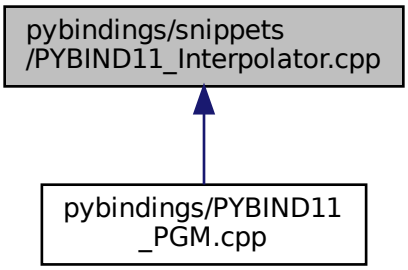
5.33.2.4 def_readwrite() [4/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW & ElectricalLoad::max_load_kW def_↩
readwrite (
             "path_2_electrical_load_time_series" ,
             &ElectricalLoad::path_2_electrical_load_time_series  )
```

5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference

Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &InterpolatorStruct1D::n_points def_readwrite ("x_vec", &InterpolatorStruct1D::x_vec) .def_readwrite("min↩
_x"
- &InterpolatorStruct1D::n_points &InterpolatorStruct1D::min_x def_readwrite ("max_x", &Interpolator↩
Struct1D::max_x) .def_readwrite("y_vec"
- &InterpolatorStruct1D::n_points &InterpolatorStruct1D::min_x &InterpolatorStruct1D::y_vec def (pybind11↩
::init())
- &InterpolatorStruct2D::n_rows def_readwrite ("n_cols", &InterpolatorStruct2D::n_cols) .def_readwrite("x_↩
vec"
- &InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec def_readwrite ("min_x", &InterpolatorStruct2↩
D::min_x) .def_readwrite("max_x"
- &InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x def_readwrite
("y_vec", &InterpolatorStruct2D::y_vec) .def_readwrite("min_y"
- &InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D::min_y
def_readwrite ("max_y", &InterpolatorStruct2D::max_y) .def_readwrite("z_matrix"
- &Interpolator::interp_map_1D def_readwrite ("path_map_1D", &Interpolator::path_map_1D) .def_↩
readwrite("interp_map_2D"

5.34.1 Detailed Description

Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Interpolator class. Only public attributes/methods are bound!

5.34.2 Function Documentation

5.34.2.1 def()

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x & InterpolatorStruct1D::y_vec
def (
            pybind11::init()  )
```

5.34.2.2 def_readwrite() [1/7]

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x def_readwrite (
            "max_x" ,
            &InterpolatorStruct1D::max_x  )
```

5.34.2.3 def_readwrite() [2/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x &
InterpolatorStruct2D::min_y def_readwrite (
            "max_y" ,
            &InterpolatorStruct2D::max_y  )
```

5.34.2.4 def_readwrite() [3/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec def_readwrite (
            "min_x" ,
            &InterpolatorStruct2D::min_x  )
```

5.34.2.5 def_readwrite() [4/7]

```
& InterpolatorStruct2D::n_rows def_readwrite (
            "n_cols" ,
            &InterpolatorStruct2D::n_cols  )
```

5.34.2.6 def_readwrite() [5/7]

```
& Interpolator::interp_map_1D def_readwrite (
            "path_map_1D" ,
            &Interpolator::path_map_1D  )
```

5.34.2.7 def_readwrite() [6/7]

```
& InterpolatorStruct1D::n_points def_readwrite (
             "x_vec" ,
             &InterpolatorStruct1D::x_vec  )
```

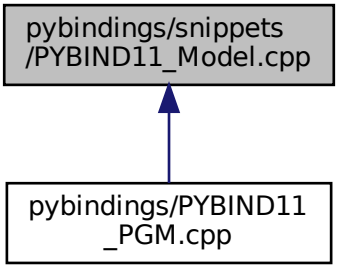
5.34.2.8 def_readwrite() [7/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x
def_readwrite (
             "y_vec" ,
             &InterpolatorStruct2D::y_vec  )
```

5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference

Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Variables

- &ModelInputs::path_2_electrical_load_time_series def_readwrite("control_mode", &ModelInputs::control_↩ mode) .def(pybind11 &Model::total_fuel_consumed_L def_readwrite ("total_emissions", &Model::total_↩ emissions) .def_readwrite("net_present_cost"

5.35.1 Detailed Description

Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Model class. Only public attributes/methods are bound!

5.35.2 Variable Documentation

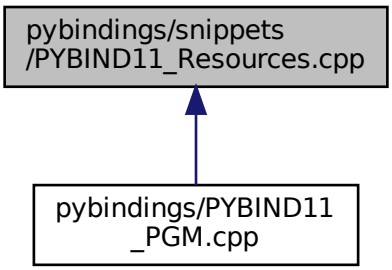
5.35.2.1 def_readwrite

```
& ModelInputs::path_2_electrical_load_time_series def_readwrite ("control_mode", &Model↩
Inputs::control_mode) .def(pybind11 & Model::total_fuel_consumed_L & Model::net_present_cost &
Model::total_dispatch_discharge_kWh & Model::controller & Model::resources & Model::noncombustion_ptr_vec
def_readwrite("renewable_ptr_vec", &Model::renewable_ptr_vec) .def_readwrite("storage_ptr_vec"
(
            "total_emissions" ,
            &Model::total_emissions  )
```

5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference

Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &Resources::resource_map_1D def_readwrite ("string_map_1D", &Resources::string_map_1D) .def_↩ readwrite("path_map_1D"
- &Resources::resource_map_1D &Resources::path_map_1D def_readwrite ("resource_map_2D", &Resources↩ ::resource_map_2D) .def_readwrite("string_map_2D"

5.36.1 Detailed Description

Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Resources class. Only public attributes/methods are bound!

5.36.2 Function Documentation

5.36.2.1 def_readwrite() [1/2]

```
& Resources::resource_map_1D & Resources::path_map_1D def_readwrite (
              "resource_map_2D" ,
              &Resources::resource_map_2D  )
```

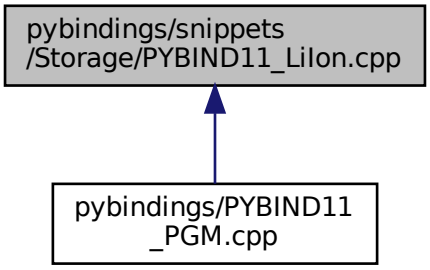
5.36.2.2 def_readwrite() [2/2]

```
& Resources::resource_map_1D def_readwrite (
              "string_map_1D" ,
              &Resources::string_map_1D  )
```

5.37 pybindings/snippets/Storage/PYBIND11_LiIon.cpp File Reference

Bindings file for the LiIon class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- &LiIonInputs::storage_inputs def_readwrite ("capital_cost", &LiIonInputs::capital_cost) .def_readwrite("operation↩ _maintenance_cost_kWh"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh def_readwrite ("init_SOC", &LiIonInputs::init_SOC) .def_readwrite("min_SOC"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC def_readwrite ("hysteresis_SOC", &LiIonInputs::hysteresis_SOC) .def_readwrite("max_SOC"

- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC def_readwrite ("charging_efficiency", &LilonInputs::charging_efficiency) .def_↩ readwrite("discharging_efficiency"
- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency def_readwrite ("replace_SOH", &LilonInputs↩ ::replace_SOH) .def_readwrite("power_degradation_flag"
- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::power_degradation_flag def_readwrite ("degradation_alpha", &LilonInputs::degradation_alpha) .def_readwrite("degradation_beta"
- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::power_degradation_flag &LilonInputs::degradation_beta def_readwrite ("degradation_B_hat_cal_0", &LilonInputs::degradation_↩ B_hat_cal_0) .def_readwrite("degradation_r_cal"
- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::power_degradation_flag &LilonInputs::degradation_beta &LilonInputs::degradation_r_cal def_readwrite ("degradation_Ea_cal_0", &LilonInputs::degradation_Ea_cal_0) .def_readwrite("degradation_a_cal"
- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::power_degradation_flag &LilonInputs::degradation_beta &LilonInputs::degradation_r_cal &LilonInputs::degradation_a_cal def_readwrite ("degradation_s_cal", &LilonInputs::degradation_s_cal) .def_readwrite("gas_constant_JmolK"

Variables

- &LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::power_degradation_flag &LilonInputs::degradation_beta &LilonInputs::degradation_r_cal &LilonInputs::degradation_a_cal &LilonInputs::gas_constant_ def_readwrite("gas_constant_JmolK", &LilonInputs::gas_constant_JmolK) .def(pybind11 &Lilon::power_degradation_flag def_readwrite ("dynamic_energy_capacity_kWh", &Lilon::dynamic_energy_capacity_kWh) .def_readwrite("dynamic↩ _power_capacity_kW"

5.37.1 Detailed Description

Bindings file for the Lilon class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Lilon class. Only public attributes/methods are bound!

5.37.2 Function Documentation

5.37.2.1 def_readwrite() [1/9]

```
& LiIonInputs::storage_inputs def_readwrite (
            "capital_cost" ,
            &LiIonInputs::capital_cost  )
```

5.37.2.2 def_readwrite() [2/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC def_readwrite (
             "charging_efficiency" ,
             &LiIonInputs::charging_efficiency  )
```

5.37.2.3 def_readwrite() [3/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
def_readwrite (
             "degradation_alpha" ,
             &LiIonInputs::degradation_alpha  )
```

5.37.2.4 def_readwrite() [4/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta def_readwrite (
             "degradation_B_hat_cal_0" ,
             &LiIonInputs::degradation_B_hat_cal_0  )
```

5.37.2.5 def_readwrite() [5/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal def_readwrite (
             "degradation_Ea_cal_0" ,
             &LiIonInputs::degradation_Ea_cal_0  )
```

5.37.2.6 def_readwrite() [6/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal & LiIonInputs::degradation_a_cal
def_readwrite (
             "degradation_s_cal" ,
             &LiIonInputs::degradation_s_cal  )
```

5.37.2.7 def_readwrite() [7/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
def_readwrite (
             "hysteresis_SOC" ,
             &LiIonInputs::hysteresis_SOC  )
```

5.37.2.8 def_readwrite() [8/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh def_readwrite (
             "init_SOC" ,
             &LiIonInputs::init_SOC  )
```

5.37.2.9 def_readwrite() [9/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency def_readwrite (
             "replace_SOH" ,
             &LiIonInputs::replace_SOH  )
```

5.37.3 Variable Documentation

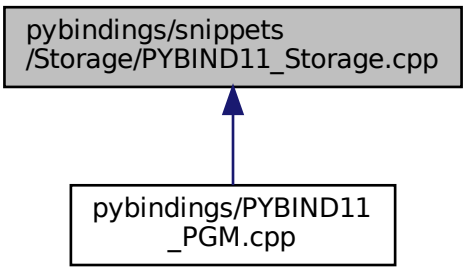
5.37.3.1 def_readwrite

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal & LiIonInputs::degradation_a_cal
& LiIonInputs::gas_constant_JmolK def_readwrite ("gas_constant_JmolK", &LiIonInputs::gas_↩
constant_JmolK) .def(pybind11 & LiIon::power_degradation_flag & LiIon::dynamic_power_capacity_kW
& LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal & LiIon::degradation_a_cal
& LiIon::gas_constant_JmolK & LiIon::init_SOC & LiIon::hysteresis_SOC & LiIon::charging_efficiency
def_readwrite("discharging_efficiency", &LiIon::discharging_efficiency) .def_readwrite("SOH_↩
vec" (
             "dynamic_energy_capacity_kWh" ,
             &LiIon::dynamic_energy_capacity_kWh  )
```

5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference

Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp.

This graph shows which files directly or indirectly include this file:



Functions

- StorageType::LIION value ("N_STORAGE_TYPES", StorageType::N_STORAGE_TYPES)
- &StorageInputs::print_flag def_readwrite ("is_sunk", &StorageInputs::is_sunk) .def_readwrite("power_↩ capacity_kW"
- &StorageInputs::print_flag &StorageInputs::power_capacity_kW def_readwrite ("energy_capacity_kWh", &StorageInputs::energy_capacity_kWh) .def_readwrite("nominal_inflation_annual"

Variables

- &StorageInputs::print_flag &StorageInputs::power_capacity_kW &StorageInputs::nominal_inflation_annual def_readwrite("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11 &Storage::type def_readwrite ("interpolator", &Storage::interpolator) .def_readwrite("print_flag"

5.38.1 Detailed Description

Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Storage class. Only public attributes/methods are bound!

5.38.2 Function Documentation

5.38.2.1 def_readwrite() [1/2]

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW def_readwrite (
              "energy_capacity_kWh" ,
              &StorageInputs::energy_capacity_kWh  )
```

5.38.2.2 def_readwrite() [2/2]

```
& StorageInputs::print_flag def_readwrite (
              "is_sunk" ,
              &StorageInputs::is_sunk  )
```

5.38.2.3 value()

```
StorageType::LIION value (
              "N_STORAGE_TYPES" ,
              StorageType::N_STORAGE_TYPES  )
```

5.38.3 Variable Documentation

5.38.3.1 def_readwrite

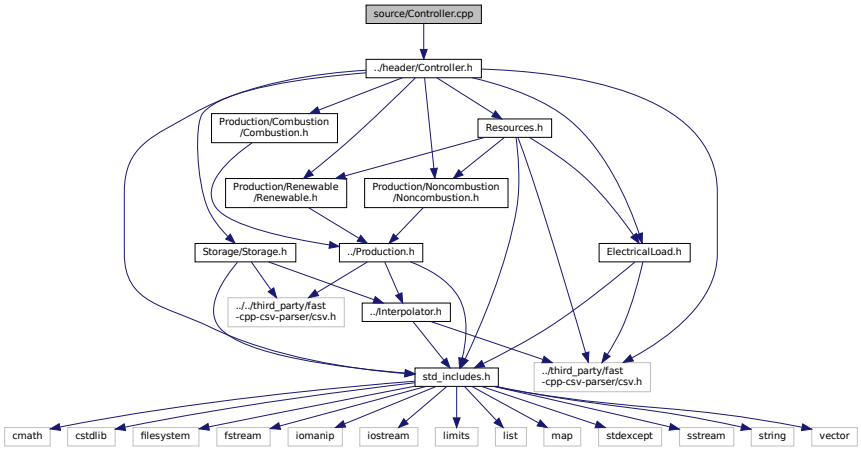
```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW & StorageInputs::nominal_inflation_annual
def_readwrite ("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11
& Storage::type & Storage::print_flag & Storage::is_sunk & Storage::n_replacements & Storage::power_capacity_k
& Storage::charge_kWh & Storage::nominal_inflation_annual & Storage::real_discount_annual &
Storage::operation_maintenance_cost_kWh & Storage::total_discharge_kWh & Storage::type_str &
Storage::charging_power_vec_kW def_readwrite("discharging_power_vec_kW", &Storage::discharging↩
_power_vec_kW) .def_readwrite("capital_cost_vec" (
              "interpolator" ,
              &Storage::interpolator  )
```

5.39 source/Controller.cpp File Reference

Implementation file for the Controller class.

```
#include "../header/Controller.h"
```

Include dependency graph for Controller.cpp:



5.39.1 Detailed Description

Implementation file for the Controller class.

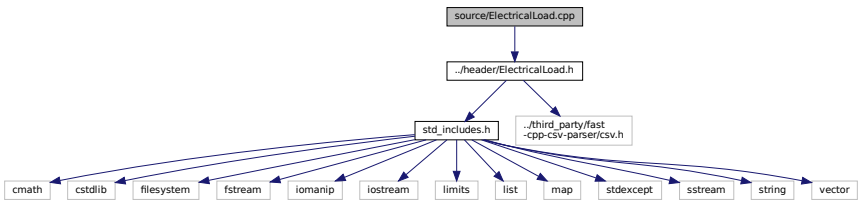
A class which contains a various dispatch control logic. Intended to serve as a component class of Controller.

5.40 source/ElectricalLoad.cpp File Reference

Implementation file for the ElectricalLoad class.

```
#include "../header/ElectricalLoad.h"
```

Include dependency graph for ElectricalLoad.cpp:



5.40.1 Detailed Description

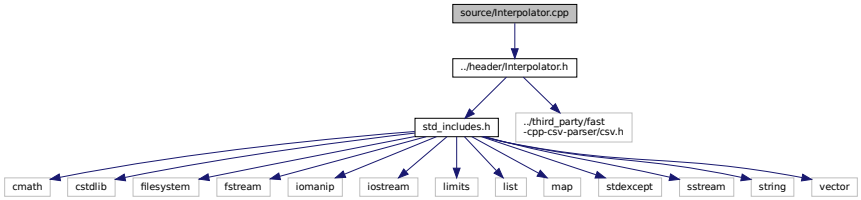
Implementation file for the ElectricalLoad class.

A class which contains time and electrical load data. Intended to serve as a component class of Model.

5.41 source/Interpolator.cpp File Reference

Implementation file for the Interpolator class.

`#include "../header/Interpolator.h"`
Include dependency graph for Interpolator.cpp:



5.41.1 Detailed Description

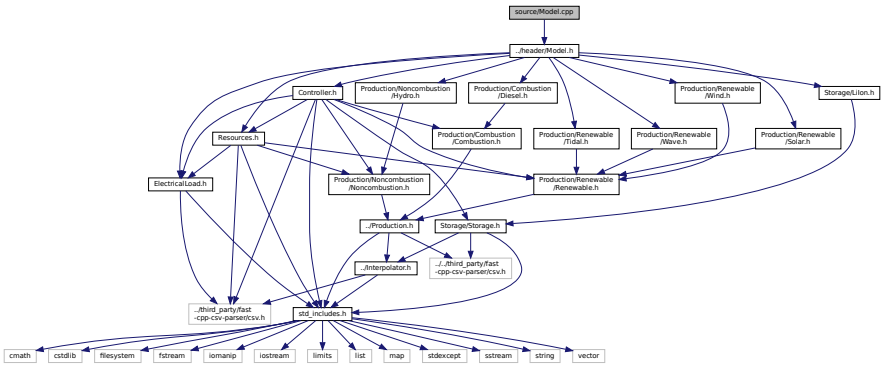
Implementation file for the Interpolator class.

A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies.

5.42 source/Model.cpp File Reference

Implementation file for the Model class.

`#include "../header/Model.h"`
Include dependency graph for Model.cpp:



5.42.1 Detailed Description

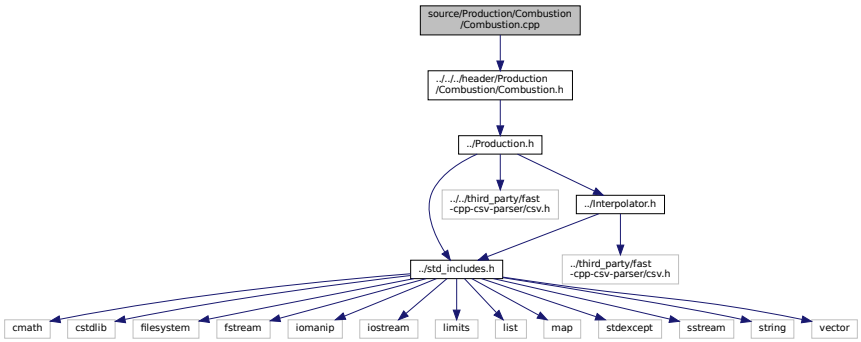
Implementation file for the Model class.

A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

5.43 source/Production/Combustion/Combustion.cpp File Reference

Implementation file for the Combustion class.

`#include "../../../header/Production/Combustion/Combustion.h"`
Include dependency graph for Combustion.cpp:



5.43.1 Detailed Description

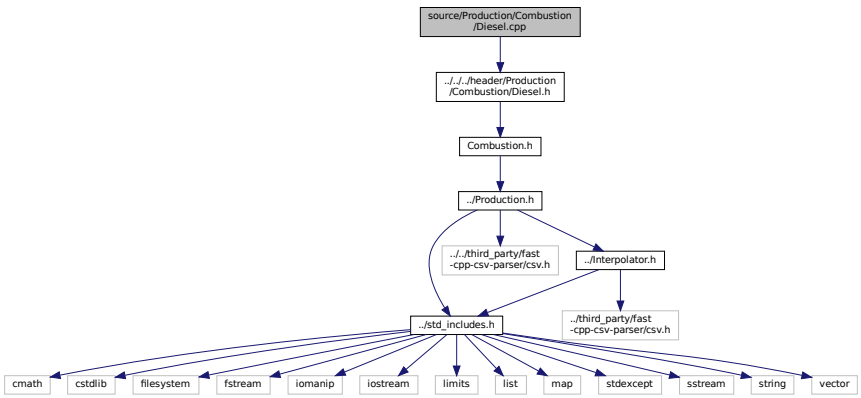
Implementation file for the Combustion class.

The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

5.44 source/Production/Combustion/Diesel.cpp File Reference

Implementation file for the Diesel class.

`#include "../../../header/Production/Combustion/Diesel.h"`
Include dependency graph for Diesel.cpp:



5.44.1 Detailed Description

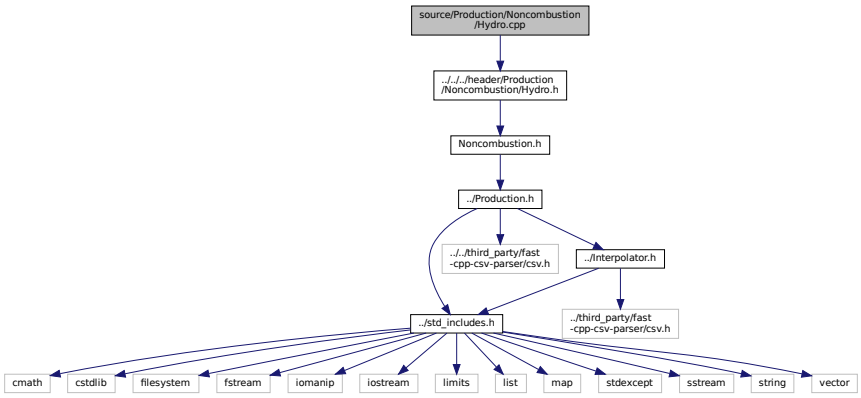
Implementation file for the Diesel class.

A derived class of the Combustion branch of Production which models production using a diesel generator.

5.45 source/Production/Noncombustion/Hydro.cpp File Reference

Implementation file for the Hydro class.

```
#include "../../../header/Production/Noncombustion/Hydro.h"
```
Include dependency graph for Hydro.cpp:



5.45.1 Detailed Description

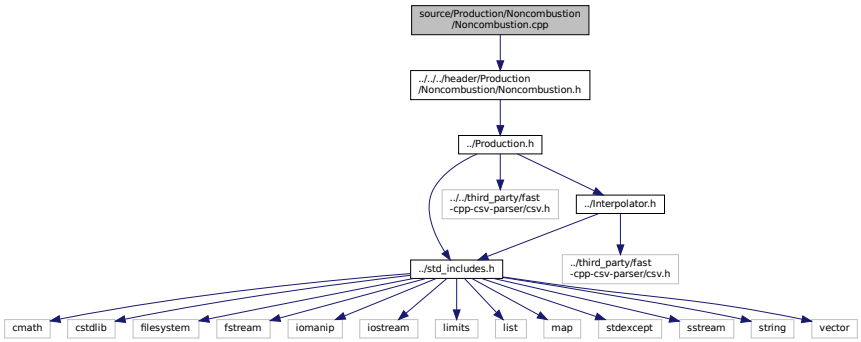
Implementation file for the Hydro class.

A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not).

5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference

Implementation file for the Noncombustion class.

`#include "../../../header/Production/Noncombustion/Noncombustion.h"`
Include dependency graph for Noncombustion.cpp:



5.46.1 Detailed Description

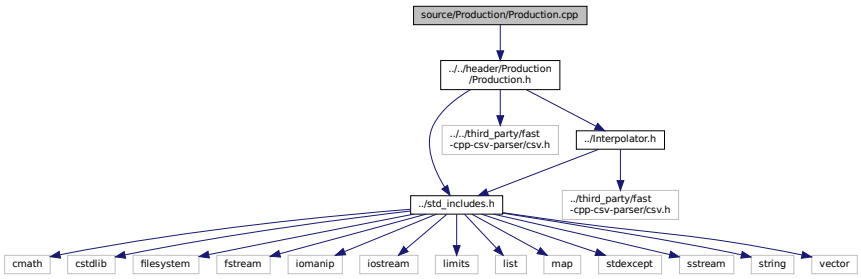
Implementation file for the Noncombustion class.

The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

5.47 source/Production/Production.cpp File Reference

Implementation file for the Production class.

`#include "../../header/Production/Production.h"`
Include dependency graph for Production.cpp:



5.47.1 Detailed Description

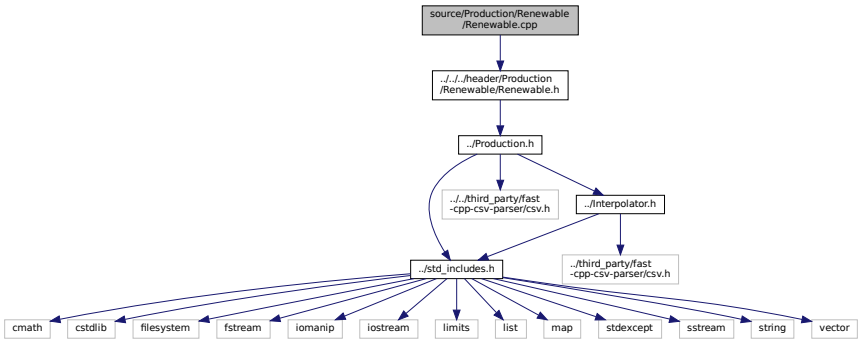
Implementation file for the Production class.

The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.48 source/Production/Renewable/Renewable.cpp File Reference

Implementation file for the Renewable class.

`#include "../../../header/Production/Renewable/Renewable.h"`
Include dependency graph for Renewable.cpp:



5.48.1 Detailed Description

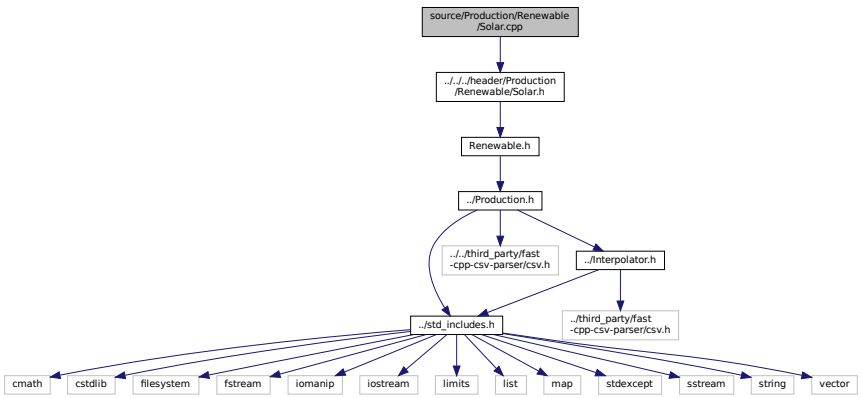
Implementation file for the Renewable class.

The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

5.49 source/Production/Renewable/Solar.cpp File Reference

Implementation file for the Solar class.

`#include "../../../header/Production/Renewable/Solar.h"`
Include dependency graph for Solar.cpp:



5.49.1 Detailed Description

Implementation file for the Solar class.

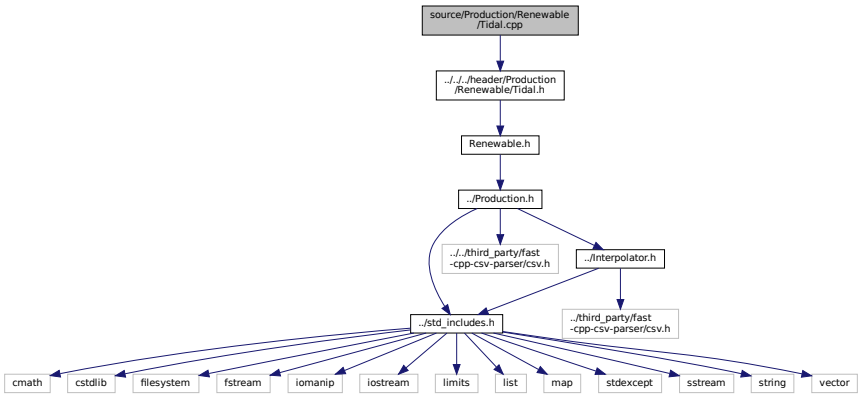
A derived class of the Renewable branch of Production which models solar production.

5.50 source/Production/Renewable/Tidal.cpp File Reference

Implementation file for the Tidal class.

```
#include "../../../header/Production/Renewable/Tidal.h"
```

Include dependency graph for Tidal.cpp:



5.50.1 Detailed Description

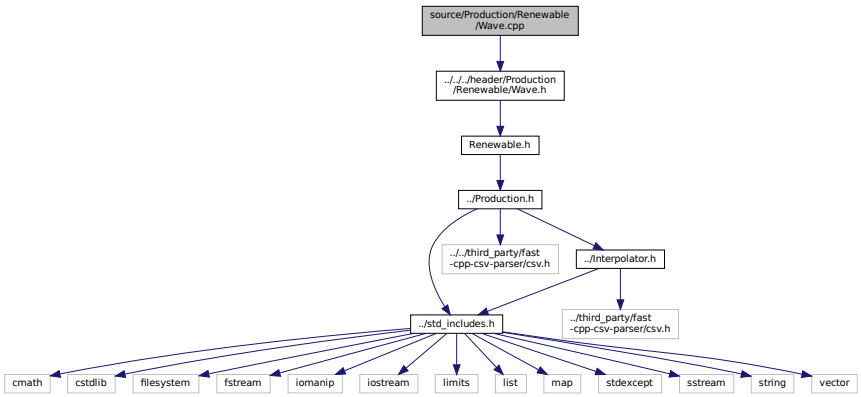
Implementation file for the Tidal class.

A derived class of the Renewable branch of Production which models tidal production.

5.51 source/Production/Renewable/Wave.cpp File Reference

Implementation file for the Wave class.

`#include "../../../header/Production/Renewable/Wave.h"`
Include dependency graph for Wave.cpp:



5.51.1 Detailed Description

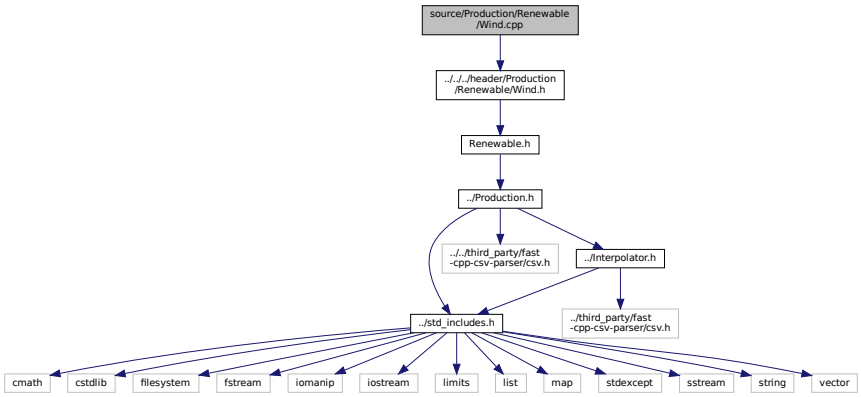
Implementation file for the Wave class.

A derived class of the Renewable branch of Production which models wave production.

5.52 source/Production/Renewable/Wind.cpp File Reference

Implementation file for the Wind class.

`#include "../../../header/Production/Renewable/Wind.h"`
Include dependency graph for Wind.cpp:



5.52.1 Detailed Description

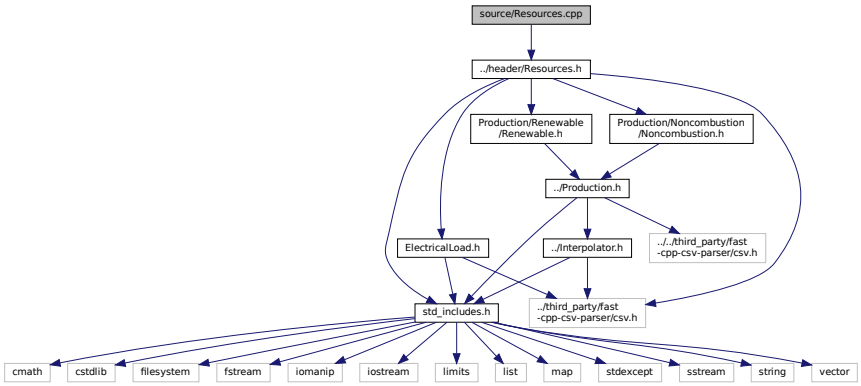
Implementation file for the Wind class.

A derived class of the Renewable branch of Production which models wind production.

5.53 source/Resources.cpp File Reference

Implementation file for the Resources class.

```
#include "../header/Resources.h"
```
Include dependency graph for Resources.cpp:



5.53.1 Detailed Description

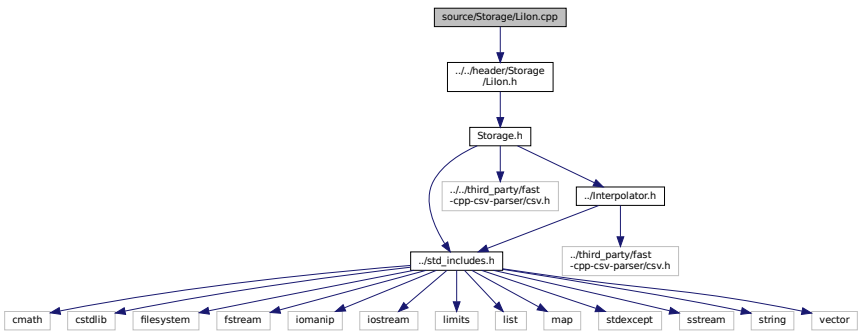
Implementation file for the Resources class.

A class which contains renewable resource data. Intended to serve as a component class of Model.

5.54 source/Storage/LiIon.cpp File Reference

Implementation file for the LiIon class.

```
#include "../../header/Storage/LiIon.h"
```
Include dependency graph for LiIon.cpp:



5.54.1 Detailed Description

Implementation file for the LiIon class.

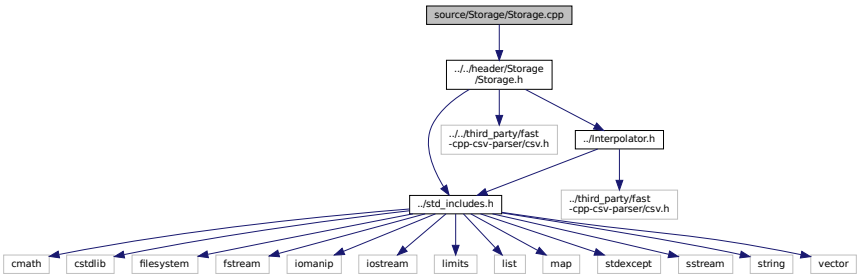
A derived class of Storage which models energy storage by way of lithium-ion batteries.

5.55 source/Storage/Storage.cpp File Reference

Implementation file for the Storage class.

```
#include "../../header/Storage/Storage.h"
```

Include dependency graph for Storage.cpp:



5.55.1 Detailed Description

Implementation file for the Storage class.

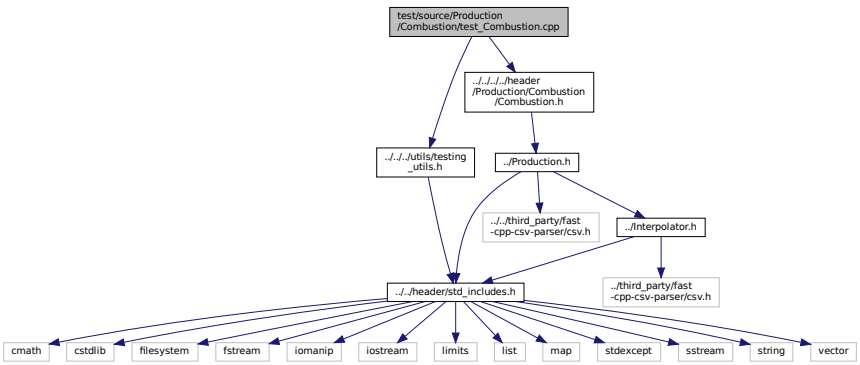
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

5.56 test/source/Production/Combustion/test_Combustion.cpp File Reference

Testing suite for Combustion class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Combustion/Combustion.h"
```

Include dependency graph for test_Combustion.cpp:



Functions

- Combustion * testConstruct_Combustion (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Combustion object and spot check some post-construction attributes.
- int main (int argc, char **argv)

5.56.1 Detailed Description

Testing suite for Combustion class.

A suite of tests for the Combustion class.

5.56.2 Function Documentation

5.56.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
122 {
123     #ifdef _WIN32
124         activateVirtualTerminal();
125     #endif  /* _WIN32 */
126 
127     printGold("\tTesting Production <-- Combustion");
128 
129     srand(time(NULL));
130 
131 
132     std::vector<double> time_vec_hrs (8760, 0);
133     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
134         time_vec_hrs[i] = i;
135     }
136 
137     Combustion* test_combustion_ptr = testConstruct_Combustion(&time_vec_hrs);
138 
139 
140     try {
141         //...
142     }
143 
144 
145     catch (...) {
146         delete test_combustion_ptr;
147 
148         printGold(" ................. ");
149         printRed("FAIL");
150         std::cout << std::endl;
151         throw;
152     }
153 
154 
155     delete test_combustion_ptr;
156 
157     printGold(" ................. ");
158     printGreen("PASS");
159     std::cout << std::endl;
160     return 0;
161 
162 }   /* main() */
```

5.56.2.2 testConstruct_Combustion()

```
Combustion * testConstruct_Combustion (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Combustion object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Combustion object.

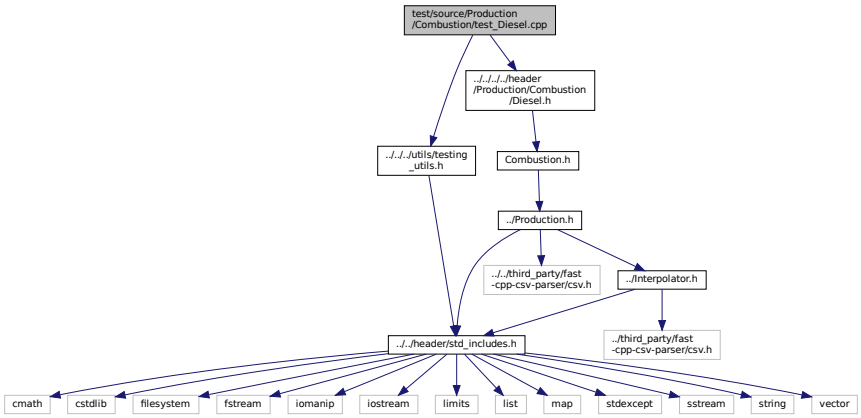
```
40 {
41     CombustionInputs combustion_inputs;
42 
43     Combustion* test_combustion_ptr = new Combustion(
44         8760,
45         1,
46         combustion_inputs,
47         time_vec_hrs_ptr
48     );
49 
50     testTruth(
51         not combustion_inputs.production_inputs.print_flag,
52         __FILE__,
53         __LINE__
54     );
55 
56     testFloatEquals(
57         test_combustion_ptr->fuel_consumption_vec_L.size(),
58         8760,
59         __FILE__,
60         __LINE__
61     );
62 
63     testFloatEquals(
64         test_combustion_ptr->fuel_cost_vec.size(),
65         8760,
66         __FILE__,
67         __LINE__
68     );
69 
70     testFloatEquals(
71         test_combustion_ptr->CO2_emissions_vec_kg.size(),
72         8760,
73         __FILE__,
74         __LINE__
75     );
76 
77     testFloatEquals(
78         test_combustion_ptr->CO_emissions_vec_kg.size(),
79         8760,
80         __FILE__,
81         __LINE__
82     );
83 
84     testFloatEquals(
85         test_combustion_ptr->NOx_emissions_vec_kg.size(),
86         8760,
87         __FILE__,
88         __LINE__
89     );
90 
91     testFloatEquals(
92         test_combustion_ptr->SOx_emissions_vec_kg.size(),
93         8760,
94         __FILE__,
95         __LINE__
96     );
97 
98     testFloatEquals(
99         test_combustion_ptr->CH4_emissions_vec_kg.size(),
100         8760,
101         __FILE__,
102         __LINE__
103     );
104 
105     testFloatEquals(
106         test_combustion_ptr->PM_emissions_vec_kg.size(),
107         8760,
108         __FILE__,
109         __LINE__
110     );
111 
112     return test_combustion_ptr;
113 }   /* testConstruct_Combustion() */
```

5.57 test/source/Production/Combustion/test_Diesel.cpp File Reference

Testing suite for Diesel class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Combustion/Diesel.h"
```

Include dependency graph for test_Diesel.cpp:



Functions

- Combustion * testConstruct_Diesel (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Diesel object and spot check some post-construction attributes.
- Combustion * testConstructLookup_Diesel (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Diesel object using fuel consumption lookup.
- void testBadConstruct_Diesel (std::vector< double > *time_vec_hrs_ptr)

  Function to test the trying to construct a Diesel object given bad inputs is being handled as expected.
- void testCapacityConstraint_Diesel (Combustion *test_diesel_ptr)

  Test to check that the installed capacity constraint is active and behaving as expected.
- void testMinimumLoadRatioConstraint_Diesel (Combustion *test_diesel_ptr)

  Test to check that the minimum load ratio constraint is active and behaving as expected.
- void testCommit_Diesel (Combustion *test_diesel_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Diesel object.
- void testMinimumRuntimeConstraint_Diesel (Combustion *test_diesel_ptr)

  Function to check that the minimum runtime constraint is active and behaving as expected.
- void testFuelConsumptionEmissions_Diesel (Combustion *test_diesel_ptr)

  Function to test that post-commit fuel consumption and emissions are > 0 when the test Diesel object is running, and = 0 when it is not (as expected).
- void testEconomics_Diesel (Combustion *test_diesel_ptr)

  Function to test that the post-commit model economics for the test Diesel object are as expected (> 0 when running, = 0 when not).
- void testFuelLookup_Diesel (Combustion *test_diesel_lookup_ptr)

  Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.
- int main (int argc, char **argv)

5.57.1 Detailed Description

Testing suite for Diesel class.

A suite of tests for the Diesel class.

5.57.2 Function Documentation

5.57.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
698 {
699     #ifdef _WIN32
700         activateVirtualTerminal();
701     #endif  /* _WIN32 */
702 
703     printGold("\tTesting Production <-- Combustion <-- Diesel");
704 
705     srand(time(NULL));
706 
707 
708     std::vector<double> time_vec_hrs (8760, 0);
709     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
710         time_vec_hrs[i] = i;
711     }
712 
713 
714     Combustion* test_diesel_ptr = testConstruct_Diesel(&time_vec_hrs);
715     Combustion* test_diesel_lookup_ptr = testConstructLookup_Diesel(&time_vec_hrs);
716 
717     try {
718         testBadConstruct_Diesel(&time_vec_hrs);
719 
720         testCapacityConstraint_Diesel(test_diesel_ptr);
721         testMinimumLoadRatioConstraint_Diesel(test_diesel_ptr);
722 
723         testCommit_Diesel(test_diesel_ptr);
724 
725         testMinimumRuntimeConstraint_Diesel(test_diesel_ptr);
726 
727         testFuelConsumptionEmissions_Diesel(test_diesel_ptr);
728         testEconomics_Diesel(test_diesel_ptr);
729 
730         testFuelLookup_Diesel(test_diesel_lookup_ptr);
731     }
732 
733 
734     catch (...) {
735         delete test_diesel_ptr;
736         delete test_diesel_lookup_ptr;
737 
738         printGold(" ..... ");
739         printRed("FAIL");
740         std::cout « std::endl;
741         throw;
742     }
743 
744 
745     delete test_diesel_ptr;
746     delete test_diesel_lookup_ptr;
747 
748     printGold(" ..... ");
749     printGreen("PASS");
750     std::cout « std::endl;
751     return 0;
752 
753 }   /* main() */
```

5.57.2.2 testBadConstruct_Diesel()

```
void testBadConstruct_Diesel (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Diesel object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
171 {
172     bool error_flag = true;
173 
174     try {
175         DieselInputs bad_diesel_inputs;
176         bad_diesel_inputs.fuel_cost_L = -1;
177 
178         Diesel bad_diesel(
179             8760,
180             1,
181             bad_diesel_inputs,
182             time_vec_hrs_ptr
183         );
184 
185         error_flag = false;
186     } catch (...) {
187         // Task failed successfully! =P
188     }
189     if (not error_flag) {
190         expectedErrorNotDetected(__FILE__, __LINE__);
191     }
192 
193     return;
194 }   /* testBadConstruct_Diesel() */
```

5.57.2.3 testCapacityConstraint_Diesel()

```
void testCapacityConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Test to check that the installed capacity constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
212 {
213     testFloatEquals(
214         test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
215         test_diesel_ptr->capacity_kW,
216         __FILE__,
217         __LINE__
218     );
219 
220     return;
221 }   /* testCapacityConstraint_Diesel() */
```

5.57.2.4 testCommit_Diesel()

```
void testCommit_Diesel (
            Combustion * test_diesel_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Diesel object.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
271 {
272     std::vector<double> dt_vec_hrs (48, 1);
273 
274     std::vector<double> load_vec_kW = {
275         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
276         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
277         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
278         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
279     };
280 
281     double load_kW = 0;
282     double production_kW = 0;
283     double roll = 0;
284 
285     for (int i = 0; i < 48; i++) {
286         roll = (double)rand() / RAND_MAX;
287 
288         if (roll >= 0.95) {
289             roll = 1.25;
290         }
291 
292         load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
293         load_kW = load_vec_kW[i];
294 
295         production_kW = test_diesel_ptr->requestProductionkW(
296             i,
297             dt_vec_hrs[i],
298             load_kW
299         );
300 
301         load_kW = test_diesel_ptr->commit(
302             i,
303             dt_vec_hrs[i],
304             production_kW,
305             load_kW
306         );
307 
308         // load_kW <= load_vec_kW (i.e., after vs before)
309         testLessThanOrEqualTo(
310             load_kW,
311             load_vec_kW[i],
312             __FILE__,
313             __LINE__
314         );
315 
316         // production = dispatch + storage + curtailment
317         testFloatEquals(
318             test_diesel_ptr->production_vec_kW[i] -
319             test_diesel_ptr->dispatch_vec_kW[i] -
320             test_diesel_ptr->storage_vec_kW[i] -
321             test_diesel_ptr->curtailment_vec_kW[i],
322             0,
323             __FILE__,
324             __LINE__
325         );
326 
327         // capacity constraint
328         if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
329             testFloatEquals(
330                 test_diesel_ptr->production_vec_kW[i],
331                 test_diesel_ptr->capacity_kW,
332                 __FILE__,
333                 __LINE__
334             );
335         }
336 
337         // minimum load ratio constraint
338         else if (
339             test_diesel_ptr->is_running and
340             test_diesel_ptr->production_vec_kW[i] > 0 and
341             load_vec_kW[i] <
342             ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
343         ) {
344             testFloatEquals(
345                 test_diesel_ptr->production_vec_kW[i],
346                 ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
347                     test_diesel_ptr->capacity_kW,
```

```
348                 __FILE__,
349                 __LINE__
350             );
351         }
352     }
353
354     return;
355 }   /* testCommit_Diesel() */
```

5.57.2.5 testConstruct_Diesel()

```
Combustion * testConstruct_Diesel (
             std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Diesel object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Combustion pointer to a test Diesel object.

```
40 {
41     DieselInputs diesel_inputs;
42
43     Combustion* test_diesel_ptr =  new Diesel(
44         8760,
45         1,
46         diesel_inputs,
47         time_vec_hrs_ptr
48     );
49
50     testTruth(
51         not diesel_inputs.combustion_inputs.production_inputs.print_flag,
52         __FILE__,
53         __LINE__
54     );
55
56     testFloatEquals(
57         test_diesel_ptr->type,
58         CombustionType :: DIESEL,
59         __FILE__,
60         __LINE__
61     );
62
63     testTruth(
64         test_diesel_ptr->type_str == "DIESEL",
65         __FILE__,
66         __LINE__
67     );
68
69     testFloatEquals(
70         test_diesel_ptr->linear_fuel_slope_LkWh,
71         0.265675,
72         __FILE__,
73         __LINE__
74     );
75
76     testFloatEquals(
77         test_diesel_ptr->linear_fuel_intercept_LkWh,
78         0.026676,
79         __FILE__,
80         __LINE__
81     );
82
83     testFloatEquals(
84         test_diesel_ptr->capital_cost,
85         94125.375446,
86         __FILE__,
87         __LINE__
```

```
88      );
89
90      testFloatEquals(
91          test_diesel_ptr->operation_maintenance_cost_kWh,
92          0.069905,
93          __FILE__,
94          __LINE__
95      );
96
97      testFloatEquals(
98          ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
99          0.2,
100          __FILE__,
101          __LINE__
102      );
103
104      testFloatEquals(
105          ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
106          4,
107          __FILE__,
108          __LINE__
109      );
110
111      testFloatEquals(
112          test_diesel_ptr->replace_running_hrs,
113          30000,
114          __FILE__,
115          __LINE__
116      );
117
118      return test_diesel_ptr;
119  }   /* testConstruct_Diesel() */
```

5.57.2.6 testConstructLookup_Diesel()

```
Combustion * testConstructLookup_Diesel (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Diesel object using fuel consumption lookup.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Combustion pointer to a test Diesel object.

```
138 {
139      DieselInputs diesel_inputs;
140
141      diesel_inputs.combustion_inputs.fuel_mode = FuelMode :: FUEL_MODE_LOOKUP;
142      diesel_inputs.combustion_inputs.path_2_fuel_interp_data =
143          "data/test/interpolation/diesel_fuel_curve.csv";
144
145      Combustion* test_diesel_lookup_ptr = new Diesel(
146          8760,
147          1,
148          diesel_inputs,
149          time_vec_hrs_ptr
150      );
151
152      return test_diesel_lookup_ptr;
153  }   /* testConstructLookup_Diesel() */
```

5.57.2.7 testEconomics_Diesel()

```
void testEconomics_Diesel (
            Combustion * test_diesel_ptr )
```

Function to test that the post-commit model economics for the test Diesel object are as expected ($>$ 0 when running, = 0 when not).

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
575 {
576     std::vector<bool> expected_is_running_vec = {
577         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
578         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
579         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
580         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
581     };
582
583     bool is_running = false;
584
585     for (int i = 0; i < 48; i++) {
586         is_running = test_diesel_ptr->is_running_vec[i];
587
588         testFloatEquals(
589             is_running,
590             expected_is_running_vec[i],
591             __FILE__,
592             __LINE__
593         );
594
595         // O&M, fuel consumption, and emissions > 0 whenever diesel is running
596         if (is_running) {
597             testGreaterThan(
598                 test_diesel_ptr->operation_maintenance_cost_vec[i],
599                 0,
600                 __FILE__,
601                 __LINE__
602             );
603         }
604
605         // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
606         else {
607             testFloatEquals(
608                 test_diesel_ptr->operation_maintenance_cost_vec[i],
609                 0,
610                 __FILE__,
611                 __LINE__
612             );
613         }
614     }
615
616     return;
617 }   /* testEconomics_Diesel() */
```

5.57.2.8 testFuelConsumptionEmissions_Diesel()

```
void testFuelConsumptionEmissions_Diesel (
            Combustion * test_diesel_ptr )
```

Function to test that post-commit fuel consumption and emissions are $>$ 0 when the test Diesel object is running, and = 0 when it is not (as expected).

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
417 {
418     std::vector<bool> expected_is_running_vec = {
419         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
420         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
421         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
422         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
423     };
424 
425     bool is_running = false;
426 
427     for (int i = 0; i < 48; i++) {
428         is_running = test_diesel_ptr->is_running_vec[i];
429 
430         testFloatEquals(
431             is_running,
432             expected_is_running_vec[i],
433             __FILE__,
434             __LINE__
435         );
436 
437         // O&M, fuel consumption, and emissions > 0 whenever diesel is running
438         if (is_running) {
439             testGreaterThan(
440                 test_diesel_ptr->fuel_consumption_vec_L[i],
441                 0,
442                 __FILE__,
443                 __LINE__
444             );
445 
446             testGreaterThan(
447                 test_diesel_ptr->fuel_cost_vec[i],
448                 0,
449                 __FILE__,
450                 __LINE__
451             );
452 
453             testGreaterThan(
454                 test_diesel_ptr->CO2_emissions_vec_kg[i],
455                 0,
456                 __FILE__,
457                 __LINE__
458             );
459 
460             testGreaterThan(
461                 test_diesel_ptr->CO_emissions_vec_kg[i],
462                 0,
463                 __FILE__,
464                 __LINE__
465             );
466 
467             testGreaterThan(
468                 test_diesel_ptr->NOx_emissions_vec_kg[i],
469                 0,
470                 __FILE__,
471                 __LINE__
472             );
473 
474             testGreaterThan(
475                 test_diesel_ptr->SOx_emissions_vec_kg[i],
476                 0,
477                 __FILE__,
478                 __LINE__
479             );
480 
481             testGreaterThan(
482                 test_diesel_ptr->CH4_emissions_vec_kg[i],
483                 0,
484                 __FILE__,
485                 __LINE__
486             );
487 
488             testGreaterThan(
489                 test_diesel_ptr->PM_emissions_vec_kg[i],
490                 0,
491                 __FILE__,
492                 __LINE__
493             );
494         }
495 
496         // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
497         else {
498             testFloatEquals(
499                 test_diesel_ptr->fuel_consumption_vec_L[i],
500                 0,
501                 __FILE__,
502                 __LINE__
503             );
```

```
504
505             testFloatEquals(
506                 test_diesel_ptr->fuel_cost_vec[i],
507                 0,
508                 __FILE__,
509                 __LINE__
510             );
511
512             testFloatEquals(
513                 test_diesel_ptr->CO2_emissions_vec_kg[i],
514                 0,
515                 __FILE__,
516                 __LINE__
517             );
518
519             testFloatEquals(
520                 test_diesel_ptr->CO_emissions_vec_kg[i],
521                 0,
522                 __FILE__,
523                 __LINE__
524             );
525
526             testFloatEquals(
527                 test_diesel_ptr->NOx_emissions_vec_kg[i],
528                 0,
529                 __FILE__,
530                 __LINE__
531             );
532
533             testFloatEquals(
534                 test_diesel_ptr->SOx_emissions_vec_kg[i],
535                 0,
536                 __FILE__,
537                 __LINE__
538             );
539
540             testFloatEquals(
541                 test_diesel_ptr->CH4_emissions_vec_kg[i],
542                 0,
543                 __FILE__,
544                 __LINE__
545             );
546
547             testFloatEquals(
548                 test_diesel_ptr->PM_emissions_vec_kg[i],
549                 0,
550                 __FILE__,
551                 __LINE__
552             );
553         }
554     }
555
556     return;
557 }   /* testFuelConsumptionEmissions_Diesel() */
```

5.57.2.9 testFuelLookup_Diesel()

```
void testFuelLookup_Diesel (
            Combustion * test_diesel_lookup_ptr )
```

Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.

Parameters

test_diesel_lookup_ptr	A Combustion pointer to the test Diesel object using fuel consumption lookup.

```
636 {
637     std::vector<double> load_ratio_vec = {
638         0,
639         0.170812859791767,
640         0.322739274162545,
641         0.369750203682042,
642         0.443532869135929,
643         0.471567864244626,
644         0.536513734479662,
```

```
645         0.586125806988674,
646         0.601101175455075,
647         0.658356862575221,
648         0.70576929893201,
649         0.784069734739331,
650         0.805765927542453,
651         0.884747873186048,
652         0.930870496062112,
653         0.979415217694769,
654         1
655     };
656
657     std::vector<double> expected_fuel_consumption_vec_L = {
658         4.68079520372916,
659         8.35159603357656,
660         11.7422361561399,
661         12.9931187917615,
662         14.8786636301325,
663         15.5746957307243,
664         17.1419229487141,
665         18.3041866133728,
666         18.6530540913696,
667         19.9569217633299,
668         21.012354614584,
669         22.7142305879957,
670         23.1916726441968,
671         24.8602332554707,
672         25.8172124624032,
673         26.8256741279932,
674         27.254952
675     };
676
677     for (size_t i = 0; i < load_ratio_vec.size(); i++) {
678         testFloatEquals(
679             test_diesel_lookup_ptr->getFuelConsumptionL(
680                 1, load_ratio_vec[i] * test_diesel_lookup_ptr->capacity_kW
681             ),
682             expected_fuel_consumption_vec_L[i],
683             __FILE__,
684             __LINE__
685         );
686     }
687
688     return;
689 }   /* testFuelLookup_Diesel() */
```

5.57.2.10 testMinimumLoadRatioConstraint_Diesel()

```
void testMinimumLoadRatioConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Test to check that the minimum load ratio constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
239 {
240     testFloatEquals(
241         test_diesel_ptr->requestProductionkW(
242             0,
243             1,
244             0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
245                 test_diesel_ptr->capacity_kW
246         ),
247         ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
248         __FILE__,
249         __LINE__
250     );
251
252     return;
253 }   /* testMinimumLoadRatioConstraint_Diesel() */
```

5.57.2.11 testMinimumRuntimeConstraint_Diesel()

```
void testMinimumRuntimeConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Function to check that the minimum runtime constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

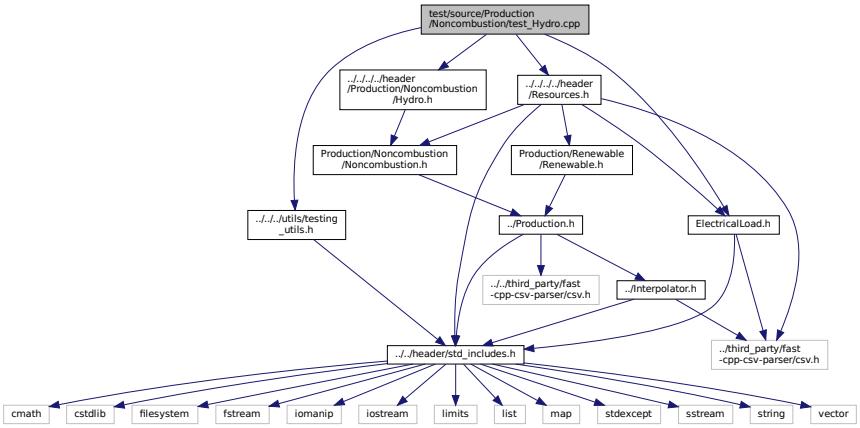
```
373 {
374     std::vector<double> load_vec_kW = {
375         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
376         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
377         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
378         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
379     };
380
381     std::vector<bool> expected_is_running_vec = {
382         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
383         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
384         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
385         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
386     };
387
388     for (int i = 0; i < 48; i++) {
389         testFloatEquals(
390             test_diesel_ptr->is_running_vec[i],
391             expected_is_running_vec[i],
392             __FILE__,
393             __LINE__
394         );
395     }
396
397     return;
398 }   /* testMinimumRuntimeConstraint_Diesel() */
```

5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference

Testing suite for Hydro class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Resources.h"
#include "../../../../header/ElectricalLoad.h"
#include "../../../../header/Production/Noncombustion/Hydro.h"
```

Include dependency graph for test_Hydro.cpp:



Functions

- Noncombustion ∗ testConstruct_Hydro (HydroInputs hydro_inputs, std::vector< double > ∗time_vec_hrs_↩ ptr)

  A function to construct a Hydro object and spot check some post-construction attributes.
- void testEfficiencyInterpolation_Hydro (Noncombustion ∗test_hydro_ptr)

  Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.
- void testCommit_Hydro (Noncombustion ∗test_hydro_ptr, Resources ∗test_resources_ptr)
- int main (int argc, char ∗∗argv)

5.58.1 Detailed Description

Testing suite for Hydro class.

A suite of tests for the Hydro class.

5.58.2 Function Documentation

5.58.2.1 main()

```
int main (
            int argc,
            char ** argv )
305 {
306     #ifdef _WIN32
307         activateVirtualTerminal();
308     #endif  /* _WIN32 */
309
310     printGold("\tTesting Production <-- Noncombustion <-- Hydro");
311
312     srand(time(NULL));
```

```
313
314
315     std::vector<double> time_vec_hrs (8760, 0);
316     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
317         time_vec_hrs[i] = i;
318     }
319
320     std::string path_2_electrical_load_time_series =
321         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
322
323     ElectricalLoad* test_electrical_load_ptr =
324         new ElectricalLoad(path_2_electrical_load_time_series);
325
326     Resources* test_resources_ptr = new Resources();
327
328     HydroInputs hydro_inputs;
329     int hydro_resource_key = 0;
330
331     hydro_inputs.reservoir_capacity_m3 = 10000;
332     hydro_inputs.resource_key = hydro_resource_key;
333
334     Noncombustion* test_hydro_ptr = testConstruct_Hydro(hydro_inputs, &time_vec_hrs);
335
336     std::string path_2_hydro_resource_data =
337         "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
338
339     test_resources_ptr->addResource(
340         NoncombustionType::HYDRO,
341         path_2_hydro_resource_data,
342         hydro_resource_key,
343         test_electrical_load_ptr
344     );
345
346
347     try {
348         testEfficiencyInterpolation_Hydro(test_hydro_ptr);
349         testCommit_Hydro(test_hydro_ptr, test_resources_ptr);
350     }
351
352
353     catch (...) {
354         delete test_electrical_load_ptr;
355         delete test_resources_ptr;
356         delete test_hydro_ptr;
357
358         printGold(" ... ");
359         printRed("FAIL");
360         std::cout « std::endl;
361         throw;
362     }
363
364
365     delete test_electrical_load_ptr;
366     delete test_resources_ptr;
367     delete test_hydro_ptr;
368
369     printGold(" ... ");
370     printGreen("PASS");
371     std::cout « std::endl;
372     return 0;
373
374 }   /* main() */
```

5.58.2.2 testCommit_Hydro()

```
void testCommit_Hydro (
             Noncombustion * test_hydro_ptr,
             Resources * test_resources_ptr )
222 {
223     double load_kW = 100 * (double)rand() / RAND_MAX;
224     double production_kW = 0;
225
226     for (int i = 0; i < 8760; i++) {
227         production_kW = test_hydro_ptr->requestProductionkW(
228             i,
229             1,
230             load_kW,
231             test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
232         );
```

```
233
234         load_kW = test_hydro_ptr->commit(
235             i,
236             1,
237             production_kW,
238             load_kW,
239             test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
240         );
241
242         testGreaterThanOrEqualTo(
243             test_hydro_ptr->production_vec_kW[i],
244             0,
245             __FILE__,
246             __LINE__
247         );
248
249         testLessThanOrEqualTo(
250             test_hydro_ptr->production_vec_kW[i],
251             test_hydro_ptr->capacity_kW,
252             __FILE__,
253             __LINE__
254         );
255
256         testFloatEquals(
257             test_hydro_ptr->production_vec_kW[i] -
258             test_hydro_ptr->dispatch_vec_kW[i] -
259             test_hydro_ptr->curtailment_vec_kW[i] -
260             test_hydro_ptr->storage_vec_kW[i],
261             0,
262             __FILE__,
263             __LINE__
264         );
265
266         testGreaterThanOrEqualTo(
267             ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
268             0,
269             __FILE__,
270             __LINE__
271         );
272
273         testLessThanOrEqualTo(
274             ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
275             ((Hydro*)test_hydro_ptr)->maximum_flow_m3hr,
276             __FILE__,
277             __LINE__
278         );
279
280         testGreaterThanOrEqualTo(
281             ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
282             0,
283             __FILE__,
284             __LINE__
285         );
286
287         testLessThanOrEqualTo(
288             ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
289             ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
290             __FILE__,
291             __LINE__
292         );
293     }
294
295     return;
296 }   /* testCommit_Hydro() */
```

5.58.2.3 testConstruct_Hydro()

```
Hydro *Noncombustion * testConstruct_Hydro (
            HydroInputs hydro_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Hydro object and spot check some post-construction attributes.

Returns

A Noncombustion pointer to a test Hydro object.

```
47 {
48     Noncombustion* test_hydro_ptr =  new Hydro(
49         8760,
50         1,
51         hydro_inputs,
52         time_vec_hrs_ptr
53     );
54 
55     testTruth(
56         not hydro_inputs.noncombustion_inputs.production_inputs.print_flag,
57         __FILE__,
58         __LINE__
59     );
60 
61     testFloatEquals(
62         test_hydro_ptr->n_points,
63         8760,
64         __FILE__,
65         __LINE__
66     );
67 
68     testFloatEquals(
69         test_hydro_ptr->type,
70         NoncombustionType :: HYDRO,
71         __FILE__,
72         __LINE__
73     );
74 
75     testTruth(
76         test_hydro_ptr->type_str == "HYDRO",
77         __FILE__,
78         __LINE__
79     );
80 
81     testFloatEquals(
82         ((Hydro*)test_hydro_ptr)->turbine_type,
83         HydroTurbineType :: HYDRO_TURBINE_PELTON,
84         __FILE__,
85         __LINE__
86     );
87 
88     testFloatEquals(
89         ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
90         10000,
91         __FILE__,
92         __LINE__
93     );
94 
95     return test_hydro_ptr;
96 }   /* testConstruct_Hydro() */
```

5.58.2.4 testEfficiencyInterpolation_Hydro()

```
void testEfficiencyInterpolation_Hydro (
            Noncombustion * test_hydro_ptr )
```

Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.

Parameters

test_hydro_ptr	A Noncombustion pointer to the test Hydro object.

```
115 {
116     std::vector<double> expected_gen_power_ratios = {
117         0,   0.1, 0.2,  0.3, 0.4, 0.5,
118         0.6, 0.7, 0.75, 0.8, 0.9, 1
119     };
120 
121     std::vector<double> expected_gen_efficiencies = {
```

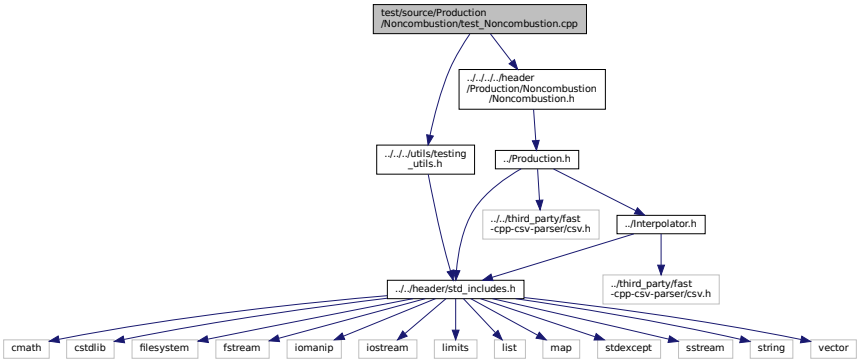
```
122         0.000, 0.800, 0.900, 0.913,
123         0.925, 0.943, 0.947, 0.950,
124         0.953, 0.954, 0.956, 0.958
125     };
126
127     double query = 0;
128     for (size_t i = 0; i < expected_gen_power_ratios.size(); i++) {
129         testFloatEquals(
130             test_hydro_ptr->interpolator.interp_map_1D[
131                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
132             ].x_vec[i],
133             expected_gen_power_ratios[i],
134             __FILE__,
135             __LINE__
136         );
137
138         testFloatEquals(
139             test_hydro_ptr->interpolator.interp_map_1D[
140                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
141             ].y_vec[i],
142             expected_gen_efficiencies[i],
143             __FILE__,
144             __LINE__
145         );
146
147         if (i < expected_gen_power_ratios.size() - 1) {
148             query = expected_gen_power_ratios[i] + ((double)rand() / RAND_MAX) *
149                 (expected_gen_power_ratios[i + 1] - expected_gen_power_ratios[i]);
150
151             test_hydro_ptr->interpolator.interp1D(
152                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
153                 query
154             );
155         }
156     }
157
158     std::vector<double> expected_turb_power_ratios = {
159         0,   0.1, 0.2, 0.3, 0.4,
160         0.5, 0.6, 0.7, 0.8, 0.9,
161         1
162     };
163
164     std::vector<double> expected_turb_efficiencies = {
165         0.000, 0.780, 0.855, 0.875, 0.890,
166         0.900, 0.908, 0.913, 0.918, 0.908,
167         0.880
168     };
169
170     for (size_t i = 0; i < expected_turb_power_ratios.size(); i++) {
171         testFloatEquals(
172             test_hydro_ptr->interpolator.interp_map_1D[
173                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
174             ].x_vec[i],
175             expected_turb_power_ratios[i],
176             __FILE__,
177             __LINE__
178         );
179
180         testFloatEquals(
181             test_hydro_ptr->interpolator.interp_map_1D[
182                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
183             ].y_vec[i],
184             expected_turb_efficiencies[i],
185             __FILE__,
186             __LINE__
187         );
188
189         if (i < expected_turb_power_ratios.size() - 1) {
190             query = expected_turb_power_ratios[i] + ((double)rand() / RAND_MAX) *
191                 (expected_turb_power_ratios[i + 1] - expected_turb_power_ratios[i]);
192
193             test_hydro_ptr->interpolator.interp1D(
194                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
195                 query
196             );
197         }
198     }
199
200     return;
201 }   /* testEfficiencyInterpolation_Hydro() */
```

5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference

Testing suite for Noncombustion class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Noncombustion/Noncombustion.h"
```

Include dependency graph for test_Noncombustion.cpp:



Functions

- Noncombustion * testConstruct_Noncombustion (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Noncombustion object and spot check some post-construction attributes.
- int main (int argc, char **argv)

5.59.1 Detailed Description

Testing suite for Noncombustion class.

A suite of tests for the Noncombustion class.

5.59.2 Function Documentation

5.59.2.1 main()

```
int main (
              int argc,
              char ** argv )
74 {
75     #ifdef _WIN32
76         activateVirtualTerminal();
77     #endif  /* _WIN32 */
78
79     printGold("\tTesting Production <-- Noncombustion");
80
81     srand(time(NULL));
82
83
84     std::vector<double> time_vec_hrs (8760, 0);
85     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
86         time_vec_hrs[i] = i;
87     }
88
89     Noncombustion* test_noncombustion_ptr = testConstruct_Noncombustion(&time_vec_hrs);
90
91
92     try {
93         //...
94     }
95
96
97     catch (...) {
98         delete test_noncombustion_ptr;
99
100         printGold(" ..... ");
101         printRed("FAIL");
102         std::cout << std::endl;
103         throw;
104     }
105
106
107     delete test_noncombustion_ptr;
108
109     printGold(" ..... ");
110     printGreen("PASS");
111     std::cout << std::endl;
112     return 0;
113
114 }   /* main() */
```

5.59.2.2 testConstruct_Noncombustion()

```
Noncombustion * testConstruct_Noncombustion (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Noncombustion object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Noncombustion object.

```
40 {
41     NoncombustionInputs noncombustion_inputs;
42
43     Noncombustion* test_noncombustion_ptr =
44         new Noncombustion(
45             8760,
46             1,
```

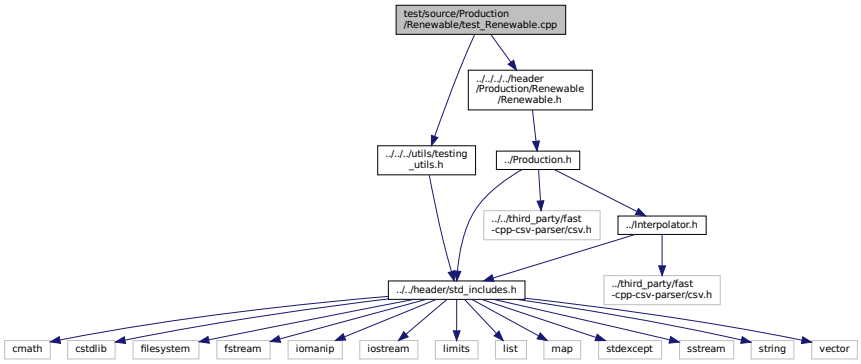
```
47              noncombustion_inputs,
48              time_vec_hrs_ptr
49          );
50
51      testTruth(
52          not noncombustion_inputs.production_inputs.print_flag,
53          __FILE__,
54          __LINE__
55      );
56
57      testFloatEquals(
58          test_noncombustion_ptr->n_points,
59          8760,
60          __FILE__,
61          __LINE__
62      );
63
64      return test_noncombustion_ptr;
65  }   /* testConstruct_Noncombustion() */
```

5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference

Testing suite for Renewable class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Renewable.h"
```

Include dependency graph for test_Renewable.cpp:



Functions

- Renewable * testConstruct_Renewable (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Renewable object and spot check some post-construction attributes.
- int main (int argc, char **argv)

5.60.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

5.60.2 Function Documentation

5.60.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
73 {
74     #ifdef _WIN32
75         activateVirtualTerminal();
76     #endif  /* _WIN32 */
77
78     printGold("\tTesting Production <-- Renewable");
79
80     srand(time(NULL));
81
82
83     std::vector<double> time_vec_hrs (8760, 0);
84     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
85         time_vec_hrs[i] = i;
86     }
87
88     Renewable* test_renewable_ptr = testConstruct_Renewable(&time_vec_hrs);
89
90
91     try {
92         //...
93     }
94
95
96     catch (...) {
97         delete test_renewable_ptr;
98
99         printGold(" .................. ");
100         printRed("FAIL");
101         std::cout << std::endl;
102         throw;
103     }
104
105
106     delete test_renewable_ptr;
107
108     printGold(" .................. ");
109     printGreen("PASS");
110     std::cout << std::endl;
111     return 0;
112
113 }   /* main() */
```

5.60.2.2 testConstruct_Renewable()

```
Renewable * testConstruct_Renewable (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Renewable object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Renewable object.

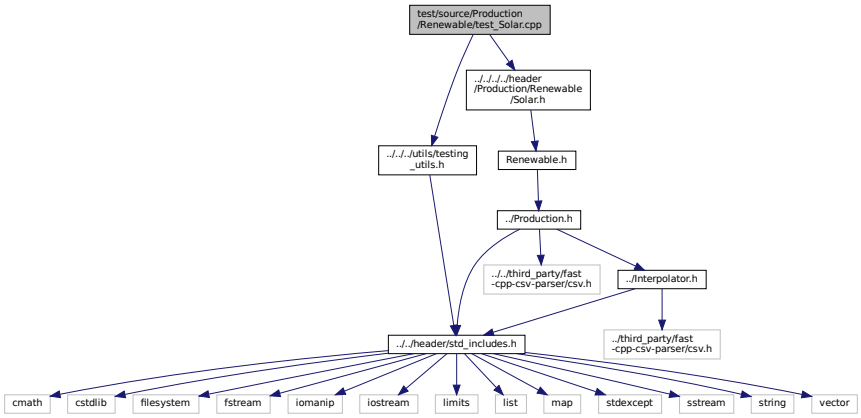
```
40 {
41     RenewableInputs renewable_inputs;
42 
43     Renewable* test_renewable_ptr = new Renewable(
44         8760,
45         1,
46         renewable_inputs,
47         time_vec_hrs_ptr
48     );
49 
50     testTruth(
51         not renewable_inputs.production_inputs.print_flag,
52         __FILE__,
53         __LINE__
54     );
55 
56     testFloatEquals(
57         test_renewable_ptr->n_points,
58         8760,
59         __FILE__,
60         __LINE__
61     );
62 
63     return test_renewable_ptr;
64 }   /* testConstruct_Renewable() */
```

5.61 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for Solar class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Solar.h"
```

Include dependency graph for test_Solar.cpp:



Functions

- Renewable * testConstruct_Solar (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Solar object and spot check some post-construction attributes.
- void testBadConstruct_Solar (std::vector< double > *time_vec_hrs_ptr)

  Function to test the trying to construct a Solar object given bad inputs is being handled as expected.
- void testProductionOverride_Solar (std::string path_2_normalized_production_time_series, std::vector< double > *time_vec_hrs_ptr)

Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.

- void testProductionConstraint_Solar (Renewable *test_solar_ptr)

  Function to test that the production constraint is active and behaving as expected.

- void testCommit_Solar (Renewable *test_solar_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Solar object. Uses a randomized resource input.

- void testEconomics_Solar (Renewable *test_solar_ptr)
- int main (int argc, char **argv)

5.61.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar class.

5.61.2 Function Documentation

5.61.2.1 main()

```
int main (
              int argc,
              char ** argv )
440 {
441     #ifdef _WIN32
442         activateVirtualTerminal();
443     #endif  /* _WIN32 */
444
445     printGold("\tTesting Production <-- Renewable <-- Solar");
446
447     srand(time(NULL));
448
449
450     std::vector<double> time_vec_hrs (8760, 0);
451     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
452         time_vec_hrs[i] = i;
453     }
454
455     Renewable* test_solar_ptr = testConstruct_Solar(&time_vec_hrs);
456
457
458     try {
459         testBadConstruct_Solar(&time_vec_hrs);
460
461         std::string path_2_normalized_production_time_series =
462             "data/test/normalized_production/normalized_solar_production.csv";
463
464         testProductionOverride_Solar(
465             path_2_normalized_production_time_series,
466             &time_vec_hrs
467         );
468
469         testProductionConstraint_Solar(test_solar_ptr);
470
471         testCommit_Solar(test_solar_ptr);
472         testEconomics_Solar(test_solar_ptr);
473     }
474
475
476     catch (...) {
477         delete test_solar_ptr;
478
479         printGold(" ....... ");
480         printRed("FAIL");
```

```
481         std::cout « std::endl;
482         throw;
483     }
484 
485 
486     delete test_solar_ptr;
487 
488     printGold(" ....... ");
489     printGreen("PASS");
490     std::cout « std::endl;
491     return 0;
492 
493 }   /* main() */
```

5.61.2.2 testBadConstruct_Solar()

```
void testBadConstruct_Solar (
             std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Solar object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
109 {
110     bool error_flag = true;
111 
112     try {
113         SolarInputs bad_solar_inputs;
114         bad_solar_inputs.derating = -1;
115 
116         Solar bad_solar(8760, 1, bad_solar_inputs, time_vec_hrs_ptr);
117 
118         error_flag = false;
119     } catch (...) {
120         // Task failed successfully! =P
121     }
122     if (not error_flag) {
123         expectedErrorNotDetected(__FILE__, __LINE__);
124     }
125 
126     return;
127 }   /* testBadConstruct_Solar() */
```

5.61.2.3 testCommit_Solar()

```
void testCommit_Solar (
             Renewable * test_solar_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Solar object. Uses a randomized resource input.

Parameters

test_solar_ptr	A Renewable pointer to the test Solar object.

```
289 {
290     std::vector<double> dt_vec_hrs (48, 1);
291 
292     std::vector<double> load_vec_kW = {
```

```
293         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
294         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
295         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
296         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
297     };
298
299     double load_kW = 0;
300     double production_kW = 0;
301     double roll = 0;
302     double solar_resource_kWm2 = 0;
303
304     for (int i = 0; i < 48; i++) {
305         roll = (double)rand() / RAND_MAX;
306
307         solar_resource_kWm2 = roll;
308
309         roll = (double)rand() / RAND_MAX;
310
311         if (roll <= 0.1) {
312             solar_resource_kWm2 = 0;
313         }
314
315         else if (roll >= 0.95) {
316             solar_resource_kWm2 = 1.25;
317         }
318
319         roll = (double)rand() / RAND_MAX;
320
321         if (roll >= 0.95) {
322             roll = 1.25;
323         }
324
325         load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
326         load_kW = load_vec_kW[i];
327
328         production_kW = test_solar_ptr->computeProductionkW(
329             i,
330             dt_vec_hrs[i],
331             solar_resource_kWm2
332         );
333
334         load_kW = test_solar_ptr->commit(
335             i,
336             dt_vec_hrs[i],
337             production_kW,
338             load_kW
339         );
340
341         // is running (or not) as expected
342         if (solar_resource_kWm2 > 0) {
343             testTruth(
344                 test_solar_ptr->is_running,
345                 __FILE__,
346                 __LINE__
347             );
348         }
349
350         else {
351             testTruth(
352                 not test_solar_ptr->is_running,
353                 __FILE__,
354                 __LINE__
355             );
356         }
357
358         // load_kW <= load_vec_kW (i.e., after vs before)
359         testLessThanOrEqualTo(
360             load_kW,
361             load_vec_kW[i],
362             __FILE__,
363             __LINE__
364         );
365
366         // production = dispatch + storage + curtailment
367         testFloatEquals(
368             test_solar_ptr->production_vec_kW[i] -
369             test_solar_ptr->dispatch_vec_kW[i] -
370             test_solar_ptr->storage_vec_kW[i] -
371             test_solar_ptr->curtailment_vec_kW[i],
372             0,
373             __FILE__,
374             __LINE__
375         );
376
377         // capacity constraint
378         if (solar_resource_kWm2 > 1) {
379             testFloatEquals(
```

```
380                 test_solar_ptr->production_vec_kW[i],
381                 test_solar_ptr->capacity_kW,
382                 __FILE__,
383                 __LINE__
384             );
385         }
386     }
387 
388     return;
389 }   /* testCommit_Solar() */
```

5.61.2.4 testConstruct_Solar()

```
Renewable * testConstruct_Solar (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Solar object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Solar object.

```
40 {
41     SolarInputs solar_inputs;
42 
43     Renewable* test_solar_ptr = new Solar(
44         8760,
45         1,
46         solar_inputs,
47         time_vec_hrs_ptr
48     );
49 
50     testTruth(
51         not solar_inputs.renewable_inputs.production_inputs.print_flag,
52         __FILE__,
53         __LINE__
54     );
55 
56     testFloatEquals(
57         test_solar_ptr->n_points,
58         8760,
59         __FILE__,
60         __LINE__
61     );
62 
63     testFloatEquals(
64         test_solar_ptr->type,
65         RenewableType :: SOLAR,
66         __FILE__,
67         __LINE__
68     );
69 
70     testTruth(
71         test_solar_ptr->type_str == "SOLAR",
72         __FILE__,
73         __LINE__
74     );
75 
76     testFloatEquals(
77         test_solar_ptr->capital_cost,
78         350118.723363,
79         __FILE__,
80         __LINE__
81     );
82 
83     testFloatEquals(
84         test_solar_ptr->operation_maintenance_cost_kWh,
85         0.01,
```

```
86          __FILE__,
87          __LINE__
88      );
89
90      return test_solar_ptr;
91  }   /* testConstruct_Solar() */
```

5.61.2.5 testEconomics_Solar()

```
void testEconomics_Solar (
              Renewable * test_solar_ptr )
407 {
408     for (int i = 0; i < 48; i++) {
409         // resource, O&M > 0 whenever solar is running (i.e., producing)
410         if (test_solar_ptr->is_running_vec[i]) {
411             testGreaterThan(
412                 test_solar_ptr->operation_maintenance_cost_vec[i],
413                 0,
414                 __FILE__,
415                 __LINE__
416             );
417         }
418
419         // resource, O&M = 0 whenever solar is not running (i.e., not producing)
420         else {
421             testFloatEquals(
422                 test_solar_ptr->operation_maintenance_cost_vec[i],
423                 0,
424                 __FILE__,
425                 __LINE__
426             );
427         }
428     }
429
430     return;
431 }   /* testEconomics_Solar() */
```

5.61.2.6 testProductionConstraint_Solar()

```
void testProductionConstraint_Solar (
              Renewable * test_solar_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_solar_ptr	A Renewable pointer to the test Solar object.

```
254 {
255     testFloatEquals(
256         test_solar_ptr->computeProductionkW(0, 1, 2),
257         100,
258         __FILE__,
259         __LINE__
260     );
261
262     testFloatEquals(
263         test_solar_ptr->computeProductionkW(0, 1, -1),
264         0,
265         __FILE__,
266         __LINE__
267     );
268
269     return;
270 }   /* testProductionConstraint_Solar() */
```

5.61.2.7 testProductionOverride_Solar()

```
void testProductionOverride_Solar (
            std::string path_2_normalized_production_time_series,
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.

Parameters

path_2_normalized_production_time_series	A path (either relative or absolute) to the given normalized production time series data.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
154 {
155     SolarInputs solar_inputs;
156 
157     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
158         path_2_normalized_production_time_series;
159 
160     Solar test_solar_override(
161         time_vec_hrs_ptr->size(),
162         1,
163         solar_inputs,
164         time_vec_hrs_ptr
165     );
166 
167 
168     std::vector<double> expected_normalized_production_vec = {
169         0.916955708517556,
170         0.90947506148393,
171         0.38425267564517,
172         0.191510884037643,
173         0.803361391862077,
174         0.261511294927198,
175         0.221944653883198,
176         0.858495335855501,
177         0.0162863861443092,
178         0.774345409915512,
179         0.354898664149867,
180         0.11158009453439,
181         0.191670176408956,
182         0.0149072402795702,
183         0.30174228469322,
184         0.0815062957850151,
185         0.776404660266821,
186         0.207069187162109,
187         0.518926216750454,
188         0.148538109788597,
189         0.443035200791027,
190         0.62119079547209,
191         0.270792717524391,
192         0.761074879460849,
193         0.0545251308358993,
194         0.0895417089500092,
195         0.21787190761933,
196         0.834403724509682,
197         0.908807953036246,
198         0.815888965292123,
199         0.416663215314571,
200         0.523649705576525,
201         0.490890480401437,
202         0.28317138282312,
203         0.877382682055847,
204         0.14972090597986,
205         0.480161632646382,
206         0.0655830129932816,
207         0.41802666403448,
208         0.48692477737368,
209         0.275957323208066,
210         0.228651250718341,
211         0.574371311550247,
212         0.251872481275769,
```

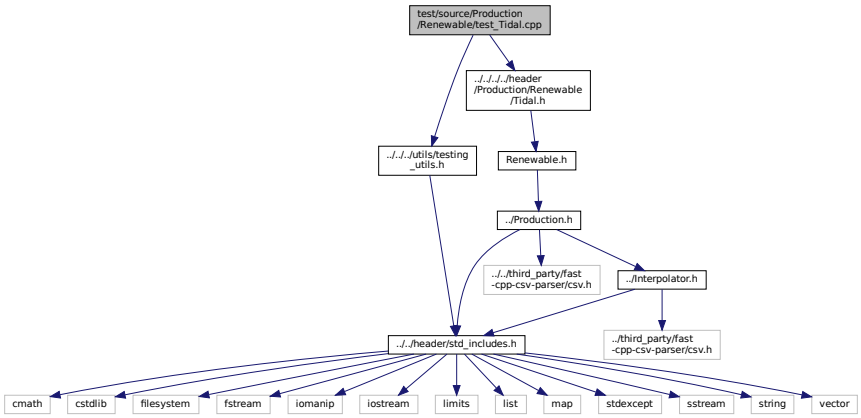
```
213         0.802697508767121,
214         0.00130607304363551,
215         0.481240172488057,
216         0.702527508293784
217     };
218
219     for (size_t i = 0; i < expected_normalized_production_vec.size(); i++) {
220         testFloatEquals(
221             test_solar_override.normalized_production_vec[i],
222             expected_normalized_production_vec[i],
223             __FILE__,
224             __LINE__
225         );
226
227         testFloatEquals(
228             test_solar_override.computeProductionkW(i, rand(), rand()),
229             test_solar_override.capacity_kW * expected_normalized_production_vec[i],
230             __FILE__,
231             __LINE__
232         );
233     }
234
235     return;
236 }   /* testProductionOverride_Solar() */
```

5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference

Testing suite for Tidal class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Tidal.h"
```

Include dependency graph for test_Tidal.cpp:



Functions

- Renewable * testConstruct_Tidal (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Tidal object and spot check some post-construction attributes.
- void testBadConstruct_Tidal (std::vector< double > *time_vec_hrs_ptr)

  Function to test the trying to construct a Tidal object given bad inputs is being handled as expected.
- void testProductionConstraint_Tidal (Renewable *test_tidal_ptr)

  Function to test that the production constraint is active and behaving as expected.
- void testCommit_Tidal (Renewable *test_tidal_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Tidal object. Uses a randomized resource input.
- void testEconomics_Tidal (Renewable *test_tidal_ptr)
- int main (int argc, char **argv)

5.62.1 Detailed Description

Testing suite for Tidal class.

A suite of tests for the Tidal class.

5.62.2 Function Documentation

5.62.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
327 {
328     #ifdef _WIN32
329         activateVirtualTerminal();
330     #endif  /* _WIN32 */
331
332     printGold("\tTesting Production <-- Renewable <-- Tidal");
333
334     srand(time(NULL));
335
336
337     std::vector<double> time_vec_hrs (8760, 0);
338     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
339         time_vec_hrs[i] = i;
340     }
341
342     Renewable* test_tidal_ptr = testConstruct_Tidal(&time_vec_hrs);
343
344
345     try {
346         testBadConstruct_Tidal(&time_vec_hrs);
347
348         testProductionConstraint_Tidal(test_tidal_ptr);
349
350         testCommit_Tidal(test_tidal_ptr);
351         testEconomics_Tidal(test_tidal_ptr);
352     }
353
354
355     catch (...) {
356         delete test_tidal_ptr;
357
358         printGold(" ....... ");
359         printRed("FAIL");
360         std::cout « std::endl;
361         throw;
362     }
363
364
365     delete test_tidal_ptr;
366
367     printGold(" ....... ");
368     printGreen("PASS");
369     std::cout « std::endl;
370     return 0;
371
372 }   /* main() */
```

5.62.2.2 testBadConstruct_Tidal()

```
void testBadConstruct_Tidal (
              std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Tidal object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
104 {
105     bool error_flag = true;
106 
107     try {
108         TidalInputs bad_tidal_inputs;
109         bad_tidal_inputs.design_speed_ms = -1;
110 
111         Tidal bad_tidal(8760, 1, bad_tidal_inputs, time_vec_hrs_ptr);
112 
113         error_flag = false;
114     } catch (...) {
115         // Task failed successfully! =P
116     }
117     if (not error_flag) {
118         expectedErrorNotDetected(__FILE__, __LINE__);
119     }
120 
121     return;
122 }   /* testBadConstruct_Tidal() */
```

5.62.2.3 testCommit_Tidal()

```
void testCommit_Tidal (
             Renewable * test_tidal_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Tidal object. Uses a randomized resource input.

Parameters

test_tidal_ptr	A Renewable pointer to the test Tidal object.

```
186 {
187     std::vector<double> dt_vec_hrs (48, 1);
188 
189     std::vector<double> load_vec_kW = {
190         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
191         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
192         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
193         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
194     };
195 
196     double load_kW = 0;
197     double production_kW = 0;
198     double roll = 0;
199     double tidal_resource_ms = 0;
200 
201     for (int i = 0; i < 48; i++) {
202         roll = (double)rand() / RAND_MAX;
203 
204         tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
205 
206         roll = (double)rand() / RAND_MAX;
207 
208         if (roll <= 0.1) {
209             tidal_resource_ms = 0;
210         }
211 
212         else if (roll >= 0.95) {
213             tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
214         }
215 
216         roll = (double)rand() / RAND_MAX;
217 
218         if (roll >= 0.95) {
219             roll = 1.25;
220         }
221 
```

```
222         load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
223         load_kW = load_vec_kW[i];
224 
225         production_kW = test_tidal_ptr->computeProductionkW(
226             i,
227             dt_vec_hrs[i],
228             tidal_resource_ms
229         );
230 
231         load_kW = test_tidal_ptr->commit(
232             i,
233             dt_vec_hrs[i],
234             production_kW,
235             load_kW
236         );
237 
238         // is running (or not) as expected
239         if (production_kW > 0) {
240             testTruth(
241                 test_tidal_ptr->is_running,
242                 __FILE__,
243                 __LINE__
244             );
245         }
246 
247         else {
248             testTruth(
249                 not test_tidal_ptr->is_running,
250                 __FILE__,
251                 __LINE__
252             );
253         }
254 
255         // load_kW <= load_vec_kW (i.e., after vs before)
256         testLessThanOrEqualTo(
257             load_kW,
258             load_vec_kW[i],
259             __FILE__,
260             __LINE__
261         );
262 
263         // production = dispatch + storage + curtailment
264         testFloatEquals(
265             test_tidal_ptr->production_vec_kW[i] -
266             test_tidal_ptr->dispatch_vec_kW[i] -
267             test_tidal_ptr->storage_vec_kW[i] -
268             test_tidal_ptr->curtailment_vec_kW[i],
269             0,
270             __FILE__,
271             __LINE__
272         );
273     }
274 
275     return;
276 }   /* testCommit_Tidal() */
```

5.62.2.4 testConstruct_Tidal()

```
Renewable * testConstruct_Tidal (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Tidal object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Tidal object.

```
40 {
```

```
41      TidalInputs tidal_inputs;
42
43      Renewable* test_tidal_ptr = new Tidal(8760, 1, tidal_inputs, time_vec_hrs_ptr);
44
45      testTruth(
46          not tidal_inputs.renewable_inputs.production_inputs.print_flag,
47          __FILE__,
48          __LINE__
49      );
50
51      testFloatEquals(
52          test_tidal_ptr->n_points,
53          8760,
54          __FILE__,
55          __LINE__
56      );
57
58      testFloatEquals(
59          test_tidal_ptr->type,
60          RenewableType :: TIDAL,
61          __FILE__,
62          __LINE__
63      );
64
65      testTruth(
66          test_tidal_ptr->type_str == "TIDAL",
67          __FILE__,
68          __LINE__
69      );
70
71      testFloatEquals(
72          test_tidal_ptr->capital_cost,
73          500237.446725,
74          __FILE__,
75          __LINE__
76      );
77
78      testFloatEquals(
79          test_tidal_ptr->operation_maintenance_cost_kWh,
80          0.069905,
81          __FILE__,
82          __LINE__
83      );
84
85      return test_tidal_ptr;
86  }   /* testConstruct_Tidal() */
```

5.62.2.5 testEconomics_Tidal()

```
void testEconomics_Tidal (
              Renewable * test_tidal_ptr )
294 {
295      for (int i = 0; i < 48; i++) {
296          // resource, O&M > 0 whenever tidal is running (i.e., producing)
297          if (test_tidal_ptr->is_running_vec[i]) {
298              testGreaterThan(
299                  test_tidal_ptr->operation_maintenance_cost_vec[i],
300                  0,
301                  __FILE__,
302                  __LINE__
303              );
304          }
305
306          // resource, O&M = 0 whenever tidal is not running (i.e., not producing)
307          else {
308              testFloatEquals(
309                  test_tidal_ptr->operation_maintenance_cost_vec[i],
310                  0,
311                  __FILE__,
312                  __LINE__
313              );
314          }
315      }
316
317      return;
318  }   /* testEconomics_Tidal() */
```

5.62.2.6 testProductionConstraint_Tidal()

```
void testProductionConstraint_Tidal (
            Renewable * test_tidal_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_tidal_ptr	A Renewable pointer to the test Tidal object.

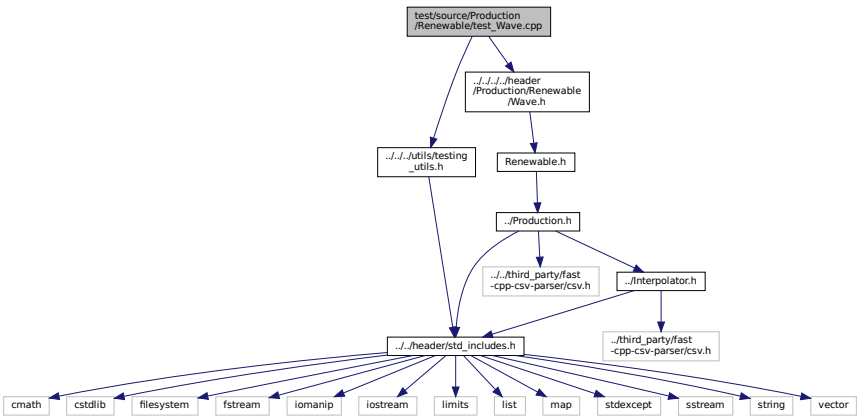
```
140 {
141     testFloatEquals(
142         test_tidal_ptr->computeProductionkW(0, 1, 1e6),
143         0,
144         __FILE__,
145         __LINE__
146     );
147
148     testFloatEquals(
149         test_tidal_ptr->computeProductionkW(
150             0,
151             1,
152             ((Tidal*)test_tidal_ptr)->design_speed_ms
153         ),
154         test_tidal_ptr->capacity_kW,
155         __FILE__,
156         __LINE__
157     );
158
159     testFloatEquals(
160         test_tidal_ptr->computeProductionkW(0, 1, -1),
161         0,
162         __FILE__,
163         __LINE__
164     );
165
166     return;
167 }   /* testProductionConstraint_Tidal() */
```

5.63 test/source/Production/Renewable/test_Wave.cpp File Reference

Testing suite for Wave class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Wave.h"
```

Include dependency graph for test_Wave.cpp:



Functions

- Renewable * testConstruct_Wave (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Wave object and spot check some post-construction attributes.
- Renewable * testConstructLookup_Wave (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Wave object using production lookup.
- void testBadConstruct_Wave (std::vector< double > *time_vec_hrs_ptr)

  Function to test the trying to construct a Wave object given bad inputs is being handled as expected.
- void testProductionConstraint_Wave (Renewable *test_wave_ptr)

  Function to test that the production constraint is active and behaving as expected.
- void testCommit_Wave (Renewable *test_wave_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wave object. Uses a randomized resource input.
- void testEconomics_Wave (Renewable *test_wave_ptr)
- void testProductionLookup_Wave (Renewable *test_wave_lookup_ptr)

  Function to test that production lookup (i.e., interpolation) is returning the expected values.
- int main (int argc, char **argv)

5.63.1 Detailed Description

Testing suite for Wave class.

A suite of tests for the Wave class.

5.63.2 Function Documentation

5.63.2.1 main()

```
int main (
              int argc,
              char ** argv )
442 {
443     #ifdef _WIN32
444         activateVirtualTerminal();
445     #endif  /* _WIN32 */
446 
447     printGold("\tTesting Production <-- Renewable <-- Wave");
448 
449     srand(time(NULL));
450 
451 
452     std::vector<double> time_vec_hrs (8760, 0);
453     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
454         time_vec_hrs[i] = i;
455     }
456 
457     Renewable* test_wave_ptr = testConstruct_Wave(&time_vec_hrs);
458     Renewable* test_wave_lookup_ptr = testConstructLookup_Wave(&time_vec_hrs);
459 
460 
461     try {
462         testBadConstruct_Wave(&time_vec_hrs);
463 
464         testProductionConstraint_Wave(test_wave_ptr);
465 
466         testCommit_Wave(test_wave_ptr);
467         testEconomics_Wave(test_wave_ptr);
468 
```

```
469         testProductionLookup_Wave(test_wave_lookup_ptr);
470     }
471
472
473     catch (...) {
474         delete test_wave_ptr;
475         delete test_wave_lookup_ptr;
476
477         printGold(" ........ ");
478         printRed("FAIL");
479         std::cout << std::endl;
480         throw;
481     }
482
483
484     delete test_wave_ptr;
485     delete test_wave_lookup_ptr;
486
487     printGold(" ........ ");
488     printGreen("PASS");
489     std::cout << std::endl;
490     return 0;
491
492 }   /* main() */
```

5.63.2.2 testBadConstruct_Wave()

```
void testBadConstruct_Wave (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Wave object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
133 {
134     bool error_flag = true;
135
136     try {
137         WaveInputs bad_wave_inputs;
138         bad_wave_inputs.design_significant_wave_height_m = -1;
139
140         Wave bad_wave(8760, 1, bad_wave_inputs, time_vec_hrs_ptr);
141
142         error_flag = false;
143     } catch (...) {
144         // Task failed successfully! =P
145     }
146     if (not error_flag) {
147         expectedErrorNotDetected(__FILE__, __LINE__);
148     }
149
150     return;
151 }   /* testBadConstruct_Wave() */
```

5.63.2.3 testCommit_Wave()

```
void testCommit_Wave (
            Renewable * test_wave_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wave object. Uses a randomized resource input.

Parameters

test_wave_ptr	A Renewable pointer to the test Wave object.

```
204 {
205     std::vector<double> dt_vec_hrs (48, 1);
206 
207     std::vector<double> load_vec_kW = {
208         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
209         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
210         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
211         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
212     };
213 
214     double load_kW = 0;
215     double production_kW = 0;
216     double roll = 0;
217     double significant_wave_height_m = 0;
218     double energy_period_s = 0;
219 
220     for (int i = 0; i < 48; i++) {
221         roll = (double)rand() / RAND_MAX;
222 
223         if (roll <= 0.05) {
224             roll = 0;
225         }
226 
227         significant_wave_height_m = roll *
228             ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
229 
230         roll = (double)rand() / RAND_MAX;
231 
232         if (roll <= 0.05) {
233             roll = 0;
234         }
235 
236         energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
237 
238         roll = (double)rand() / RAND_MAX;
239 
240         if (roll >= 0.95) {
241             roll = 1.25;
242         }
243 
244         load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
245         load_kW = load_vec_kW[i];
246 
247         production_kW = test_wave_ptr->computeProductionkW(
248             i,
249             dt_vec_hrs[i],
250             significant_wave_height_m,
251             energy_period_s
252         );
253 
254         load_kW = test_wave_ptr->commit(
255             i,
256             dt_vec_hrs[i],
257             production_kW,
258             load_kW
259         );
260 
261         // is running (or not) as expected
262         if (production_kW > 0) {
263             testTruth(
264                 test_wave_ptr->is_running,
265                 __FILE__,
266                 __LINE__
267             );
268         }
269 
270         else {
271             testTruth(
272                 not test_wave_ptr->is_running,
273                 __FILE__,
274                 __LINE__
275             );
276         }
277 
278         // load_kW <= load_vec_kW (i.e., after vs before)
279         testLessThanOrEqualTo(
280             load_kW,
281             load_vec_kW[i],
282             __FILE__,
283             __LINE__
284         );
285 
```

```
286         // production = dispatch + storage + curtailment
287         testFloatEquals(
288             test_wave_ptr->production_vec_kW[i] -
289             test_wave_ptr->dispatch_vec_kW[i] -
290             test_wave_ptr->storage_vec_kW[i] -
291             test_wave_ptr->curtailment_vec_kW[i],
292             0,
293             __FILE__,
294             __LINE__
295         );
296     }
297 
298     return;
299 }   /* testCommit_Wave() */
```

5.63.2.4 testConstruct_Wave()

```
Renewable * testConstruct_Wave (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Wave object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wave object.

```
40 {
41     WaveInputs wave_inputs;
42 
43     Renewable* test_wave_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);
44 
45     testTruth(
46         not wave_inputs.renewable_inputs.production_inputs.print_flag,
47         __FILE__,
48         __LINE__
49     );
50 
51     testFloatEquals(
52         test_wave_ptr->n_points,
53         8760,
54         __FILE__,
55         __LINE__
56     );
57 
58     testFloatEquals(
59         test_wave_ptr->type,
60         RenewableType :: WAVE,
61         __FILE__,
62         __LINE__
63     );
64 
65     testTruth(
66         test_wave_ptr->type_str == "WAVE",
67         __FILE__,
68         __LINE__
69     );
70 
71     testFloatEquals(
72         test_wave_ptr->capital_cost,
73         850831.063539,
74         __FILE__,
75         __LINE__
76     );
77 
78     testFloatEquals(
79         test_wave_ptr->operation_maintenance_cost_kWh,
80         0.069905,
81         __FILE__,
```

```
82          __LINE__
83      );
84
85      return test_wave_ptr;
86  }   /* testConstruct_Wave() */
```

5.63.2.5 testConstructLookup_Wave()

```
Renewable * testConstructLookup_Wave (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Wave object using production lookup.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wave object.

```
105 {
106     WaveInputs wave_inputs;
107
108     wave_inputs.power_model = WavePowerProductionModel :: WAVE_POWER_LOOKUP;
109     wave_inputs.path_2_normalized_performance_matrix =
110         "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
111
112     Renewable* test_wave_lookup_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);
113
114     return test_wave_lookup_ptr;
115 }   /* testConstructLookup_Wave() */
```

5.63.2.6 testEconomics_Wave()

```
void testEconomics_Wave (
              Renewable * test_wave_ptr )
317 {
318     for (int i = 0; i < 48; i++) {
319         // resource, O&M > 0 whenever wave is running (i.e., producing)
320         if (test_wave_ptr->is_running_vec[i]) {
321             testGreaterThan(
322                 test_wave_ptr->operation_maintenance_cost_vec[i],
323                 0,
324                 __FILE__,
325                 __LINE__
326             );
327         }
328
329         // resource, O&M = 0 whenever wave is not running (i.e., not producing)
330         else {
331             testFloatEquals(
332                 test_wave_ptr->operation_maintenance_cost_vec[i],
333                 0,
334                 __FILE__,
335                 __LINE__
336             );
337         }
338     }
339
340     return;
341 }   /* testEconomics_Wave() */
```

5.63.2.7 testProductionConstraint_Wave()

```
void testProductionConstraint_Wave (
            Renewable * test_wave_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_wave_ptr	A Renewable pointer to the test Wave object.

```
169 {
170     testFloatEquals(
171         test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
172         0,
173         __FILE__,
174         __LINE__
175     );
176
177     testFloatEquals(
178         test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
179         0,
180         __FILE__,
181         __LINE__
182     );
183
184     return;
185 }   /* testProductionConstraint_Wave() */
```

5.63.2.8 testProductionLookup_Wave()

```
void testProductionLookup_Wave (
            Renewable * test_wave_lookup_ptr )
```

Function to test that production lookup (i.e., interpolation) is returning the expected values.

Parameters

test_wave_lookup_ptr	A Renewable pointer to the test Wave object using production lookup.

```
360 {
361     std::vector<double> significant_wave_height_vec_m = {
362         0.389211848822208,
363         0.836477431896843,
364         1.52738334015579,
365         1.92640601114508,
366         2.27297317532019,
367         2.87416589636605,
368         3.72275770908175,
369         3.95063175885536,
370         4.68097139867404,
371         4.97775020449812,
372         5.55184219980547,
373         6.06566629451658,
374         6.27927876785062,
375         6.96218133671013,
376         7.51754442460228
377     };
378
379     std::vector<double> energy_period_vec_s = {
380         5.45741899698926,
381         6.00101329139007,
382         7.50567689404182,
383         8.77681262912881,
384         9.45143678206774,
385         10.7767876462885,
386         11.4795760857165,
387         12.9430684577599,
```

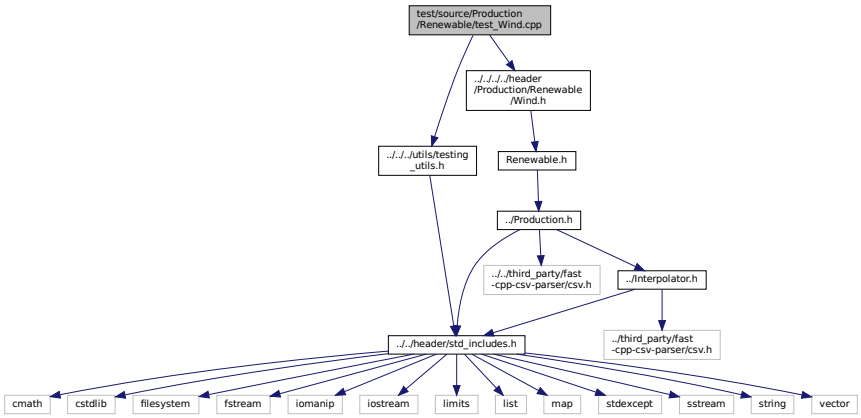
```
388         13.303544885703,
389         14.5069863517863,
390         15.1487890438045,
391         16.086524049077,
392         17.176609978648,
393         18.4155153740256,
394         19.1704554940162
395     };
396
397     std::vector<std::vector<double> expected_normalized_performance_matrix = {
398
        {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
399
        {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
400
        {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
401
        {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
402
        {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
403
        {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
404
        {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
405
        {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
406
        {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
407
        {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
408
        {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
409
        {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
410
        {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
411
        {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.510
412
        {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.484
413     };
414
415     for (size_t i = 0; i < energy_period_vec_s.size(); i++) {
416         for (size_t j = 0; j < significant_wave_height_vec_m.size(); j++) {
417             testFloatEquals(
418                 test_wave_lookup_ptr->computeProductionkW(
419                     0,
420                     1,
421                     significant_wave_height_vec_m[j],
422                     energy_period_vec_s[i]
423                 ),
424                 expected_normalized_performance_matrix[i][j] *
425                 test_wave_lookup_ptr->capacity_kW,
426                 __FILE__,
427                 __LINE__
428             );
429         }
430     }
431
432     return;
433 }   /* testProductionLookup_Wave() */
```

5.64 test/source/Production/Renewable/test_Wind.cpp File Reference

Testing suite for Wind class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Wind.h"
```

Include dependency graph for test_Wind.cpp:



Functions

- Renewable ∗ testConstruct_Wind (std::vector< double > ∗time_vec_hrs_ptr)

  A function to construct a Wind object and spot check some post-construction attributes.
- void testBadConstruct_Wind (std::vector< double > ∗time_vec_hrs_ptr)

  Function to test the trying to construct a Wind object given bad inputs is being handled as expected.
- void testProductionConstraint_Wind (Renewable ∗test_wind_ptr)

  Function to test that the production constraint is active and behaving as expected.
- void testCommit_Wind (Renewable ∗test_wind_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wind object. Uses a randomized resource input.
- void testEconomics_Wind (Renewable ∗test_wind_ptr)
- int main (int argc, char ∗∗argv)

5.64.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind class.

5.64.2 Function Documentation

5.64.2.1 main()

```
int main (
              int argc,
              char ** argv )
327 {
328     #ifdef _WIN32
329         activateVirtualTerminal();
330     #endif  /* _WIN32 */
331
332     printGold("\tTesting Production <-- Renewable <-- Wind");
333
334     srand(time(NULL));
335
336
337     std::vector<double> time_vec_hrs (8760, 0);
338     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
339         time_vec_hrs[i] = i;
340     }
341
342     Renewable* test_wind_ptr = testConstruct_Wind(&time_vec_hrs);
343
344
345     try {
346         testBadConstruct_Wind(&time_vec_hrs);
347
348         testProductionConstraint_Wind(test_wind_ptr);
349
350         testCommit_Wind(test_wind_ptr);
351         testEconomics_Wind(test_wind_ptr);
352     }
353
354
355     catch (...) {
356         delete test_wind_ptr;
357
358         printGold(" ........ ");
359         printRed("FAIL");
360         std::cout « std::endl;
361         throw;
362     }
363
364
365     delete test_wind_ptr;
366
367     printGold(" ........ ");
368     printGreen("PASS");
369     std::cout « std::endl;
370     return 0;
371
372 }   /* main() */
```

5.64.2.2 testBadConstruct_Wind()

```
void testBadConstruct_Wind (
              std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Wind object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
104 {
105     bool error_flag = true;
106
107     try {
108         WindInputs bad_wind_inputs;
109         bad_wind_inputs.design_speed_ms = -1;
110
111         Wind bad_wind(8760, 1, bad_wind_inputs, time_vec_hrs_ptr);
112
```

```
113         error_flag = false;
114     } catch (...) {
115         // Task failed successfully! =P
116     }
117     if (not error_flag) {
118         expectedErrorNotDetected(__FILE__, __LINE__);
119     }
120 
121     return;
122 }   /* testBadConstruct_Wind() */
```

5.64.2.3 testCommit_Wind()

```
void testCommit_Wind (
            Renewable * test_wind_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wind object. Uses a randomized resource input.

Parameters

test_wind_ptr	A Renewable pointer to the test Wind object.

```
186 {
187     std::vector<double> dt_vec_hrs (48, 1);
188 
189     std::vector<double> load_vec_kW = {
190         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
191         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
192         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
193         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
194     };
195 
196     double load_kW = 0;
197     double production_kW = 0;
198     double roll = 0;
199     double wind_resource_ms = 0;
200 
201     for (int i = 0; i < 48; i++) {
202         roll = (double)rand() / RAND_MAX;
203 
204         wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
205 
206         roll = (double)rand() / RAND_MAX;
207 
208         if (roll <= 0.1) {
209             wind_resource_ms = 0;
210         }
211 
212         else if (roll >= 0.95) {
213             wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
214         }
215 
216         roll = (double)rand() / RAND_MAX;
217 
218         if (roll >= 0.95) {
219             roll = 1.25;
220         }
221 
222         load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
223         load_kW = load_vec_kW[i];
224 
225         production_kW = test_wind_ptr->computeProductionkW(
226             i,
227             dt_vec_hrs[i],
228             wind_resource_ms
229         );
230 
231         load_kW = test_wind_ptr->commit(
232             i,
233             dt_vec_hrs[i],
234             production_kW,
235             load_kW
236         );
```

```
237
238         // is running (or not) as expected
239         if (production_kW > 0) {
240             testTruth(
241                 test_wind_ptr->is_running,
242                 __FILE__,
243                 __LINE__
244             );
245         }
246
247         else {
248             testTruth(
249                 not test_wind_ptr->is_running,
250                 __FILE__,
251                 __LINE__
252             );
253         }
254
255         // load_kW <= load_vec_kW (i.e., after vs before)
256         testLessThanOrEqualTo(
257             load_kW,
258             load_vec_kW[i],
259             __FILE__,
260             __LINE__
261         );
262
263         // production = dispatch + storage + curtailment
264         testFloatEquals(
265             test_wind_ptr->production_vec_kW[i] -
266             test_wind_ptr->dispatch_vec_kW[i] -
267             test_wind_ptr->storage_vec_kW[i] -
268             test_wind_ptr->curtailment_vec_kW[i],
269             0,
270             __FILE__,
271             __LINE__
272         );
273     }
274
275     return;
276 }   /* testCommit_Wind() */
```

5.64.2.4 testConstruct_Wind()

```
Renewable * testConstruct_Wind (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Wind object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wind object.

```
40 {
41     WindInputs wind_inputs;
42
43     Renewable* test_wind_ptr = new Wind(8760, 1, wind_inputs, time_vec_hrs_ptr);
44
45     testTruth(
46         not wind_inputs.renewable_inputs.production_inputs.print_flag,
47         __FILE__,
48         __LINE__
49     );
50
51     testFloatEquals(
52         test_wind_ptr->n_points,
53         8760,
54         __FILE__,
55         __LINE__
```

```
56      );
57
58      testFloatEquals(
59          test_wind_ptr->type,
60          RenewableType :: WIND,
61          __FILE__,
62          __LINE__
63      );
64
65      testTruth(
66          test_wind_ptr->type_str == "WIND",
67          __FILE__,
68          __LINE__
69      );
70
71      testFloatEquals(
72          test_wind_ptr->capital_cost,
73          450356.170088,
74          __FILE__,
75          __LINE__
76      );
77
78      testFloatEquals(
79          test_wind_ptr->operation_maintenance_cost_kWh,
80          0.034953,
81          __FILE__,
82          __LINE__
83      );
84
85      return test_wind_ptr;
86  }   /* testConstruct_Wind() */
```

5.64.2.5 testEconomics_Wind()

```
void testEconomics_Wind (
              Renewable * test_wind_ptr )
294 {
295     for (int i = 0; i < 48; i++) {
296         // resource, O&M > 0 whenever wind is running (i.e., producing)
297         if (test_wind_ptr->is_running_vec[i]) {
298             testGreaterThan(
299                 test_wind_ptr->operation_maintenance_cost_vec[i],
300                 0,
301                 __FILE__,
302                 __LINE__
303             );
304         }
305
306         // resource, O&M = 0 whenever wind is not running (i.e., not producing)
307         else {
308             testFloatEquals(
309                 test_wind_ptr->operation_maintenance_cost_vec[i],
310                 0,
311                 __FILE__,
312                 __LINE__
313             );
314         }
315     }
316
317     return;
318 }   /* testEconomics_Wind() */
```

5.64.2.6 testProductionConstraint_Wind()

```
void testProductionConstraint_Wind (
              Renewable * test_wind_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_wind_ptr	A Renewable pointer to the test Wind object.

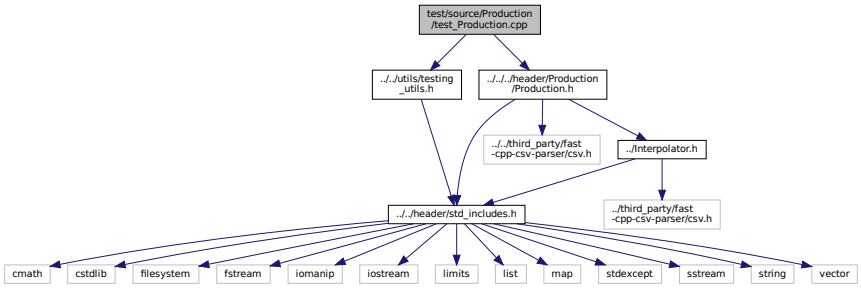
```
140 {
141     testFloatEquals(
142         test_wind_ptr->computeProductionkW(0, 1, 1e6),
143         0,
144         __FILE__,
145         __LINE__
146     );
147 
148     testFloatEquals(
149         test_wind_ptr->computeProductionkW(
150             0,
151             1,
152             ((Wind*)test_wind_ptr)->design_speed_ms
153         ),
154         test_wind_ptr->capacity_kW,
155         __FILE__,
156         __LINE__
157     );
158 
159     testFloatEquals(
160         test_wind_ptr->computeProductionkW(0, 1, -1),
161         0,
162         __FILE__,
163         __LINE__
164     );
165 
166     return;
167 }   /* testProductionConstraint_Wind() */
```

5.65 test/source/Production/test_Production.cpp File Reference

Testing suite for Production class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Production.h"
```

Include dependency graph for test_Production.cpp:



Functions

- Production * testConstruct_Production (std::vector< double > *time_vec_hrs_ptr)

  A function to construct a Production object and spot check some post-construction attributes.
- void testBadConstruct_Production (std::vector< double > *time_vec_hrs_ptr)

  Function to test the trying to construct a Production object given bad inputs is being handled as expected.
- int main (int argc, char **argv)

5.65.1 Detailed Description

Testing suite for Production class.

A suite of tests for the Production class.

5.65.2 Function Documentation

5.65.2.1 main()

```
int main (
              int argc,
              char ** argv )
178 {
179     #ifdef _WIN32
180         activateVirtualTerminal();
181     #endif  /* _WIN32 */
182
183     printGold("\tTesting Production");
184
185     srand(time(NULL));
186
187
188     std::vector<double> time_vec_hrs (8760, 0);
189     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
190         time_vec_hrs[i] = i;
191     }
192
193     Production* test_production_ptr = testConstruct_Production(&time_vec_hrs);
194
195
196     try {
197         testBadConstruct_Production(&time_vec_hrs);
198     }
199
200
201     catch (...) {
202         delete test_production_ptr;
203
204         printGold(" ............................... ");
205         printRed("FAIL");
206         std::cout « std::endl;
207         throw;
208     }
209
210
211     delete test_production_ptr;
212
213     printGold(" ............................... ");
214     printGreen("PASS");
215     std::cout « std::endl;
216     return 0;
217
218 }   /* main() */
```

5.65.2.2 testBadConstruct_Production()

```
void testBadConstruct_Production (
              std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Production object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
152 {
153     bool error_flag = true;
154 
155     try {
156         ProductionInputs production_inputs;
157 
158         Production bad_production(0, 1, production_inputs, time_vec_hrs_ptr);
159 
160         error_flag = false;
161     } catch (...) {
162         // Task failed successfully! =P
163     }
164     if (not error_flag) {
165         expectedErrorNotDetected(__FILE__, __LINE__);
166     }
167 
168     return;
169 }   /* testBadConstruct_Production() */
```

5.65.2.3 testConstruct_Production()

```
Production * testConstruct_Production (
              std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Production object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Production object.

```
40 {
41     ProductionInputs production_inputs;
42 
43     Production* test_production_ptr = new Production(
44         8760,
45         1,
46         production_inputs,
47         time_vec_hrs_ptr
48     );
49 
50     testTruth(
51         not production_inputs.print_flag,
52         __FILE__,
53         __LINE__
54     );
55 
56     testFloatEquals(
57         production_inputs.nominal_inflation_annual,
58         0.02,
59         __FILE__,
60         __LINE__
61     );
62 
63     testFloatEquals(
64         production_inputs.nominal_discount_annual,
65         0.04,
66         __FILE__,
67         __LINE__
68     );
69 
70     testFloatEquals(
71         test_production_ptr->n_points,
```

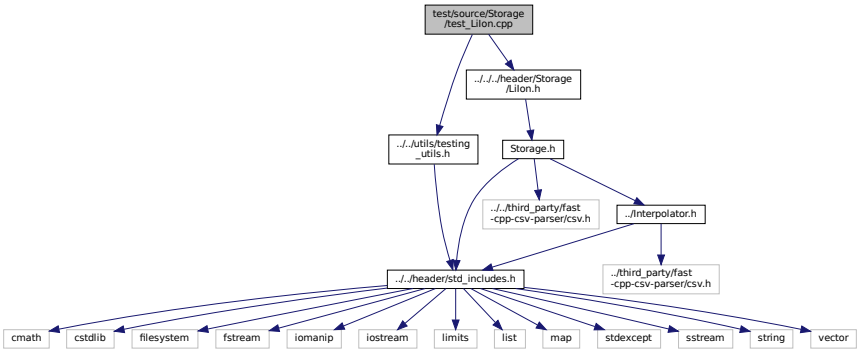
```
72          8760,
73          __FILE__,
74          __LINE__
75      );
76
77      testFloatEquals(
78          test_production_ptr->capacity_kW,
79          100,
80          __FILE__,
81          __LINE__
82      );
83
84      testFloatEquals(
85          test_production_ptr->real_discount_annual,
86          0.0196078431372549,
87          __FILE__,
88          __LINE__
89      );
90
91      testFloatEquals(
92          test_production_ptr->production_vec_kW.size(),
93          8760,
94          __FILE__,
95          __LINE__
96      );
97
98      testFloatEquals(
99          test_production_ptr->dispatch_vec_kW.size(),
100          8760,
101          __FILE__,
102          __LINE__
103      );
104
105      testFloatEquals(
106          test_production_ptr->storage_vec_kW.size(),
107          8760,
108          __FILE__,
109          __LINE__
110      );
111
112      testFloatEquals(
113          test_production_ptr->curtailment_vec_kW.size(),
114          8760,
115          __FILE__,
116          __LINE__
117      );
118
119      testFloatEquals(
120          test_production_ptr->capital_cost_vec.size(),
121          8760,
122          __FILE__,
123          __LINE__
124      );
125
126      testFloatEquals(
127          test_production_ptr->operation_maintenance_cost_vec.size(),
128          8760,
129          __FILE__,
130          __LINE__
131      );
132
133      return test_production_ptr;
134  }   /* testConstruct_Production() */
```

5.66 test/source/Storage/test_LiIon.cpp File Reference

Testing suite for LiIon class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Storage/LiIon.h"
```

Include dependency graph for test_LiIon.cpp:



Functions

- Storage ∗ testConstruct_LiIon (void)

  A function to construct a LiIon object and spot check some post-construction attributes.
- void testBadConstruct_LiIon (void)

  Function to test the trying to construct a LiIon object given bad inputs is being handled as expected.
- void testCommitCharge_LiIon (Storage ∗test_liion_ptr)

  A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.
- void testCommitDischarge_LiIon (Storage ∗test_liion_ptr)

  A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.
- int main (int argc, char ∗∗argv)

5.66.1 Detailed Description

Testing suite for LiIon class.

A suite of tests for the LiIon class.

5.66.2 Function Documentation

5.66.2.1 main()

```
int main (
            int argc,
            char ** argv )
306 {
307     #ifdef _WIN32
308         activateVirtualTerminal();
309     #endif  /* _WIN32 */
310
311     printGold("\tTesting Storage <-- LiIon");
312
313     srand(time(NULL));
314
```

```
315
316     Storage* test_liion_ptr = testConstruct_LiIon();
317
318
319     try {
320         testBadConstruct_LiIon();
321
322         testCommitCharge_LiIon(test_liion_ptr);
323         testCommitDischarge_LiIon(test_liion_ptr);
324     }
325
326
327     catch (...) {
328         delete test_liion_ptr;
329
330         printGold(" ......................... ");
331         printRed("FAIL");
332         std::cout « std::endl;
333         throw;
334     }
335
336
337     delete test_liion_ptr;
338
339     printGold(" ......................... ");
340     printGreen("PASS");
341     std::cout « std::endl;
342     return 0;
343
344 }   /* main() */
```

5.66.2.2 testBadConstruct_LiIon()

```
void testBadConstruct_LiIon (
              void  )
```

Function to test the trying to construct a LiIon object given bad inputs is being handled as expected.

```
149 {
150     bool error_flag = true;
151
152     try {
153         LiIonInputs bad_liion_inputs;
154         bad_liion_inputs.min_SOC = -1;
155
156         LiIon bad_liion(8760, 1, bad_liion_inputs);
157
158         error_flag = false;
159     } catch (...) {
160         // Task failed successfully! =P
161     }
162     if (not error_flag) {
163         expectedErrorNotDetected(__FILE__, __LINE__);
164     }
165
166     return;
167 }   /* testBadConstruct_LiIon() */
```

5.66.2.3 testCommitCharge_LiIon()

```
void testCommitCharge_LiIon (
              Storage * test_liion_ptr )
```

A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.

Parameters

test_liion_ptr	A Storage pointer to a test LiIon object.

```
185 {
186     double dt_hrs = 1;
187 
188     testFloatEquals(
189         test_liion_ptr->getAvailablekW(dt_hrs),
190         100,    // hits power capacity constraint
191         __FILE__,
192         __LINE__
193     );
194 
195     testFloatEquals(
196         test_liion_ptr->getAcceptablekW(dt_hrs),
197         100,    // hits power capacity constraint
198         __FILE__,
199         __LINE__
200     );
201 
202     test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing in
203 
204     testFloatEquals(
205         test_liion_ptr->getAvailablekW(dt_hrs),
206         0,    // is already hitting power capacity constraint
207         __FILE__,
208         __LINE__
209     );
210 
211     testFloatEquals(
212         test_liion_ptr->getAcceptablekW(dt_hrs),
213         0,    // is already hitting power capacity constraint
214         __FILE__,
215         __LINE__
216     );
217 
218     test_liion_ptr->commitCharge(0, dt_hrs, 100);
219 
220     testFloatEquals(
221         test_liion_ptr->power_kW,
222         0,
223         __FILE__,
224         __LINE__
225     );
226 
227     return;
228 }   /* testCommitCharge_LiIon() */
```

5.66.2.4 testCommitDischarge_LiIon()

```
void testCommitDischarge_LiIon (
            Storage * test_liion_ptr )
```

A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.

Parameters

test_liion_ptr	A Storage pointer to a test LiIon object.

```
246 {
247     double dt_hrs = 1;
248     double load_kW = 100;
249 
250     testFloatEquals(
251         test_liion_ptr->getAvailablekW(dt_hrs),
252         100,    // hits power capacity constraint
253         __FILE__,
254         __LINE__
255     );
256 
257     testFloatEquals(
258         test_liion_ptr->getAcceptablekW(dt_hrs),
259         100,    // hits power capacity constraint
260         __FILE__,
261         __LINE__
262     );
263 
264     test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing out
```

```
265
266     testFloatEquals(
267         test_liion_ptr->getAvailablekW(dt_hrs),
268         0,    // is already hitting power capacity constraint
269         __FILE__,
270         __LINE__
271     );
272
273     testFloatEquals(
274         test_liion_ptr->getAcceptablekW(dt_hrs),
275         0,    // is already hitting power capacity constraint
276         __FILE__,
277         __LINE__
278     );
279
280     load_kW = test_liion_ptr->commitDischarge(0, dt_hrs, 100, load_kW);
281
282     testFloatEquals(
283         load_kW,
284         0,
285         __FILE__,
286         __LINE__
287     );
288
289     testFloatEquals(
290         test_liion_ptr->power_kW,
291         0,
292         __FILE__,
293         __LINE__
294     );
295
296     return;
297 }   /* testCommitDischarge_LiIon() */
```

5.66.2.5 testConstruct_LiIon()

```
Storage * testConstruct_LiIon (
            void  )
```

A function to construct a LiIon object and spot check some post-construction attributes.

Returns

A Storage pointer to a test LiIon object.

```
38 {
39     LiIonInputs liion_inputs;
40
41     Storage* test_liion_ptr = new LiIon(8760, 1, liion_inputs);
42
43     testTruth(
44         test_liion_ptr->type_str == "LIION",
45         __FILE__,
46         __LINE__
47     );
48
49     testFloatEquals(
50         ((LiIon*)test_liion_ptr)->init_SOC,
51         0.5,
52         __FILE__,
53         __LINE__
54     );
55
56     testFloatEquals(
57         ((LiIon*)test_liion_ptr)->min_SOC,
58         0.15,
59         __FILE__,
60         __LINE__
61     );
62
63     testFloatEquals(
64         ((LiIon*)test_liion_ptr)->hysteresis_SOC,
65         0.5,
66         __FILE__,
67         __LINE__
68     );
```

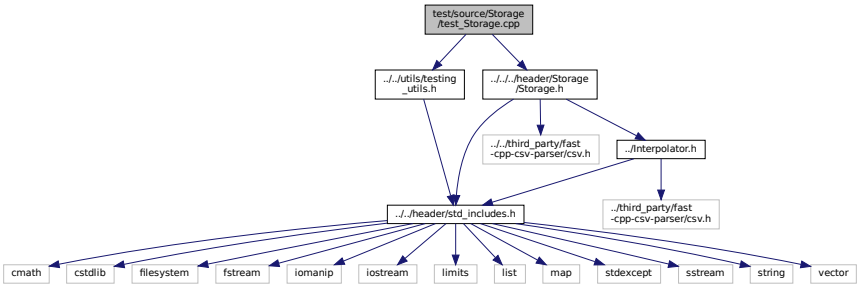
```
69
70      testFloatEquals(
71          ((LiIon*)test_liion_ptr)->max_SOC,
72          0.9,
73          __FILE__,
74          __LINE__
75      );
76
77      testFloatEquals(
78          ((LiIon*)test_liion_ptr)->charging_efficiency,
79          0.9,
80          __FILE__,
81          __LINE__
82      );
83
84      testFloatEquals(
85          ((LiIon*)test_liion_ptr)->discharging_efficiency,
86          0.9,
87          __FILE__,
88          __LINE__
89      );
90
91      testFloatEquals(
92          ((LiIon*)test_liion_ptr)->replace_SOH,
93          0.8,
94          __FILE__,
95          __LINE__
96      );
97
98      testFloatEquals(
99          ((LiIon*)test_liion_ptr)->power_kW,
100         0,
101         __FILE__,
102         __LINE__
103     );
104
105     testFloatEquals(
106         ((LiIon*)test_liion_ptr)->SOH_vec.size(),
107         8760,
108         __FILE__,
109         __LINE__
110     );
111
112     testTruth(
113         not ((LiIon*)test_liion_ptr)->power_degradation_flag,
114         __FILE__,
115         __LINE__
116     );
117
118     testFloatEquals(
119         test_liion_ptr->energy_capacity_kWh,
120         ((LiIon*)test_liion_ptr)->dynamic_energy_capacity_kWh,
121         __FILE__,
122         __LINE__
123     );
124
125     testFloatEquals(
126         test_liion_ptr->power_capacity_kW,
127         ((LiIon*)test_liion_ptr)->dynamic_power_capacity_kW,
128         __FILE__,
129         __LINE__
130     );
131
132     return test_liion_ptr;
133 }   /* testConstruct_LiIon() */
```

5.67 test/source/Storage/test_Storage.cpp File Reference

Testing suite for Storage class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Storage/Storage.h"
```

Include dependency graph for test_Storage.cpp:



Functions

- Storage * testConstruct_Storage (void)

  A function to construct a Storage object and spot check some post-construction attributes.
- void testBadConstruct_Storage (void)

  Function to test the trying to construct a Storage object given bad inputs is being handled as expected.
- int main (int argc, char **argv)

5.67.1 Detailed Description

Testing suite for Storage class.

A suite of tests for the Storage class.

5.67.2 Function Documentation

5.67.2.1 main()

```
int main (
            int argc,
            char ** argv )
136 {
137     #ifdef _WIN32
138         activateVirtualTerminal();
139     #endif  /* _WIN32 */
140
141     printGold("\tTesting Storage");
142
143     srand(time(NULL));
144
145
146     Storage* test_storage_ptr = testConstruct_Storage();
147
148
149     try {
150         testBadConstruct_Storage();
151     }
152
153
154     catch (...) {
```

```
155         delete test_storage_ptr;
156
157         printGold(" ................................. ");
158         printRed("FAIL");
159         std::cout « std::endl;
160         throw;
161     }
162
163
164     delete test_storage_ptr;
165
166     printGold(" ................................. ");
167     printGreen("PASS");
168     std::cout « std::endl;
169     return 0;
170
171 }   /* main() */
```

5.67.2.2 testBadConstruct_Storage()

```
void testBadConstruct_Storage (
             void  )
```

Function to test the trying to construct a Storage object given bad inputs is being handled as expected.

```
109 {
110     bool error_flag = true;
111
112     try {
113         StorageInputs bad_storage_inputs;
114         bad_storage_inputs.energy_capacity_kWh = 0;
115
116         Storage bad_storage(8760, 1, bad_storage_inputs);
117
118         error_flag = false;
119     } catch (...) {
120         // Task failed successfully! =P
121     }
122     if (not error_flag) {
123         expectedErrorNotDetected(__FILE__, __LINE__);
124     }
125
126     return;
127 }   /* testBadConstruct_Storage() */
```

5.67.2.3 testConstruct_Storage()

```
Storage * testConstruct_Storage (
             void  )
```

A function to construct a Storage object and spot check some post-construction attributes.

Returns

A Renewable pointer to a test Storage object.

```
38 {
39     StorageInputs storage_inputs;
40
41     Storage* test_storage_ptr = new Storage(8760, 1, storage_inputs);
42
43     testFloatEquals(
44         test_storage_ptr->power_capacity_kW,
45         100,
46         __FILE__,
47         __LINE__
48     );
49
```

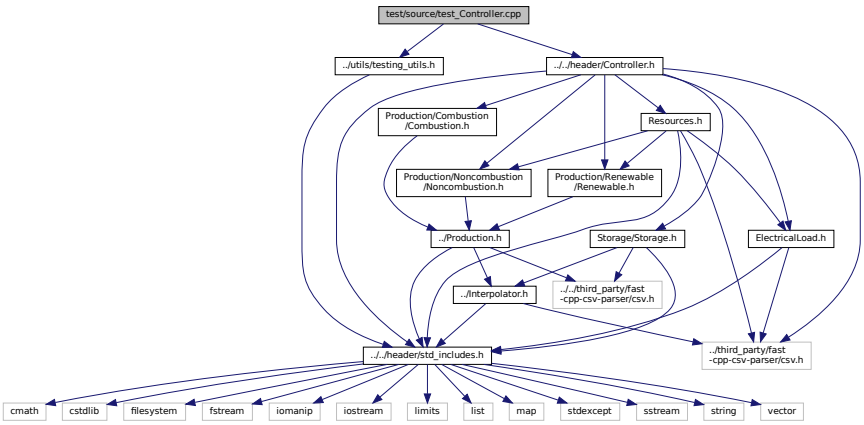
```
50      testFloatEquals(
51          test_storage_ptr->energy_capacity_kWh,
52          1000,
53          __FILE__,
54          __LINE__
55      );
56
57      testFloatEquals(
58          test_storage_ptr->charge_vec_kWh.size(),
59          8760,
60          __FILE__,
61          __LINE__
62      );
63
64      testFloatEquals(
65          test_storage_ptr->charging_power_vec_kW.size(),
66          8760,
67          __FILE__,
68          __LINE__
69      );
70
71      testFloatEquals(
72          test_storage_ptr->discharging_power_vec_kW.size(),
73          8760,
74          __FILE__,
75          __LINE__
76      );
77
78      testFloatEquals(
79          test_storage_ptr->capital_cost_vec.size(),
80          8760,
81          __FILE__,
82          __LINE__
83      );
84
85      testFloatEquals(
86          test_storage_ptr->operation_maintenance_cost_vec.size(),
87          8760,
88          __FILE__,
89          __LINE__
90      );
91
92      return test_storage_ptr;
93  }   /* testConstruct_Storage() */
```

5.68 test/source/test_Controller.cpp File Reference

Testing suite for Controller class.

```
#include "../utils/testing_utils.h"
#include "../../header/Controller.h"
```

Include dependency graph for test_Controller.cpp:



Functions

- Controller * testConstruct_Controller (void)

  A function to construct a Controller object.
- int main (int argc, char **argv)

5.68.1 Detailed Description

Testing suite for Controller class.

A suite of tests for the Controller class.

5.68.2 Function Documentation

5.68.2.1 main()

```
int main (
              int argc,
              char ** argv )
50 {
51     #ifdef _WIN32
52         activateVirtualTerminal();
53     #endif  /* _WIN32 */
54
55     printGold("\tTesting Controller");
56
57     srand(time(NULL));
58
59
60     Controller* test_controller_ptr = testConstruct_Controller();
61
62
63     try {
64         //...
65     }
66
67
68     catch (...) {
69         delete test_controller_ptr;
70
71         printGold(" ............................... ");
72         printRed("FAIL");
73         std::cout << std::endl;
74         throw;
75     }
76
77
78     delete test_controller_ptr;
79
80     printGold(" ............................... ");
81     printGreen("PASS");
82     std::cout << std::endl;
83     return 0;
84 }   /* main() */
```

5.68.2.2 testConstruct_Controller()

```
Controller * testConstruct_Controller (
             void  )
```

A function to construct a Controller object.

Returns

A pointer to a test Controller object.

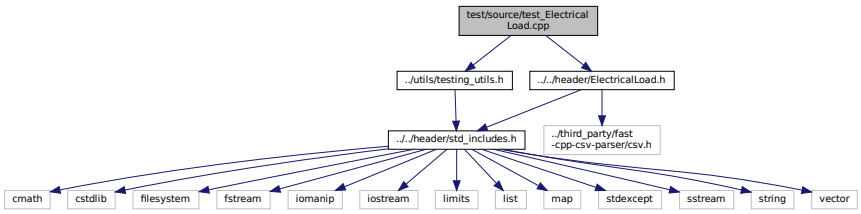
```
37 {
38     Controller* test_controller_ptr = new Controller();
39
40     return test_controller_ptr;
41 }   /* testConstruct_Controller() */
```

5.69 test/source/test_ElectricalLoad.cpp File Reference

Testing suite for ElectricalLoad class.

```
#include "../utils/testing_utils.h"
#include "../../header/ElectricalLoad.h"
```

Include dependency graph for test_ElectricalLoad.cpp:



Functions

- ElectricalLoad * testConstruct_ElectricalLoad (void)

  A function to construct an ElectricalLoad object.
- void testPostConstructionAttributes_ElectricalLoad (ElectricalLoad *test_electrical_load_ptr)

  A function to check the values of various post-construction attributes.
- void testDataRead_ElectricalLoad (ElectricalLoad *test_electrical_load_ptr)

  A function to check the values read into the test ElectricalLoad object.
- int main (int argc, char **argv)

5.69.1 Detailed Description

Testing suite for ElectricalLoad class.

A suite of tests for the ElectricalLoad class.

5.69.2 Function Documentation

5.69.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
223 {
224     #ifdef _WIN32
225         activateVirtualTerminal();
226     #endif  /* _WIN32 */
227 
228     printGold("\tTesting ElectricalLoad");
229 
230     srand(time(NULL));
231 
232 
233     ElectricalLoad* test_electrical_load_ptr = testConstruct_ElectricalLoad();
234 
235 
236     try {
237         testPostConstructionAttributes_ElectricalLoad(test_electrical_load_ptr);
238         testDataRead_ElectricalLoad(test_electrical_load_ptr);
239     }
240 
241 
242     catch (...) {
243         delete test_electrical_load_ptr;
244 
245         printGold(" ........................... ");
246         printRed("FAIL");
247         std::cout « std::endl;
248         throw;
249     }
250 
251 
252     delete test_electrical_load_ptr;
253 
254     printGold(" ........................... ");
255     printGreen("PASS");
256     std::cout « std::endl;
257     return 0;
258 }   /* main() */
```

5.69.2.2 testConstruct_ElectricalLoad()

```
ElectricalLoad * testConstruct_ElectricalLoad (
              void  )
```

A function to construct an ElectricalLoad object.

Returns

A pointer to a test ElectricalLoad object.

```
37 {
38     std::string path_2_electrical_load_time_series =
39         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
40 
41     ElectricalLoad* test_electrical_load_ptr =
42         new ElectricalLoad(path_2_electrical_load_time_series);
43 
44     testTruth(
45         test_electrical_load_ptr->path_2_electrical_load_time_series ==
46         path_2_electrical_load_time_series,
47         __FILE__,
48         __LINE__
49     );
50 
51     return test_electrical_load_ptr;
52 }   /* testConstruct_ElectricalLoad() */
```

5.69.2.3 testDataRead_ElectricalLoad()

```
void testDataRead_ElectricalLoad (
            ElectricalLoad * test_electrical_load_ptr )
```

A function to check the values read into the test ElectricalLoad object.

Parameters

test_electrical_load_ptr	A pointer to the test ElectricalLoad object.

```
128 {
129     std::vector<double> expected_dt_vec_hrs (48, 1);
130 
131     std::vector<double> expected_time_vec_hrs = {
132          0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
133         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
134         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
135         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
136     };
137 
138     std::vector<double> expected_load_vec_kW = {
139         360.253836463674,
140         355.171277826775,
141         353.776453532298,
142         353.75405737934,
143         346.592867404975,
144         340.132411175118,
145         337.354867340578,
146         340.644115618736,
147         363.639028500678,
148         378.787797779238,
149         372.215798201712,
150         395.093925731298,
151         402.325427142659,
152         386.907725462306,
153         380.709170928091,
154         372.062070914977,
155         372.328646856954,
156         391.841444284136,
157         394.029351759596,
158         383.369407765254,
159         381.093099675206,
160         382.604158946193,
161         390.744843709034,
162         383.13949492437,
163         368.150393976985,
164         364.629744480226,
165         363.572736804082,
166         359.854924202248,
167         355.207590170267,
168         349.094656012401,
169         354.365935871597,
170         343.380608328546,
171         404.673065729266,
172         486.296896820126,
173         480.225974100847,
174         457.318764401085,
175         418.177339948609,
176         414.399018364126,
177         409.678420185754,
178         404.768766016563,
179         401.699589920585,
180         402.44339040654,
181         398.138372541906,
182         396.010498627646,
183         390.165117432277,
184         375.850429417013,
185         365.567100746484,
186         365.429624610923
187     };
188 
189     for (int i = 0; i < 48; i++) {
190         testFloatEquals(
191             test_electrical_load_ptr->dt_vec_hrs[i],
192             expected_dt_vec_hrs[i],
193             __FILE__,
194             __LINE__
195         );
196 
197         testFloatEquals(
```

```
198             test_electrical_load_ptr->time_vec_hrs[i],
199             expected_time_vec_hrs[i],
200             __FILE__,
201             __LINE__
202         );
203 
204         testFloatEquals(
205             test_electrical_load_ptr->load_vec_kW[i],
206             expected_load_vec_kW[i],
207             __FILE__,
208             __LINE__
209         );
210 
211     }
212 
213     return;
214 }   /* testDataRead_ElectricalLoad() */
```

5.69.2.4 testPostConstructionAttributes_ElectricalLoad()

```
void testPostConstructionAttributes_ElectricalLoad (
            ElectricalLoad * test_electrical_load_ptr )
```

A function to check the values of various post-construction attributes.

Parameters

test_electrical_load_ptr	A pointer to the test ElectricalLoad object.

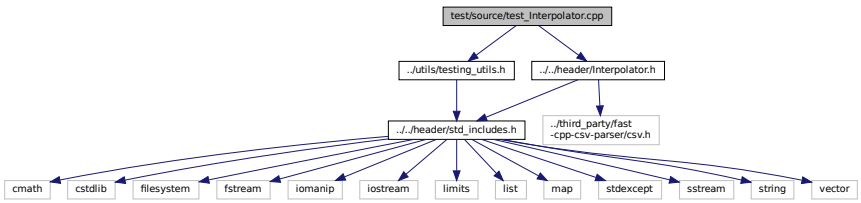
```
73 {
74     testFloatEquals(
75         test_electrical_load_ptr->n_points,
76         8760,
77         __FILE__,
78         __LINE__
79     );
80 
81     testFloatEquals(
82         test_electrical_load_ptr->n_years,
83         0.999886,
84         __FILE__,
85         __LINE__
86     );
87 
88     testFloatEquals(
89         test_electrical_load_ptr->min_load_kW,
90         82.1211213927802,
91         __FILE__,
92         __LINE__
93     );
94 
95     testFloatEquals(
96         test_electrical_load_ptr->mean_load_kW,
97         258.373472633202,
98         __FILE__,
99         __LINE__
100     );
101 
102 
103     testFloatEquals(
104         test_electrical_load_ptr->max_load_kW,
105         500,
106         __FILE__,
107         __LINE__
108     );
109 
110     return;
111 }   /* testPostConstructionAttributes_ElectricalLoad() */
```

5.70 test/source/test_Interpolator.cpp File Reference

Testing suite for Interpolator class.

```
#include "../utils/testing_utils.h"
#include "../../header/Interpolator.h"
```

Include dependency graph for test_Interpolator.cpp:



Functions

- Interpolator * testConstruct_Interpolator (void)

  A function to construct an Interpolator object.
- void testDataRead1D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_1D, std::string path_2↩ _data_1D)

  A function to check the 1D data values read into the Interpolator object.
- void testBadIndexing1D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_bad)

  A function to check if bad key errors are being handled properly.
- void testInvalidInterpolation1D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_1D)

  Function to check if attempting to interpolate outside the given 1D data domain is handled properly.
- void testInterpolation1D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_1D)

  Function to check that the Interpolator object is returning the expected 1D interpolation values.
- void testDataRead2D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_2D, std::string path_2↩ _data_2D)

  A function to check the 2D data values read into the Interpolator object.
- void testInvalidInterpolation2D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_2D)

  Function to check if attempting to interpolate outside the given 2D data domain is handled properly.
- void testInterpolation2D_Interpolator (Interpolator *test_interpolator_ptr, int data_key_2D)

  Function to check that the Interpolator object is returning the expected 2D interpolation values.
- int main (int argc, char **argv)

5.70.1 Detailed Description

Testing suite for Interpolator class.

A suite of tests for the Interpolator class.

5.70.2 Function Documentation

5.70.2.1 main()

```
int main (
              int argc,
              char ** argv )
700 {
701     #ifdef _WIN32
702         activateVirtualTerminal();
703     #endif  /* _WIN32 */
704
705     printGold("\n\tTesting Interpolator");
706
707     srand(time(NULL));
708
709
710     Interpolator* test_interpolator_ptr = testConstruct_Interpolator();
711
712
713     try {
714         int data_key_1D = 1;
715         std::string path_2_data_1D =
716             "data/test/interpolation/diesel_fuel_curve.csv";
717
718         testDataRead1D_Interpolator(test_interpolator_ptr, data_key_1D, path_2_data_1D);
719         testBadIndexing1D_Interpolator(test_interpolator_ptr, -99);
720         testInvalidInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
721         testInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
722
723
724         int data_key_2D = 2;
725         std::string path_2_data_2D =
726             "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
727
728         testDataRead2D_Interpolator(test_interpolator_ptr, data_key_2D, path_2_data_2D);
729         testInvalidInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
730         testInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
731     }
732
733
734     catch (...) {
735         delete test_interpolator_ptr;
736
737         printGold(" ............................. ");
738         printRed("FAIL");
739         std::cout « std::endl;
740         throw;
741     }
742
743
744     delete test_interpolator_ptr;
745
746     printGold(" ............................. ");
747     printGreen("PASS");
748     std::cout « std::endl;
749     return 0;
750 }   /* main() */
```

5.70.2.2 testBadIndexing1D_Interpolator()

```
void testBadIndexing1D_Interpolator (
              Interpolator * test_interpolator_ptr,
              int data_key_bad )
```

A function to check if bad key errors are being handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_bad	A key used to index into the Interpolator object.

```
187 {
```

```
188     bool error_flag = true;
189
190     try {
191         test_interpolator_ptr->interp1D(data_key_bad, 0);
192         error_flag = false;
193     } catch (...) {
194         // Task failed successfully! =P
195     }
196     if (not error_flag) {
197         expectedErrorNotDetected(__FILE__, __LINE__);
198     }
199
200     return;
201 }   /* testBadIndexing1D_Interpolator() */
```

5.70.2.3 testConstruct_Interpolator()

```
Interpolator * testConstruct_Interpolator (
            void  )
```

A function to construct an Interpolator object.

Returns

A pointer to a test Interpolator object.

```
37 {
38     Interpolator* test_interpolator_ptr = new Interpolator();
39
40     return test_interpolator_ptr;
41 }   /* testConstruct_Interpolator() */
```

5.70.2.4 testDataRead1D_Interpolator()

```
void testDataRead1D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key,
            std::string path_2_data_1D )
```

A function to check the 1D data values read into the Interpolator object.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.
path_2_data_1D	A path (either relative or absolute) to the interpolation data.

```
70 {
71     test_interpolator_ptr->addData1D(data_key_1D, path_2_data_1D);
72
73     testTruth(
74         test_interpolator_ptr->path_map_1D[data_key_1D] == path_2_data_1D,
75         __FILE__,
76         __LINE__
77     );
78
79     testFloatEquals(
80         test_interpolator_ptr->interp_map_1D[data_key_1D].n_points,
81         16,
82         __FILE__,
```

```
83          __LINE__
84      );
85
86      testFloatEquals(
87          test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec.size(),
88          16,
89          __FILE__,
90          __LINE__
91      );
92
93      std::vector<double> expected_x_vec = {
94          0,
95          0.3,
96          0.35,
97          0.4,
98          0.45,
99          0.5,
100          0.55,
101          0.6,
102          0.65,
103          0.7,
104          0.75,
105          0.8,
106          0.85,
107          0.9,
108          0.95,
109          1
110      };
111
112      std::vector<double> expected_y_vec = {
113          4.68079520372916,
114          11.1278522361839,
115          12.4787834830748,
116          13.7808847600209,
117          15.0417468303382,
118          16.277263,
119          17.4612831516442,
120          18.6279054806525,
121          19.7698039220515,
122          20.8893499214868,
123          21.955378,
124          23.0690535155297,
125          24.1323614374927,
126          25.1797231192866,
127          26.2122451458747,
128          27.254952
129      };
130
131      for (int i = 0; i < test_interpolator_ptr->interp_map_1D[data_key_1D].n_points; i++) {
132          testFloatEquals(
133              test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec[i],
134              expected_x_vec[i],
135              __FILE__,
136              __LINE__
137          );
138
139          testFloatEquals(
140              test_interpolator_ptr->interp_map_1D[data_key_1D].y_vec[i],
141              expected_y_vec[i],
142              __FILE__,
143              __LINE__
144          );
145      }
146
147      testFloatEquals(
148          test_interpolator_ptr->interp_map_1D[data_key_1D].min_x,
149          expected_x_vec[0],
150          __FILE__,
151          __LINE__
152      );
153
154      testFloatEquals(
155          test_interpolator_ptr->interp_map_1D[data_key_1D].max_x,
156          expected_x_vec[expected_x_vec.size() - 1],
157          __FILE__,
158          __LINE__
159      );
160
161      return;
162  }   /* testDataRead1D_Interpolator() */
```

5.70.2.5 testDataRead2D_Interpolator()

```
void testDataRead2D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key,
            std::string path_2_data_2D )
```

A function to check the 2D data values read into the Interpolator object.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.
path_2_data_2D	A path (either relative or absolute) to the interpolation data.

```
377 {
378     test_interpolator_ptr->addData2D(data_key_2D, path_2_data_2D);
379 
380     testTruth(
381         test_interpolator_ptr->path_map_2D[data_key_2D] == path_2_data_2D,
382         __FILE__,
383         __LINE__
384     );
385 
386     testFloatEquals(
387         test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows,
388         16,
389         __FILE__,
390         __LINE__
391     );
392 
393     testFloatEquals(
394         test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols,
395         16,
396         __FILE__,
397         __LINE__
398     );
399 
400     testFloatEquals(
401         test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec.size(),
402         16,
403         __FILE__,
404         __LINE__
405     );
406 
407     testFloatEquals(
408         test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec.size(),
409         16,
410         __FILE__,
411         __LINE__
412     );
413 
414     testFloatEquals(
415         test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix.size(),
416         16,
417         __FILE__,
418         __LINE__
419     );
420 
421     testFloatEquals(
422         test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[0].size(),
423         16,
424         __FILE__,
425         __LINE__
426     );
427 
428     std::vector<double> expected_x_vec = {
429         0.25, 0.75, 1.25, 1.75, 2.25, 2.75, 3.25, 3.75, 4.25, 4.75, 5.25, 5.75, 6.25, 6.75, 7.25, 7.75
430     };
431 
432     std::vector <double> expected_y_vec = {
433         5,
434         6,
435         7,
436         8,
437         9,
438         10,
439         11,
```

```
440         12,
441         13,
442         14,
443         15,
444         16,
445         17,
446         18,
447         19,
448         20
449     };
450
451     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; i++) {
452         testFloatEquals(
453             test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec[i],
454             expected_x_vec[i],
455             __FILE__,
456             __LINE__
457         );
458     }
459
460     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
461         testFloatEquals(
462             test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec[i],
463             expected_y_vec[i],
464             __FILE__,
465             __LINE__
466         );
467     }
468
469     testFloatEquals(
470         test_interpolator_ptr->interp_map_2D[data_key_2D].min_x,
471         expected_x_vec[0],
472         __FILE__,
473         __LINE__
474     );
475
476     testFloatEquals(
477         test_interpolator_ptr->interp_map_2D[data_key_2D].max_x,
478         expected_x_vec[expected_x_vec.size() - 1],
479         __FILE__,
480         __LINE__
481     );
482
483     testFloatEquals(
484         test_interpolator_ptr->interp_map_2D[data_key_2D].min_y,
485         expected_y_vec[0],
486         __FILE__,
487         __LINE__
488     );
489
490     testFloatEquals(
491         test_interpolator_ptr->interp_map_2D[data_key_2D].max_y,
492         expected_y_vec[expected_y_vec.size() - 1],
493         __FILE__,
494         __LINE__
495     );
496
497     std::vector<std::vector<double> expected_z_matrix = {
498         {0, 0.129128125, 0.268078125, 0.404253125, 0.537653125, 0.668278125, 0.796128125, 0.921203125,
    1, 1, 1, 0, 0, 0, 0, 0},
499         {0, 0.11160375, 0.24944375, 0.38395375, 0.51513375, 0.64298375, 0.76750375, 0.88869375, 1, 1, 1,
    1, 1, 1, 1, 1},
500         {0, 0.094079375, 0.230809375, 0.363654375, 0.492614375, 0.617689375, 0.738879375, 0.856184375,
    0.969604375, 1, 1, 1, 1, 1, 1, 1},
501         {0, 0.076555, 0.212175, 0.343355, 0.470095, 0.592395, 0.710255, 0.823675, 0.932655, 1, 1, 1, 1,
    1, 1, 1},
502         {0, 0.059030625, 0.193540625, 0.323055625, 0.447575625, 0.567100625, 0.681630625, 0.791165625,
    0.895705625, 0.995250625, 1, 1, 1, 1, 1, 1},
503         {0, 0.04150625, 0.17490625, 0.30275625, 0.42505625, 0.54180625, 0.65300625, 0.75865625,
    0.85875625, 0.95330625, 1, 1, 1, 1, 1, 1},
504         {0, 0.023981875, 0.156271875, 0.282456875, 0.402536875, 0.516511875, 0.624381875, 0.726146875,
    0.821806875, 0.911361875, 0.994811875, 1, 1, 1, 1, 1},
505         {0, 0.0064575, 0.1376375, 0.2621575, 0.3800175, 0.4912175, 0.5957575, 0.6936375, 0.7848575,
    0.8694175, 0.9473175, 1, 1, 1, 1, 1},
506         {0, 0, 0.119003125, 0.241858125, 0.357498125, 0.465923125, 0.567133125, 0.661128125,
    0.747908125, 0.827473125, 0.899823125, 0.964958125, 1, 1, 1, 1},
507         {0, 0, 0.10036875, 0.22155875, 0.33497875, 0.44062875, 0.53850875, 0.62861875, 0.71095875,
    0.78552875, 0.85232875, 0.91135875, 0.96261875, 1, 1, 1},
508         {0, 0, 0.081734375, 0.201259375, 0.312459375, 0.415334375, 0.509884375, 0.596109375,
    0.674009375, 0.743584375, 0.804834375, 0.857759375, 0.902359375, 0.938634375, 0.966584375,
    0.986209375},
509         {0, 0, 0.0631, 0.18096, 0.28994, 0.39004, 0.48126, 0.5636, 0.63706, 0.70164, 0.75734, 0.80416,
    0.8421, 0.87116, 0.89134, 0.90264},
510         {0, 0, 0.044465625, 0.160660625, 0.267420625, 0.364745625, 0.452635625, 0.531090625,
    0.600110625, 0.659695625, 0.709845625, 0.750560625, 0.781840625, 0.803685624999999, 0.816095625,
    0.819070625},
511         {0, 0, 0.02583125, 0.14036125, 0.24490125, 0.33945125, 0.42401125, 0.49858125, 0.56316125,
```

```
       0.61775125, 0.66235125, 0.69696125, 0.72158125, 0.73621125, 0.74085125, 0.73550125},
512         {0, 0, 0.007196875, 0.120061875, 0.222381875, 0.314156875, 0.395386875, 0.466071875,
       0.526211875, 0.575806875, 0.614856875, 0.643361875, 0.661321875, 0.668736875, 0.665606875,
       0.651931875},
513         {0, 0, 0, 0.0997625, 0.1998625, 0.2888625, 0.3667625, 0.4335625, 0.4892625, 0.5338625,
       0.5673625, 0.5897625, 0.6010625, 0.6012625, 0.5903625, 0.5683625}
514     };
515
516     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
517         for (int j = 0; j < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; j++) {
518             testFloatEquals(
519                 test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[i][j],
520                 expected_z_matrix[i][j],
521                 __FILE__,
522                 __LINE__
523             );
524         }
525     }
526
527     return;
528 }   /* testDataRead2D_Interpolator() */
```

5.70.2.6 testInterpolation1D_Interpolator()

```
void testInterpolation1D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_1D )
```

Function to check that the Interpolator object is returning the expected 1D interpolation values.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.

```
297 {
298     std::vector<double> interp_x_vec = {
299         0,
300         0.170812859791767,
301         0.322739274162545,
302         0.369750203682042,
303         0.443532869135929,
304         0.471567864244626,
305         0.536513734479662,
306         0.586125806988674,
307         0.601101175455075,
308         0.658356862575221,
309         0.70576929893201,
310         0.784069734739331,
311         0.805765927542453,
312         0.884747873186048,
313         0.930870496062112,
314         0.979415217694769,
315         1
316     };
317
318     std::vector<double> expected_interp_y_vec = {
319         4.68079520372916,
320         8.35159603357656,
321         11.7422361561399,
322         12.9931187917615,
323         14.8786636301325,
324         15.5746957307243,
325         17.1419229487141,
326         18.3041866133728,
327         18.6530540913696,
328         19.9569217633299,
329         21.012354614584,
330         22.7142305879957,
331         23.1916726441968,
332         24.8602332554707,
333         25.8172124624032,
334         26.8256741279932,
335         27.254952
```

```
336     };
337
338     for (size_t i = 0; i < interp_x_vec.size(); i++) {
339         testFloatEquals(
340             test_interpolator_ptr->interp1D(data_key_1D, interp_x_vec[i]),
341             expected_interp_y_vec[i],
342             __FILE__,
343             __LINE__
344         );
345     }
346
347     return;
348 }   /* testInterpolation1D_Interpolator() */
```

5.70.2.7 testInterpolation2D_Interpolator()

```
void testInterpolation2D_Interpolator (
             Interpolator * test_interpolator_ptr,
             int data_key_2D )
```

Function to check that the Interpolator object is returning the expected 2D interpolation values.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.

```
624 {
625     std::vector<double> interp_x_vec = {
626         0.389211848822208,
627         0.836477431896843,
628         1.52738334015579,
629         1.92640601114508,
630         2.27297317532019,
631         2.87416589636605,
632         3.72275770908175,
633         3.95063175885536,
634         4.68097139867404,
635         4.97775020449812,
636         5.55184219980547,
637         6.06566629451658,
638         6.27927876785062,
639         6.96218133671013,
640         7.51754442460228
641     };
642
643     std::vector<double> interp_y_vec = {
644         5.45741899698926,
645         6.00101329139007,
646         7.50567689404182,
647         8.77681262912881,
648         9.45143678206774,
649         10.7767876462885,
650         11.4795760857165,
651         12.9430684577599,
652         13.303544885703,
653         14.5069863517863,
654         15.1487890438045,
655         16.086524049077,
656         17.176609978648,
657         18.4155153740256,
658         19.1704554940162
659     };
660
661     std::vector<std::vector<double» expected_interp_z_matrix = {
662
        {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
663
        {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
664
        {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
665
        {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
```

```
666
        {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
667
        {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
668
        {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
669
        {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.6578
670
        {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
671
        {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
672
        {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
673
        {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
674
        {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
675
        {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.510
676
        {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.484
677     };
678
679     for (size_t i = 0; i < interp_y_vec.size(); i++) {
680         for (size_t j = 0; j < interp_x_vec.size(); j++) {
681             testFloatEquals(
682                 test_interpolator_ptr->interp2D(data_key_2D, interp_x_vec[j], interp_y_vec[i]),
683                 expected_interp_z_matrix[i][j],
684                 __FILE__,
685                 __LINE__
686             );
687         }
688     }
689
690     return;
691 }   /* testInterpolation2D_Interpolator() */
```

5.70.2.8 testInvalidInterpolation1D_Interpolator()

```
void testInvalidInterpolation1D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_1D )
```

Function to check if attempting to interpolate outside the given 1D data domain is handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.

```
227 {
228     bool error_flag = true;
229
230     try {
231         test_interpolator_ptr->interp1D(data_key_1D, -1);
232         error_flag = false;
233     } catch (...) {
234         // Task failed successfully! =P
235     }
236     if (not error_flag) {
237         expectedErrorNotDetected(__FILE__, __LINE__);
238     }
239
240     try {
241         test_interpolator_ptr->interp1D(data_key_1D, 2);
242         error_flag = false;
243     } catch (...) {
244         // Task failed successfully! =P
245     }
246     if (not error_flag) {
247         expectedErrorNotDetected(__FILE__, __LINE__);
248     }
249
```

```
250     try {
251         test_interpolator_ptr->interp1D(data_key_1D, 0 - FLOAT_TOLERANCE);
252         error_flag = false;
253     } catch (...) {
254         // Task failed successfully! =P
255     }
256     if (not error_flag) {
257         expectedErrorNotDetected(__FILE__, __LINE__);
258     }
259
260     try {
261         test_interpolator_ptr->interp1D(data_key_1D, 1 + FLOAT_TOLERANCE);
262         error_flag = false;
263     } catch (...) {
264         // Task failed successfully! =P
265     }
266     if (not error_flag) {
267         expectedErrorNotDetected(__FILE__, __LINE__);
268     }
269
270     return;
271 }   /* testInvalidInterpolation1D_Interpolator() */
```

5.70.2.9 testInvalidInterpolation2D_Interpolator()

```
void testInvalidInterpolation2D_Interpolator (
              Interpolator * test_interpolator_ptr,
              int data_key_2D )
```

Function to check if attempting to interpolate outside the given 2D data domain is handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.

```
554 {
555     bool error_flag = true;
556
557     try {
558         test_interpolator_ptr->interp2D(data_key_2D, -1, 6);
559         error_flag = false;
560     } catch (...) {
561         // Task failed successfully! =P
562     }
563     if (not error_flag) {
564         expectedErrorNotDetected(__FILE__, __LINE__);
565     }
566
567     try {
568         test_interpolator_ptr->interp2D(data_key_2D, 99, 6);
569         error_flag = false;
570     } catch (...) {
571         // Task failed successfully! =P
572     }
573     if (not error_flag) {
574         expectedErrorNotDetected(__FILE__, __LINE__);
575     }
576
577     try {
578         test_interpolator_ptr->interp2D(data_key_2D, 0.75, -1);
579         error_flag = false;
580     } catch (...) {
581         // Task failed successfully! =P
582     }
583     if (not error_flag) {
584         expectedErrorNotDetected(__FILE__, __LINE__);
585     }
586
587     try {
588         test_interpolator_ptr->interp2D(data_key_2D, 0.75, 99);
589         error_flag = false;
590     } catch (...) {
591         // Task failed successfully! =P
```

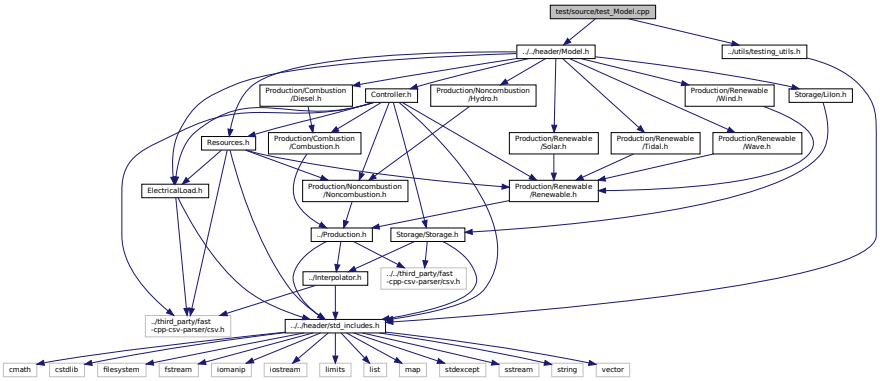
```
592     }
593     if (not error_flag) {
594         expectedErrorNotDetected(__FILE__, __LINE__);
595     }
596 
597     return;
598 }   /* testInvalidInterpolation2D_Interpolator() */
```

5.71 test/source/test_Model.cpp File Reference

Testing suite for Model class.

```
#include "../utils/testing_utils.h"
#include "../../header/Model.h"
```

Include dependency graph for test_Model.cpp:



Functions

- Model * testConstruct_Model (ModelInputs test_model_inputs)
- void testBadConstruct_Model (void)

  Function to check if passing bad ModelInputs to the Model constructor is handled appropriately.
- void testPostConstructionAttributes_Model (Model *test_model_ptr)

  A function to check the values of various post-construction attributes.
- void testElectricalLoadData_Model (Model *test_model_ptr)

  Function to check the values read into the ElectricalLoad component of the test Model object.
- void testAddSolarResource_Model (Model *test_model_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

  Function to test adding a solar resource and then check the values read into the Resources component of the test Model object.
- void testAddTidalResource_Model (Model *test_model_ptr, std::string path_2_tidal_resource_data, int tidal_resource_key)

  Function to test adding a tidal resource and then check the values read into the Resources component of the test Model object.
- void testAddWaveResource_Model (Model *test_model_ptr, std::string path_2_wave_resource_data, int wave_resource_key)

  Function to test adding a wave resource and then check the values read into the Resources component of the test Model object.
- void testAddWindResource_Model (Model *test_model_ptr, std::string path_2_wind_resource_data, int wind_resource_key)

Function to test adding a wind resource and then check the values read into the Resources component of the test Model object.

- void testAddHydroResource_Model (Model *test_model_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)

  Function to test adding a hydro resource and then check the values read into the Resources component of the test Model object.
- void testAddHydro_Model (Model *test_model_ptr, int hydro_resource_key)

  Function to test adding a hydroelectric asset to the test Model object, and then spot check some post-add attributes.
- void testAddDiesel_Model (Model *test_model_ptr)

  Function to test adding a suite of diesel generators to the test Model object, and then spot check some post-add attributes.
- void testAddSolar_Model (Model *test_model_ptr, int solar_resource_key)

  Function to test adding a solar PV array to the test Model object and then spot check some post-add attributes.
- void testAddSolar_productionOverride_Model (Model *test_model_ptr, std::string path_2_normalized_↩ production_time_series)

  Function to test adding a solar PV array to the test Model object using the production override feature, and then spot check some post-add attributes.
- void testAddTidal_Model (Model *test_model_ptr, int tidal_resource_key)

  Function to test adding a tidal turbine to the test Model object and then spot check some post-add attributes.
- void testAddWave_Model (Model *test_model_ptr, int wave_resource_key)

  Function to test adding a wave energy converter to the test Model object and then spot check some post-add attributes.
- void testAddWind_Model (Model *test_model_ptr, int wind_resource_key)

  Function to test adding a wind turbine to the test Model object and then spot check some post-add attributes.
- void testAddLiIon_Model (Model *test_model_ptr)

  Function to test adding a lithium ion battery energy storage system to the test Model object and then spot check some post-add attributes.
- void testLoadBalance_Model (Model *test_model_ptr)

  Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the Controller as expected.
- void testEconomics_Model (Model *test_model_ptr)

  Function to check that the modelled economic metrics are > 0.
- void testFuelConsumptionEmissions_Model (Model *test_model_ptr)

  Function to check that the modelled fuel consumption and emissions are > 0.
- int main (int argc, char **argv)

5.71.1 Detailed Description

Testing suite for Model class.

A suite of tests for the Model class.

5.71.2 Function Documentation

5.71.2.1 main()

```
int main (
              int argc,
              char ** argv )
1465 {
1466     #ifdef _WIN32
1467         activateVirtualTerminal();
1468     #endif  /* _WIN32 */
1469
1470     printGold("\tTesting Model");
1471     std::cout « std::flush;
1472
1473     srand(time(NULL));
1474
1475
1476     std::string path_2_electrical_load_time_series =
1477         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
1478
1479     ModelInputs test_model_inputs;
1480     test_model_inputs.path_2_electrical_load_time_series =
1481         path_2_electrical_load_time_series;
1482
1483     Model* test_model_ptr = testConstruct_Model(test_model_inputs);
1484
1485
1486     try {
1487         testBadConstruct_Model();
1488         testPostConstructionAttributes_Model(test_model_ptr);
1489         testElectricalLoadData_Model(test_model_ptr);
1490
1491
1492         int solar_resource_key = 0;
1493         std::string path_2_solar_resource_data =
1494             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
1495
1496         testAddSolarResource_Model(
1497             test_model_ptr,
1498             path_2_solar_resource_data,
1499             solar_resource_key
1500         );
1501
1502
1503         int tidal_resource_key = 1;
1504         std::string path_2_tidal_resource_data =
1505             "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
1506
1507         testAddTidalResource_Model(
1508             test_model_ptr,
1509             path_2_tidal_resource_data,
1510             tidal_resource_key
1511         );
1512
1513
1514         int wave_resource_key = 2;
1515         std::string path_2_wave_resource_data =
1516             "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
1517
1518         testAddWaveResource_Model(
1519             test_model_ptr,
1520             path_2_wave_resource_data,
1521             wave_resource_key
1522         );
1523
1524
1525         int wind_resource_key = 3;
1526         std::string path_2_wind_resource_data =
1527             "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
1528
1529         testAddWindResource_Model(
1530             test_model_ptr,
1531             path_2_wind_resource_data,
1532             wind_resource_key
1533         );
1534
1535
1536         int hydro_resource_key = 4;
1537         std::string path_2_hydro_resource_data =
1538             "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
1539
1540         testAddHydroResource_Model(
1541             test_model_ptr,
1542             path_2_hydro_resource_data,
1543             hydro_resource_key
1544         );
```

```
1545
1546
1547         std::string path_2_normalized_production_time_series =
1548                 "data/test/normalized_production/normalized_solar_production.csv";
1549
1550         // looping solely for the sake of profiling (also tests reset(), which is
1551         // needed for wrapping PGMcpp in an optimizer)
1552         for (int i = 0; i < 1000; i++) {
1553             test_model_ptr->reset();
1554
1555
1556             testAddHydro_Model(test_model_ptr, hydro_resource_key);
1557             testAddDiesel_Model(test_model_ptr);
1558             testAddSolar_Model(test_model_ptr, solar_resource_key);
1559
1560             testAddSolar_productionOverride_Model(
1561                 test_model_ptr,
1562                 path_2_normalized_production_time_series
1563             );
1564
1565             testAddTidal_Model(test_model_ptr, tidal_resource_key);
1566             testAddWave_Model(test_model_ptr, wave_resource_key);
1567             testAddWind_Model(test_model_ptr, wind_resource_key);
1568
1569
1570             test_model_ptr->run();
1571         }
1572
1573
1574         testLoadBalance_Model(test_model_ptr);
1575         testEconomics_Model(test_model_ptr);
1576         testFuelConsumptionEmissions_Model(test_model_ptr);
1577
1578         test_model_ptr->writeResults("test/test_results/");
1579     }
1580
1581
1582     catch (...) {
1583         delete test_model_ptr;
1584
1585         printGold(" ..................................... ");
1586         printRed("FAIL");
1587         std::cout << std::endl;
1588         throw;
1589     }
1590
1591
1592     delete test_model_ptr;
1593
1594     printGold(" ..................................... ");
1595     printGreen("PASS");
1596     std::cout << std::endl;
1597     return 0;
1598 }   /* main() */
```

5.71.2.2 testAddDiesel_Model()

```
void testAddDiesel_Model (
            Model * test_model_ptr )
```

Function to test adding a suite of diesel generators to the test Model object, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

```
893 {
894     DieselInputs diesel_inputs;
895     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
896     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
897
898     test_model_ptr->addDiesel(diesel_inputs);
899
900     testFloatEquals(
```

```
901         test_model_ptr->combustion_ptr_vec.size(),
902         1,
903         __FILE__,
904         __LINE__
905     );
906 
907     testFloatEquals(
908         test_model_ptr->combustion_ptr_vec[0]->type,
909         CombustionType :: DIESEL,
910         __FILE__,
911         __LINE__
912     );
913 
914     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
915 
916     test_model_ptr->addDiesel(diesel_inputs);
917 
918     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
919 
920     test_model_ptr->addDiesel(diesel_inputs);
921 
922     testFloatEquals(
923         test_model_ptr->combustion_ptr_vec.size(),
924         3,
925         __FILE__,
926         __LINE__
927     );
928 
929     std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
930 
931     for (int i = 0; i < 3; i++) {
932         testFloatEquals(
933             test_model_ptr->combustion_ptr_vec[i]->capacity_kW,
934             expected_diesel_capacity_vec_kW[i],
935             __FILE__,
936             __LINE__
937         );
938     }
939 
940     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
941 
942     for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
943         test_model_ptr->addDiesel(diesel_inputs);
944     }
945 
946     return;
947 }   /* testAddDiesel_Model() */
```

5.71.2.3 testAddHydro_Model()

```
void testAddHydro_Model (
              Model * test_model_ptr,
              int hydro_resource_key )
```

Function to test adding a hydroelectric asset to the test Model object, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
hydro_resource_key	A key used to index into the Resources component of the test Model object.

```
843 {
844     HydroInputs hydro_inputs;
845     hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
846     hydro_inputs.reservoir_capacity_m3 = 100000;
847     hydro_inputs.init_reservoir_state = 0.5;
848     hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
849     hydro_inputs.resource_key = hydro_resource_key;
850 
851     test_model_ptr->addHydro(hydro_inputs);
852 
853     testFloatEquals(
854         test_model_ptr->noncombustion_ptr_vec.size(),
```

```
855         1,
856         __FILE__,
857         __LINE__
858     );
859
860     testFloatEquals(
861         test_model_ptr->noncombustion_ptr_vec[0]->type,
862         NoncombustionType :: HYDRO,
863         __FILE__,
864         __LINE__
865     );
866
867     testFloatEquals(
868         test_model_ptr->noncombustion_ptr_vec[0]->resource_key,
869         hydro_resource_key,
870         __FILE__,
871         __LINE__
872     );
873
874     return;
875 }   /* testAddHydro_Model() */
```

5.71.2.4 testAddHydroResource_Model()

```
void testAddHydroResource_Model (
              Model * test_model_ptr,
              std::string path_2_hydro_resource_data,
              int hydro_resource_key )
```

Function to test adding a hydro resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_hydro_resource_data	A path (either relative or absolute) to the hydro resource data.
hydro_resource_key	A key used to index into the Resources component of the test Model object.

```
748 {
749     test_model_ptr->addResource(
750         NoncombustionType :: HYDRO,
751         path_2_hydro_resource_data,
752         hydro_resource_key
753     );
754
755     std::vector<double> expected_hydro_resource_vec_ms = {
756         2167.91531556942,
757         2046.58261560569,
758         2007.85941123153,
759         2000.11477247929,
760         1917.50527264453,
761         1963.97311577093,
762         1908.46985899809,
763         1886.5267112678,
764         1965.26388854254,
765         1953.64692935289,
766         2084.01504296306,
767         2272.46796101188,
768         2520.29645627096,
769         2715.203242423,
770         2720.36633563203,
771         3130.83228077221,
772         3289.59741021591,
773         3981.45195965772,
774         5295.45929491303,
775         7084.47124360523,
776         7709.20557708454,
777         7436.85238642936,
778         7235.49173429668,
779         6710.14695517339,
780         6015.71085806577,
781         5279.97001316337,
```

```
782         4877.24870889801,
783         4421.60569340303,
784         3919.49483690424,
785         3498.70270322341,
786         3274.10813058883,
787         3147.61233529349,
788         2904.94693324343,
789         2805.55738101,
790         2418.32535637171,
791         2398.96375630723,
792         2260.85100182222,
793         2157.58912702878,
794         2019.47637254377,
795         1913.63295220712,
796         1863.29279076589,
797         1748.41395678279,
798         1695.49224555317,
799         1599.97501375715,
800         1559.96103873397,
801         1505.74855473274,
802         1438.62833664765,
803         1384.41585476901
804     };
805
806     for (size_t i = 0; i < expected_hydro_resource_vec_ms.size(); i++) {
807         testFloatEquals(
808             test_model_ptr->resources.resource_map_1D[hydro_resource_key][i],
809             expected_hydro_resource_vec_ms[i],
810             __FILE__,
811             __LINE__
812         );
813     }
814
815     return;
816 }   /* testAddHydroResource_Model() */
```

5.71.2.5 testAddLiIon_Model()

```
void testAddLiIon_Model (
             Model * test_model_ptr )
```

Function to test adding a lithium ion battery energy storage system to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

```
1219 {
1220     LiIonInputs liion_inputs;
1221
1222     test_model_ptr->addLiIon(liion_inputs);
1223
1224     testFloatEquals(
1225         test_model_ptr->storage_ptr_vec.size(),
1226         1,
1227         __FILE__,
1228         __LINE__
1229     );
1230
1231     testFloatEquals(
1232         test_model_ptr->storage_ptr_vec[0]->type,
1233         StorageType :: LIION,
1234         __FILE__,
1235         __LINE__
1236     );
1237
1238     return;
1239 }   /* testAddLiIon_Model() */
```

5.71.2.6 testAddSolar_Model()

```
void testAddSolar_Model (
            Model * test_model_ptr,
            int solar_resource_key )
```

Function to test adding a solar PV array to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
solar_resource_key	A key used to index into the Resources component of the test Model object.

```
974 {
975     SolarInputs solar_inputs;
976     solar_inputs.resource_key = solar_resource_key;
977
978     test_model_ptr->addSolar(solar_inputs);
979
980     testFloatEquals(
981         test_model_ptr->renewable_ptr_vec.size(),
982         1,
983         __FILE__,
984         __LINE__
985     );
986
987     testFloatEquals(
988         test_model_ptr->renewable_ptr_vec[0]->type,
989         RenewableType :: SOLAR,
990         __FILE__,
991         __LINE__
992     );
993
994     return;
995 }   /* testAddSolar_Model() */
```

5.71.2.7 testAddSolar_productionOverride_Model()

```
void testAddSolar_productionOverride_Model (
            Model * test_model_ptr,
            std::string path_2_normalized_production_time_series )
```

Function to test adding a solar PV array to the test Model object using the production override feature, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_normalized_production_time_series	A path (either relative or absolute) to the given normalized production time series data.

```
1022 {
1023     SolarInputs solar_inputs;
1024     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
1025         path_2_normalized_production_time_series;
1026
1027     test_model_ptr->addSolar(solar_inputs);
1028
1029     testFloatEquals(
1030         test_model_ptr->renewable_ptr_vec.size(),
1031         2,
1032         __FILE__,
1033         __LINE__
1034     );
1035
```

```
1036     testFloatEquals(
1037         test_model_ptr->renewable_ptr_vec[1]->type,
1038         RenewableType :: SOLAR,
1039         __FILE__,
1040         __LINE__
1041     );
1042
1043     testTruth(
1044         test_model_ptr->renewable_ptr_vec[1]->normalized_production_series_given,
1045         __FILE__,
1046         __LINE__
1047     );
1048
1049     testTruth(
1050         test_model_ptr->renewable_ptr_vec[1]->path_2_normalized_production_time_series ==
1051         path_2_normalized_production_time_series,
1052         __FILE__,
1053         __LINE__
1054     );
1055
1056     return;
1057 }    /* testAddSolar_productionOverride_Model() */
```

5.71.2.8 testAddSolarResource_Model()

```
void testAddSolarResource_Model (
              Model * test_model_ptr,
              std::string path_2_solar_resource_data,
              int solar_resource_key )
```

Function to test adding a solar resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_solar_resource_data	A path (either relative or absolute) to the solar resource data.
solar_resource_key	A key used to index into the Resources component of the test Model object.

```
290 {
291     test_model_ptr->addResource(
292         RenewableType :: SOLAR,
293         path_2_solar_resource_data,
294         solar_resource_key
295     );
296
297     std::vector<double> expected_solar_resource_vec_kWm2 = {
298         0,
299         0,
300         0,
301         0,
302         0,
303         0,
304         8.51702662684015E-05,
305         0.000348341567045,
306         0.00213793728593,
307         0.004099863613322,
308         0.000997135230553,
309         0.009534527624657,
310         0.022927996790616,
311         0.0136071715294,
312         0.002535134127751,
313         0.005206897515821,
314         0.005627658648597,
315         0.000701186722215,
316         0.00017119827089,
317         0,
318         0,
319         0,
320         0,
321         0,
322         0,
```

```
323         0,
324         0,
325         0,
326         0,
327         0,
328         0,
329         0.000141055102242,
330         0.00084525014743,
331         0.024893647822702,
332         0.091245556190749,
333         0.158722176731637,
334         0.152859680515876,
335         0.149922903895116,
336         0.13049996570866,
337         0.03081254222795,
338         0.001218928911125,
339         0.000206092647423,
340         0,
341         0,
342         0,
343         0,
344         0,
345         0
346     };
347
348     for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
349         testFloatEquals(
350             test_model_ptr->resources.resource_map_1D[solar_resource_key][i],
351             expected_solar_resource_vec_kWm2[i],
352             __FILE__,
353             __LINE__
354         );
355     }
356
357     return;
358 }   /* testAddSolarResource_Model() */
```

5.71.2.9 testAddTidal_Model()

```
void testAddTidal_Model (
             Model * test_model_ptr,
             int tidal_resource_key )
```

Function to test adding a tidal turbine to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
tidal_resource_key	A key used to index into the Resources component of the test Model object.

```
1084 {
1085     TidalInputs tidal_inputs;
1086     tidal_inputs.resource_key = tidal_resource_key;
1087
1088     test_model_ptr->addTidal(tidal_inputs);
1089
1090     testFloatEquals(
1091         test_model_ptr->renewable_ptr_vec.size(),
1092         3,
1093         __FILE__,
1094         __LINE__
1095     );
1096
1097     testFloatEquals(
1098         test_model_ptr->renewable_ptr_vec[2]->type,
1099         RenewableType :: TIDAL,
1100         __FILE__,
1101         __LINE__
1102     );
1103
1104     return;
1105 }   /* testAddTidal_Model() */
```

5.71.2.10 testAddTidalResource_Model()

```
void testAddTidalResource_Model (
            Model * test_model_ptr,
            std::string path_2_tidal_resource_data,
            int tidal_resource_key )
```

Function to test adding a tidal resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_tidal_resource_data	A path (either relative or absolute) to the tidal resource data.
tidal_resource_key	A key used to index into the Resources component of the test Model object.

```
390 {
391     test_model_ptr->addResource(
392         RenewableType :: TIDAL,
393         path_2_tidal_resource_data,
394         tidal_resource_key
395     );
396
397     std::vector<double> expected_tidal_resource_vec_ms = {
398         0.347439913040533,
399         0.770545522195602,
400         0.731352084836198,
401         0.293389814389542,
402         0.209959110813115,
403         0.610609623896497,
404         1.78067162013604,
405         2.53522775118089,
406         2.75966627832024,
407         2.52101111143895,
408         2.05389330201031,
409         1.3461515862445,
410         0.28909254878384,
411         0.897754086048563,
412         1.71406453837407,
413         1.85047408742869,
414         1.71507908595979,
415         1.33540349705416,
416         0.434586143463003,
417         0.500623815700637,
418         1.37172172646733,
419         1.68294125491228,
420         1.56101300975417,
421         1.04925834219412,
422         0.211395463930223,
423         1.03720048903385,
424         1.85059536356448,
425         1.85203242794517,
426         1.4091471616277,
427         0.767776539039899,
428         0.251464906990961,
429         1.47018469375652,
430         2.36260493698197,
431         2.46653750048625,
432         2.12851908739291,
433         1.62783753197988,
434         0.734594890957439,
435         0.441886297300355,
436         1.6574418350918,
437         2.0684558286637,
438         1.87717416992136,
439         1.58871262337931,
440         1.03451227609235,
441         0.193371305159817,
442         0.976400122458815,
443         1.6583227369707,
444         1.76690616570953,
445         1.54801328553115
446     };
```

```
447
448     for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
449         testFloatEquals(
450             test_model_ptr->resources.resource_map_1D[tidal_resource_key][i],
451             expected_tidal_resource_vec_ms[i],
452             __FILE__,
453             __LINE__
454         );
455     }
456
457     return;
458 }   /* testAddTidalResource_Model() */
```

5.71.2.11 testAddWave_Model()

```
void testAddWave_Model (
              Model * test_model_ptr,
              int wave_resource_key )
```

Function to test adding a wave energy converter to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
wave_resource_key	A key used to index into the Resources component of the test Model object.

```
1132 {
1133     WaveInputs wave_inputs;
1134     wave_inputs.resource_key = wave_resource_key;
1135
1136     test_model_ptr->addWave(wave_inputs);
1137
1138     testFloatEquals(
1139         test_model_ptr->renewable_ptr_vec.size(),
1140         4,
1141         __FILE__,
1142         __LINE__
1143     );
1144
1145     testFloatEquals(
1146         test_model_ptr->renewable_ptr_vec[3]->type,
1147         RenewableType :: WAVE,
1148         __FILE__,
1149         __LINE__
1150     );
1151
1152     return;
1153 }   /* testAddWave_Model() */
```

5.71.2.12 testAddWaveResource_Model()

```
void testAddWaveResource_Model (
              Model * test_model_ptr,
              std::string path_2_wave_resource_data,
              int wave_resource_key )
```

Function to test adding a wave resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_wave_resource_data	A path (either relative or absolute) to the wave resource data.
wave_resource_key	A key used to index into the Resources component of the test Model object.

```
490 {
491     test_model_ptr->addResource(
492         RenewableType :: WAVE,
493         path_2_wave_resource_data,
494         wave_resource_key
495     );
496 
497     std::vector<double> expected_significant_wave_height_vec_m = {
498         4.26175222125028,
499         4.25020976167872,
500         4.25656524330349,
501         4.27193854786718,
502         4.28744955711233,
503         4.29421815278154,
504         4.2839937266082,
505         4.25716982457976,
506         4.22419391611483,
507         4.19588925217606,
508         4.17338788587412,
509         4.14672746914214,
510         4.10560041173665,
511         4.05074966447193,
512         3.9953696962433,
513         3.95316976150866,
514         3.92771018142378,
515         3.91129562488595,
516         3.89558312094911,
517         3.87861093931749,
518         3.86538307240754,
519         3.86108961027929,
520         3.86459448853189,
521         3.86796474016882,
522         3.86357412779993,
523         3.85554872014731,
524         3.86044266668675,
525         3.89445961915999,
526         3.95554798115731,
527         4.02265508610476,
528         4.07419587011404,
529         4.10314247143958,
530         4.11738045085928,
531         4.12554995596708,
532         4.12923992001675,
533         4.1229292327442,
534         4.10123955307441,
535         4.06748827895363,
536         4.0336230651344,
537         4.01134236393876,
538         4.00136570034559,
539         3.99368787690411,
540         3.97820924247644,
541         3.95369335178055,
542         3.92742545608532,
543         3.90683362771686,
544         3.89331520944006,
545         3.88256045801583
546     };
547 
548     std::vector<double> expected_energy_period_vec_s = {
549         10.4456008226821,
550         10.4614151137651,
551         10.4462827795433,
552         10.4127692097884,
553         10.3734397942723,
554         10.3408599227669,
555         10.32637292093,
556         10.3245412676322,
557         10.310409818185,
558         10.2589529840966,
559         10.1728100603103,
560         10.0862908658929,
561         10.03480243813,
562         10.023673635806,
563         10.0243418565116,
564         10.0063487117653,
565         9.96050302286607,
566         9.9011999635568,
567         9.84451822125472,
```

```
568         9.79726875879626,
569         9.75614594835158,
570         9.7173447961368,
571         9.68342904390577,
572         9.66380508567062,
573         9.6674009575699,
574         9.68927134575103,
575         9.70979984863046,
576         9.70967357906908,
577         9.68983025704562,
578         9.6722855524805,
579         9.67973599910003,
580         9.71977125328293,
581         9.78450442291421,
582         9.86532355233449,
583         9.96158937600019,
584         10.0807018356507,
585         10.2291022504937,
586         10.39458528356,
587         10.5464393581004,
588         10.6553277500484,
589         10.7245553190084,
590         10.7893127285064,
591         10.8846512240849,
592         11.0148158739075,
593         11.1544325654719,
594         11.2772785848343,
595         11.3744362756187,
596         11.4533643503183
597     };
598
599     for (size_t i = 0; i < expected_energy_period_vec_s.size(); i++) {
600         testFloatEquals(
601             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][0],
602             expected_significant_wave_height_vec_m[i],
603             __FILE__,
604             __LINE__
605         );
606
607         testFloatEquals(
608             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][1],
609             expected_energy_period_vec_s[i],
610             __FILE__,
611             __LINE__
612         );
613     }
614
615     return;
616 }   /* testAddWaveResource_Model() */
```

5.71.2.13 testAddWind_Model()

```
void testAddWind_Model (
              Model * test_model_ptr,
              int wind_resource_key )
```

Function to test adding a wind turbine to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
wind_resource_key	A key used to index into the Resources component of the test Model object.

```
1180 {
1181     WindInputs wind_inputs;
1182     wind_inputs.resource_key = wind_resource_key;
1183
1184     test_model_ptr->addWind(wind_inputs);
1185
1186     testFloatEquals(
1187         test_model_ptr->renewable_ptr_vec.size(),
1188         5,
1189         __FILE__,
```

```
1190         __LINE__
1191     );
1192
1193     testFloatEquals(
1194         test_model_ptr->renewable_ptr_vec[4]->type,
1195         RenewableType :: WIND,
1196         __FILE__,
1197         __LINE__
1198     );
1199
1200     return;
1201 }   /* testAddWind_Model() */
```

5.71.2.14 testAddWindResource_Model()

```
void testAddWindResource_Model (
             Model * test_model_ptr,
             std::string path_2_wind_resource_data,
             int wind_resource_key )
```

Function to test adding a wind resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_wind_resource_data	A path (either relative or absolute) to the wind resource data.
wind_resource_key	A key used to index into the Resources component of the test Model object.

```
648 {
649     test_model_ptr->addResource(
650         RenewableType :: WIND,
651         path_2_wind_resource_data,
652         wind_resource_key
653     );
654
655     std::vector<double> expected_wind_resource_vec_ms = {
656         6.88566688469997,
657         5.02177105466549,
658         3.74211715899568,
659         5.67169579985362,
660         4.90670669971858,
661         4.29586955031368,
662         7.41155377205065,
663         10.2243290476943,
664         13.1258696725555,
665         13.7016198628274,
666         16.2481482330233,
667         16.5096744355418,
668         13.4354482206162,
669         14.0129230731609,
670         14.5554549260515,
671         13.4454539065912,
672         13.3447169512094,
673         11.7372615098554,
674         12.7200070078013,
675         10.6421127908149,
676         6.09869498990661,
677         5.66355596602321,
678         4.97316966910831,
679         3.48937138360567,
680         2.15917470979169,
681         1.29061103587027,
682         3.43475751425219,
683         4.11706326260927,
684         4.28905275747408,
685         5.75850263196241,
686         8.98293663055264,
687         11.7069822941315,
688         12.4031987075858,
689         15.4096570910089,
690         16.6210843829552,
```

```
691         13.3421219142573,
692         15.2112831900548,
693         18.350864533037,
694         15.8751799822971,
695         15.3921198799796,
696         15.9729192868434,
697         12.4728950178772,
698         10.177050481096,
699         10.7342247355551,
700         8.98846695631389,
701         4.14671169124739,
702         3.17256452697149,
703         3.40036336968628
704     };
705
706     for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
707         testFloatEquals(
708             test_model_ptr->resources.resource_map_1D[wind_resource_key][i],
709             expected_wind_resource_vec_ms[i],
710             __FILE__,
711             __LINE__
712         );
713     }
714
715     return;
716 }   /* testAddWindResource_Model() */
```

5.71.2.15 testBadConstruct_Model()

```
void testBadConstruct_Model (
              void  )
```

Function to check if passing bad ModelInputs to the Model constructor is handled appropriately.

```
66 {
67     bool error_flag = true;
68
69     try {
70         ModelInputs bad_model_inputs;   // path_2_electrical_load_time_series left empty
71
72         Model bad_model(bad_model_inputs);
73
74         error_flag = false;
75     } catch (...) {
76         // Task failed successfully! =P
77     }
78     if (not error_flag) {
79         expectedErrorNotDetected(__FILE__, __LINE__);
80     }
81
82     try {
83         ModelInputs bad_model_inputs;
84         bad_model_inputs.path_2_electrical_load_time_series =
85             "data/test/electrical_load/bad_path_";
86         bad_model_inputs.path_2_electrical_load_time_series += std::to_string(rand());
87         bad_model_inputs.path_2_electrical_load_time_series += ".csv";
88
89         Model bad_model(bad_model_inputs);
90
91         error_flag = false;
92     } catch (...) {
93         // Task failed successfully! =P
94     }
95     if (not error_flag) {
96         expectedErrorNotDetected(__FILE__, __LINE__);
97     }
98
99     return;
100 }
```

5.71.2.16 testConstruct_Model()

```
Model* testConstruct_Model (
            ModelInputs test_model_inputs )
39 {
40     Model* test_model_ptr = new Model(test_model_inputs);
41
42     testTruth(
43         test_model_ptr->electrical_load.path_2_electrical_load_time_series ==
44         test_model_inputs.path_2_electrical_load_time_series,
45         __FILE__,
46         __LINE__
47     );
48
49     return test_model_ptr;
50 }   /* testConstruct_Model() */
```

5.71.2.17 testEconomics_Model()

```
void testEconomics_Model (
            Model * test_model_ptr )
```

Function to check that the modelled economic metrics are > 0.

Parameters

test_model_ptr	A pointer to the test Model object.

```
1372 {
1373     testGreaterThan(
1374         test_model_ptr->net_present_cost,
1375         0,
1376         __FILE__,
1377         __LINE__
1378     );
1379
1380     testGreaterThan(
1381         test_model_ptr->levellized_cost_of_energy_kWh,
1382         0,
1383         __FILE__,
1384         __LINE__
1385     );
1386
1387     return;
1388 }   /* testEconomics_Model() */
```

5.71.2.18 testElectricalLoadData_Model()

```
void testElectricalLoadData_Model (
            Model * test_model_ptr )
```

Function to check the values read into the ElectricalLoad component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.

```
173 {
174     std::vector<double> expected_dt_vec_hrs (48, 1);
175
```

```
176     std::vector<double> expected_time_vec_hrs = {
177          0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
178         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
179         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
180         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
181     };
182
183     std::vector<double> expected_load_vec_kW = {
184         360.253836463674,
185         355.171277826775,
186         353.776453532298,
187         353.75405737934,
188         346.592867404975,
189         340.132411175118,
190         337.354867340578,
191         340.644115618736,
192         363.639028500678,
193         378.787797779238,
194         372.215798201712,
195         395.093925731298,
196         402.325427142659,
197         386.907725462306,
198         380.709170928091,
199         372.062070914977,
200         372.328646856954,
201         391.841444284136,
202         394.029351759596,
203         383.369407765254,
204         381.093099675206,
205         382.604158946193,
206         390.744843709034,
207         383.13949492437,
208         368.150393976985,
209         364.629744480226,
210         363.572736804082,
211         359.854924202248,
212         355.207590170267,
213         349.094656012401,
214         354.365935871597,
215         343.380608328546,
216         404.673065729266,
217         486.296896820126,
218         480.225974100847,
219         457.318764401085,
220         418.177339948609,
221         414.399018364126,
222         409.678420185754,
223         404.768766016563,
224         401.699589920585,
225         402.44339040654,
226         398.138372541906,
227         396.010498627646,
228         390.165117432277,
229         375.850429417013,
230         365.567100746484,
231         365.429624610923
232     };
233
234     for (int i = 0; i < 48; i++) {
235         testFloatEquals(
236             test_model_ptr->electrical_load.dt_vec_hrs[i],
237             expected_dt_vec_hrs[i],
238             __FILE__,
239             __LINE__
240         );
241
242         testFloatEquals(
243             test_model_ptr->electrical_load.time_vec_hrs[i],
244             expected_time_vec_hrs[i],
245             __FILE__,
246             __LINE__
247         );
248
249         testFloatEquals(
250             test_model_ptr->electrical_load.load_vec_kW[i],
251             expected_load_vec_kW[i],
252             __FILE__,
253             __LINE__
254         );
255     }
256
257     return;
258 }   /* testElectricalLoadData_Model() */
```

5.71.2.19 testFuelConsumptionEmissions_Model()

```
void testFuelConsumptionEmissions_Model (
            Model * test_model_ptr )
```

Function to check that the modelled fuel consumption and emissions are > 0.

Parameters

test_model_ptr	A pointer to the test Model object.

```
1405 {
1406     testGreaterThan(
1407         test_model_ptr->total_fuel_consumed_L,
1408         0,
1409         __FILE__,
1410         __LINE__
1411     );
1412
1413     testGreaterThan(
1414         test_model_ptr->total_emissions.CO2_kg,
1415         0,
1416         __FILE__,
1417         __LINE__
1418     );
1419
1420     testGreaterThan(
1421         test_model_ptr->total_emissions.CO_kg,
1422         0,
1423         __FILE__,
1424         __LINE__
1425     );
1426
1427     testGreaterThan(
1428         test_model_ptr->total_emissions.NOx_kg,
1429         0,
1430         __FILE__,
1431         __LINE__
1432     );
1433
1434     testGreaterThan(
1435         test_model_ptr->total_emissions.SOx_kg,
1436         0,
1437         __FILE__,
1438         __LINE__
1439     );
1440
1441     testGreaterThan(
1442         test_model_ptr->total_emissions.CH4_kg,
1443         0,
1444         __FILE__,
1445         __LINE__
1446     );
1447
1448     testGreaterThan(
1449         test_model_ptr->total_emissions.PM_kg,
1450         0,
1451         __FILE__,
1452         __LINE__
1453     );
1454
1455     return;
1456 }   /* testFuelConsumptionEmissions_Model() */
```

5.71.2.20 testLoadBalance_Model()

```
void testLoadBalance_Model (
            Model * test_model_ptr )
```

Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the Controller as expected.

Parameters

test_model_ptr	A pointer to the test Model object.

```
1258 {
1259     double net_load_kW = 0;
1260 
1261     Combustion* combustion_ptr;
1262     Noncombustion* noncombustion_ptr;
1263     Renewable* renewable_ptr;
1264     Storage* storage_ptr;
1265 
1266     for (int i = 0; i < test_model_ptr->electrical_load.n_points; i++) {
1267         net_load_kW = test_model_ptr->controller.net_load_vec_kW[i];
1268 
1269         testLessThanOrEqualTo(
1270             test_model_ptr->controller.net_load_vec_kW[i],
1271             test_model_ptr->electrical_load.max_load_kW,
1272             __FILE__,
1273             __LINE__
1274         );
1275 
1276         for (size_t j = 0; j < test_model_ptr->combustion_ptr_vec.size(); j++) {
1277             combustion_ptr = test_model_ptr->combustion_ptr_vec[j];
1278 
1279             testFloatEquals(
1280                 combustion_ptr->production_vec_kW[i] -
1281                 combustion_ptr->dispatch_vec_kW[i] -
1282                 combustion_ptr->curtailment_vec_kW[i] -
1283                 combustion_ptr->storage_vec_kW[i],
1284                 0,
1285                 __FILE__,
1286                 __LINE__
1287             );
1288 
1289             net_load_kW -= combustion_ptr->production_vec_kW[i];
1290         }
1291 
1292         for (size_t j = 0; j < test_model_ptr->noncombustion_ptr_vec.size(); j++) {
1293             noncombustion_ptr = test_model_ptr->noncombustion_ptr_vec[j];
1294 
1295             testFloatEquals(
1296                 noncombustion_ptr->production_vec_kW[i] -
1297                 noncombustion_ptr->dispatch_vec_kW[i] -
1298                 noncombustion_ptr->curtailment_vec_kW[i] -
1299                 noncombustion_ptr->storage_vec_kW[i],
1300                 0,
1301                 __FILE__,
1302                 __LINE__
1303             );
1304 
1305             net_load_kW -= noncombustion_ptr->production_vec_kW[i];
1306         }
1307 
1308         for (size_t j = 0; j < test_model_ptr->renewable_ptr_vec.size(); j++) {
1309             renewable_ptr = test_model_ptr->renewable_ptr_vec[j];
1310 
1311             testFloatEquals(
1312                 renewable_ptr->production_vec_kW[i] -
1313                 renewable_ptr->dispatch_vec_kW[i] -
1314                 renewable_ptr->curtailment_vec_kW[i] -
1315                 renewable_ptr->storage_vec_kW[i],
1316                 0,
1317                 __FILE__,
1318                 __LINE__
1319             );
1320 
1321             net_load_kW -= renewable_ptr->production_vec_kW[i];
1322         }
1323 
1324         for (size_t j = 0; j < test_model_ptr->storage_ptr_vec.size(); j++) {
1325             storage_ptr = test_model_ptr->storage_ptr_vec[j];
1326 
1327             testTruth(
1328                 not (
1329                     storage_ptr->charging_power_vec_kW[i] > 0 and
1330                     storage_ptr->discharging_power_vec_kW[i] > 0
1331                 ),
1332                 __FILE__,
1333                 __LINE__
1334             );
1335 
1336             net_load_kW -= storage_ptr->discharging_power_vec_kW[i];
1337         }
1338 
1339         testLessThanOrEqualTo(
```

```
1340             net_load_kW,
1341             0,
1342             __FILE__,
1343             __LINE__
1344         );
1345     }
1346
1347     testFloatEquals(
1348         test_model_ptr->total_dispatch_discharge_kWh,
1349         2263351.62026685,
1350         __FILE__,
1351         __LINE__
1352     );
1353
1354     return;
1355 }   /* testLoadBalance_Model() */
```

5.71.2.21 testPostConstructionAttributes_Model()

```
void testPostConstructionAttributes_Model (
            Model * test_model_ptr )
```

A function to check the values of various post-construction attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

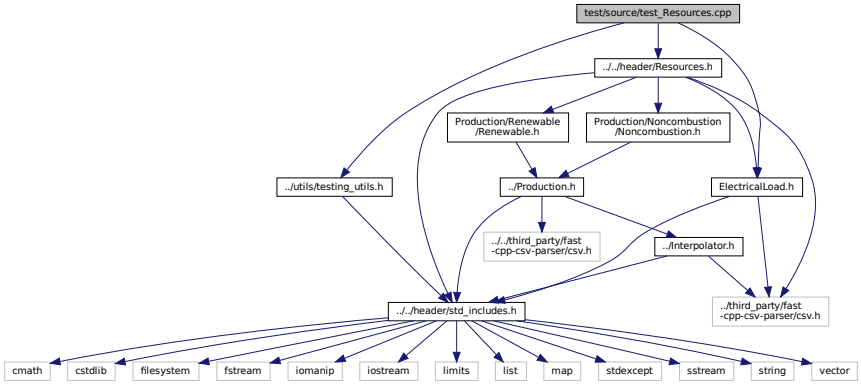
```
117 {
118     testFloatEquals(
119         test_model_ptr->electrical_load.n_points,
120         8760,
121         __FILE__,
122         __LINE__
123     );
124
125     testFloatEquals(
126         test_model_ptr->electrical_load.n_years,
127         0.999886,
128         __FILE__,
129         __LINE__
130     );
131
132     testFloatEquals(
133         test_model_ptr->electrical_load.min_load_kW,
134         82.1211213927802,
135         __FILE__,
136         __LINE__
137     );
138
139     testFloatEquals(
140         test_model_ptr->electrical_load.mean_load_kW,
141         258.373472633202,
142         __FILE__,
143         __LINE__
144     );
145
146
147     testFloatEquals(
148         test_model_ptr->electrical_load.max_load_kW,
149         500,
150         __FILE__,
151         __LINE__
152     );
153
154     return;
155 }   /* testPostConstructionAttributes_Model() */
```

5.72 test/source/test_Resources.cpp File Reference

Testing suite for Resources class.

```
#include "../utils/testing_utils.h"
#include "../../header/Resources.h"
#include "../../header/ElectricalLoad.h"
```

Include dependency graph for test_Resources.cpp:



Functions

- Resources * testConstruct_Resources (void)

  A function to construct a Resources object and spot check some post-construction attributes.
- void testAddSolarResource_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

  Function to test adding a solar resource and then check the values read into the test Resources object.
- void testBadAdd_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

  Function to test that trying to add bad resource data is being handled as expected.
- void testAddTidalResource_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_tidal_resource_data, int tidal_resource_key)

  Function to test adding a tidal resource and then check the values read into the test Resources object.
- void testAddWaveResource_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_wave_resource_data, int wave_resource_key)

  Function to test adding a wave resource and then check the values read into the test Resources object.
- void testAddWindResource_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_wind_resource_data, int wind_resource_key)

  Function to test adding a wind resource and then check the values read into the test Resources object.
- void testAddHydroResource_Resources (Resources *test_resources_ptr, ElectricalLoad *test_electrical_load_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)

  Function to test adding a hydro resource and then check the values read into the test Resources object.
- int main (int argc, char **argv)

5.72.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

5.72.2 Function Documentation

5.72.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
758 {
759     #ifdef _WIN32
760         activateVirtualTerminal();
761     #endif  /* _WIN32 */
762 
763     printGold("\tTesting Resources");
764 
765     srand(time(NULL));
766 
767 
768     std::string path_2_electrical_load_time_series =
769         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
770 
771     ElectricalLoad* test_electrical_load_ptr =
772         new ElectricalLoad(path_2_electrical_load_time_series);
773 
774     Resources* test_resources_ptr = testConstruct_Resources();
775 
776 
777     try {
778         int solar_resource_key = 0;
779         std::string path_2_solar_resource_data =
780             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
781 
782         testAddSolarResource_Resources(
783             test_resources_ptr,
784             test_electrical_load_ptr,
785             path_2_solar_resource_data,
786             solar_resource_key
787         );
788 
789         testBadAdd_Resources(
790             test_resources_ptr,
791             test_electrical_load_ptr,
792             path_2_solar_resource_data,
793             solar_resource_key
794         );
795 
796 
797         int tidal_resource_key = 1;
798         std::string path_2_tidal_resource_data =
799             "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
800 
801         testAddTidalResource_Resources(
802             test_resources_ptr,
803             test_electrical_load_ptr,
804             path_2_tidal_resource_data,
805             tidal_resource_key
806         );
807 
808 
809         int wave_resource_key = 2;
810         std::string path_2_wave_resource_data =
811             "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
812 
813         testAddWaveResource_Resources(
814             test_resources_ptr,
815             test_electrical_load_ptr,
816             path_2_wave_resource_data,
817             wave_resource_key
818         );
819 
820 
821         int wind_resource_key = 3;
822         std::string path_2_wind_resource_data =
823             "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
824 
825         testAddWindResource_Resources(
826             test_resources_ptr,
827             test_electrical_load_ptr,
828             path_2_wind_resource_data,
```

```
829             wind_resource_key
830         );
831
832
833         int hydro_resource_key = 4;
834         std::string path_2_hydro_resource_data =
835             "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
836
837         testAddHydroResource_Resources(
838             test_resources_ptr,
839             test_electrical_load_ptr,
840             path_2_hydro_resource_data,
841             hydro_resource_key
842         );
843     }
844
845
846     catch (...) {
847         delete test_electrical_load_ptr;
848         delete test_resources_ptr;
849
850         printGold(" ............................... ");
851         printRed("FAIL");
852         std::cout « std::endl;
853         throw;
854     }
855
856
857     delete test_electrical_load_ptr;
858     delete test_resources_ptr;
859
860     printGold(" ............................... ");
861     printGreen("PASS");
862     std::cout « std::endl;
863     return 0;
864 }   /* main() */
```

5.72.2.2 testAddHydroResource_Resources()

```
void testAddHydroResource_Resources (
              Resources * test_resources_ptr,
              ElectricalLoad * test_electrical_load_ptr,
              std::string path_2_hydro_resource_data,
              int hydro_resource_key )
```

Function to test adding a hydro resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_hydro_resource_data	A path (either relative or absolute) to the hydro resource data.
hydro_resource_key	A key used to index into the Resources component of the test Resources object.

```
680 {
681     test_resources_ptr->addResource(
682         NoncombustionType::HYDRO,
683         path_2_hydro_resource_data,
684         hydro_resource_key,
685         test_electrical_load_ptr
686     );
687
688     std::vector<double> expected_hydro_resource_vec_m3hr = {
689         2167.91531556942,
690         2046.58261560569,
691         2007.85941123153,
692         2000.11477247929,
693         1917.50527264453,
694         1963.97311577093,
695         1908.46985899809,
696         1886.5267112678,
```

```
697         1965.26388854254,
698         1953.64692935289,
699         2084.01504296306,
700         2272.46796101188,
701         2520.29645627096,
702         2715.203242423,
703         2720.36633563203,
704         3130.83228077221,
705         3289.59741021591,
706         3981.45195965772,
707         5295.45929491303,
708         7084.47124360523,
709         7709.20557708454,
710         7436.85238642936,
711         7235.49173429668,
712         6710.14695517339,
713         6015.71085806577,
714         5279.97001316337,
715         4877.24870889801,
716         4421.60569340303,
717         3919.49483690424,
718         3498.70270322341,
719         3274.10813058883,
720         3147.61233529349,
721         2904.94693324343,
722         2805.55738101,
723         2418.32535637171,
724         2398.96375630723,
725         2260.85100182222,
726         2157.58912702878,
727         2019.47637254377,
728         1913.63295220712,
729         1863.29279076589,
730         1748.41395678279,
731         1695.49224555317,
732         1599.97501375715,
733         1559.96103873397,
734         1505.74855473274,
735         1438.62833664765,
736         1384.41585476901
737     };
738
739     for (size_t i = 0; i < expected_hydro_resource_vec_m3hr.size(); i++) {
740         testFloatEquals(
741             test_resources_ptr->resource_map_1D[hydro_resource_key][i],
742             expected_hydro_resource_vec_m3hr[i],
743             __FILE__,
744             __LINE__
745         );
746     }
747
748     return;
749 }   /* testAddHydroResource_Resources() */
```

5.72.2.3 testAddSolarResource_Resources()

```
void testAddSolarResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_solar_resource_data,
            int solar_resource_key )
```

Function to test adding a solar resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_solar_resource_data	A path (either relative or absolute) to the solar resource data.
solar_resource_key	A key used to index into the Resources component of the test Resources object.

```
107 {
108     test_resources_ptr->addResource(
109         RenewableType::SOLAR,
110         path_2_solar_resource_data,
111         solar_resource_key,
112         test_electrical_load_ptr
113     );
114 
115     std::vector<double> expected_solar_resource_vec_kWm2 = {
116         0,
117         0,
118         0,
119         0,
120         0,
121         0,
122         8.51702662684015E-05,
123         0.000348341567045,
124         0.00213793728593,
125         0.004099863613322,
126         0.000997135230553,
127         0.009534527624657,
128         0.022927996790616,
129         0.0136071715294,
130         0.002535134127751,
131         0.005206897515821,
132         0.005627658648597,
133         0.000701186722215,
134         0.00017119827089,
135         0,
136         0,
137         0,
138         0,
139         0,
140         0,
141         0,
142         0,
143         0,
144         0,
145         0,
146         0,
147         0.000141055102242,
148         0.00084525014743,
149         0.024893647822702,
150         0.091245556190749,
151         0.158722176731637,
152         0.152859680515876,
153         0.149922903895116,
154         0.13049996570866,
155         0.03081254222795,
156         0.001218928911125,
157         0.000206092647423,
158         0,
159         0,
160         0,
161         0,
162         0,
163         0
164     };
165 
166     for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
167         testFloatEquals(
168             test_resources_ptr->resource_map_1D[solar_resource_key][i],
169             expected_solar_resource_vec_kWm2[i],
170             __FILE__,
171             __LINE__
172         );
173     }
174 
175     return;
176 }   /* testAddSolarResource_Resources() */
```

5.72.2.4 testAddTidalResource_Resources()

```
void testAddTidalResource_Resources (
             Resources * test_resources_ptr,
             ElectricalLoad * test_electrical_load_ptr,
             std::string path_2_tidal_resource_data,
             int tidal_resource_key )
```

Function to test adding a tidal resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_tidal_resource_data	A path (either relative or absolute) to the tidal resource data.
tidal_resource_key	A key used to index into the Resources component of the test Resources object.

```
307 {
308     test_resources_ptr->addResource(
309         RenewableType::TIDAL,
310         path_2_tidal_resource_data,
311         tidal_resource_key,
312         test_electrical_load_ptr
313     );
314 
315     std::vector<double> expected_tidal_resource_vec_ms = {
316         0.347439913040533,
317         0.770545522195602,
318         0.731352084836198,
319         0.293389814389542,
320         0.209959110813115,
321         0.610609623896497,
322         1.78067162013604,
323         2.53522775118089,
324         2.75966627832024,
325         2.52101111143895,
326         2.05389330201031,
327         1.3461515862445,
328         0.28909254878384,
329         0.897754086048563,
330         1.71406453837407,
331         1.85047408742869,
332         1.71507908595979,
333         1.33540349705416,
334         0.434586143463003,
335         0.500623815700637,
336         1.37172172646733,
337         1.68294125491228,
338         1.56101300975417,
339         1.04925834219412,
340         0.211395463930223,
341         1.03720048903385,
342         1.85059536356448,
343         1.85203242794517,
344         1.4091471616277,
345         0.767776539039899,
346         0.251464906990961,
347         1.47018469375652,
348         2.36260493698197,
349         2.46653750048625,
350         2.12851908739291,
351         1.62783753197988,
352         0.734594890957439,
353         0.441886297300355,
354         1.6574418350918,
355         2.0684558286637,
356         1.87717416992136,
357         1.58871262337931,
358         1.03451227609235,
359         0.193371305159817,
360         0.976400122458815,
361         1.6583227369707,
362         1.76690616570953,
363         1.54801328553115
364     };
365 
366     for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
367         testFloatEquals(
368             test_resources_ptr->resource_map_1D[tidal_resource_key][i],
369             expected_tidal_resource_vec_ms[i],
370             __FILE__,
371             __LINE__
372         );
373     }
374 
375     return;
376 }   /* testAddTidalResource_Resources() */
```

5.72.2.5 testAddWaveResource_Resources()

```
void testAddWaveResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_wave_resource_data,
            int wave_resource_key )
```

Function to test adding a wave resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_wave_resource_data	A path (either relative or absolute) to the wave resource data.
wave_resource_key	A key used to index into the Resources component of the test Resources object.

```
412 {
413     test_resources_ptr->addResource(
414         RenewableType::WAVE,
415         path_2_wave_resource_data,
416         wave_resource_key,
417         test_electrical_load_ptr
418     );
419 
420     std::vector<double> expected_significant_wave_height_vec_m = {
421         4.26175222125028,
422         4.25020976167872,
423         4.25656524330349,
424         4.27193854786718,
425         4.28744955711233,
426         4.29421815278154,
427         4.2839937266082,
428         4.25716982457976,
429         4.22419391611483,
430         4.19588925217606,
431         4.17338788587412,
432         4.14672746914214,
433         4.10560041173665,
434         4.05074966447193,
435         3.9953696962433,
436         3.95316976150866,
437         3.92771018142378,
438         3.91129562488595,
439         3.89558312094911,
440         3.87861093931749,
441         3.86538307240754,
442         3.86108961027929,
443         3.86459448853189,
444         3.86796474016882,
445         3.86357412779993,
446         3.85554872014731,
447         3.86044266668675,
448         3.89445961915999,
449         3.95554798115731,
450         4.02265508610476,
451         4.07419587011404,
452         4.10314247143958,
453         4.11738045085928,
454         4.12554995596708,
455         4.12923992001675,
456         4.1229292327442,
457         4.10123955307441,
458         4.06748827895363,
459         4.0336230651344,
460         4.01134236393876,
461         4.00136570034559,
462         3.99368787690411,
463         3.97820924247644,
464         3.95369335178055,
465         3.92742545608532,
466         3.90683362771686,
467         3.89331520944006,
468         3.88256045801583
469     };
470 
471     std::vector<double> expected_energy_period_vec_s = {
```

```
472         10.4456008226821,
473         10.4614151137651,
474         10.4462827795433,
475         10.4127692097884,
476         10.3734397942723,
477         10.3408599227669,
478         10.32637292093,
479         10.3245412676322,
480         10.310409818185,
481         10.2589529840966,
482         10.1728100603103,
483         10.0862908658929,
484         10.03480243813,
485         10.023673635806,
486         10.0243418565116,
487         10.0063487117653,
488         9.96050302286607,
489         9.9011999635568,
490         9.84451822125472,
491         9.79726875879626,
492         9.75614594835158,
493         9.7173447961368,
494         9.68342904390577,
495         9.66380508567062,
496         9.6674009575699,
497         9.68927134575103,
498         9.70979984863046,
499         9.70967357906908,
500         9.68983025704562,
501         9.6722855524805,
502         9.67973599910003,
503         9.71977125328293,
504         9.78450442291421,
505         9.86532355233449,
506         9.96158937600019,
507         10.0807018356507,
508         10.2291022504937,
509         10.39458528356,
510         10.5464393581004,
511         10.6553277500484,
512         10.7245553190084,
513         10.7893127285064,
514         10.8846512240849,
515         11.0148158739075,
516         11.1544325654719,
517         11.2772785848343,
518         11.3744362756187,
519         11.4533643503183
520     };
521
522     for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
523         testFloatEquals(
524             test_resources_ptr->resource_map_2D[wave_resource_key][i][0],
525             expected_significant_wave_height_vec_m[i],
526             __FILE__,
527             __LINE__
528         );
529
530         testFloatEquals(
531             test_resources_ptr->resource_map_2D[wave_resource_key][i][1],
532             expected_energy_period_vec_s[i],
533             __FILE__,
534             __LINE__
535         );
536     }
537
538     return;
539 }   /* testAddWaveResource_Resources() */
```

5.72.2.6 testAddWindResource_Resources()

```
void testAddWindResource_Resources (
              Resources * test_resources_ptr,
              ElectricalLoad * test_electrical_load_ptr,
              std::string path_2_wind_resource_data,
              int wind_resource_key )
```

Function to test adding a wind resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_wind_resource_data	A path (either relative or absolute) to the wind resource data.
wind_resource_key	A key used to index into the Resources component of the test Resources object.

```
575 {
576     test_resources_ptr->addResource(
577         RenewableType::WIND,
578         path_2_wind_resource_data,
579         wind_resource_key,
580         test_electrical_load_ptr
581     );
582 
583     std::vector<double> expected_wind_resource_vec_ms = {
584         6.88566688469997,
585         5.02177105466549,
586         3.74211715899568,
587         5.67169579985362,
588         4.90670669971858,
589         4.29586955031368,
590         7.41155377205065,
591         10.2243290476943,
592         13.1258696725555,
593         13.7016198628274,
594         16.2481482330233,
595         16.5096744355418,
596         13.4354482206162,
597         14.0129230731609,
598         14.5554549260515,
599         13.4454539065912,
600         13.3447169512094,
601         11.7372615098554,
602         12.7200070078013,
603         10.6421127908149,
604         6.09869498990661,
605         5.66355596602321,
606         4.97316966910831,
607         3.48937138360567,
608         2.15917470979169,
609         1.29061103587027,
610         3.43475751425219,
611         4.11706326260927,
612         4.28905275747408,
613         5.75850263196241,
614         8.98293663055264,
615         11.7069822941315,
616         12.4031987075858,
617         15.4096570910089,
618         16.6210843829552,
619         13.3421219142573,
620         15.2112831900548,
621         18.350864533037,
622         15.8751799822971,
623         15.3921198799796,
624         15.9729192868434,
625         12.4728950178772,
626         10.177050481096,
627         10.7342247355551,
628         8.98846695631389,
629         4.14671169124739,
630         3.17256452697149,
631         3.40036336968628
632     };
633 
634     for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
635         testFloatEquals(
636             test_resources_ptr->resource_map_1D[wind_resource_key][i],
637             expected_wind_resource_vec_ms[i],
638             __FILE__,
639             __LINE__
640         );
641     }
642 
643     return;
644 }   /* testAddWindResource_Resources() */
```

5.72.2.7 testBadAdd_Resources()

```
void testBadAdd_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_solar_resource_data,
            int solar_resource_key )
```

Function to test that trying to add bad resource data is being handled as expected.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_solar_resource_data	A path (either relative or absolute) to the given solar resource data.
solar_resource_key	A key for indexing into the test Resources object.

```
211 {
212     bool error_flag = true;
213 
214     try {
215         test_resources_ptr->addResource(
216             RenewableType::SOLAR,
217             path_2_solar_resource_data,
218             solar_resource_key,
219             test_electrical_load_ptr
220         );
221 
222         error_flag = false;
223     } catch (...) {
224         // Task failed successfully! =P
225     }
226     if (not error_flag) {
227         expectedErrorNotDetected(__FILE__, __LINE__);
228     }
229 
230 
231     try {
232         std::string path_2_solar_resource_data_BAD_TIMES =
233             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";
234 
235         test_resources_ptr->addResource(
236             RenewableType::SOLAR,
237             path_2_solar_resource_data_BAD_TIMES,
238             -1,
239             test_electrical_load_ptr
240         );
241 
242         error_flag = false;
243     } catch (...) {
244         // Task failed successfully! =P
245     }
246     if (not error_flag) {
247         expectedErrorNotDetected(__FILE__, __LINE__);
248     }
249 
250 
251     try {
252         std::string path_2_solar_resource_data_BAD_LENGTH =
253             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";
254 
255         test_resources_ptr->addResource(
256             RenewableType::SOLAR,
257             path_2_solar_resource_data_BAD_LENGTH,
258             -2,
259             test_electrical_load_ptr
260         );
261 
262         error_flag = false;
263     } catch (...) {
264         // Task failed successfully! =P
265     }
266     if (not error_flag) {
267         expectedErrorNotDetected(__FILE__, __LINE__);
268     }
269 
270     return;
```

```
271 }   /* testBadAdd_Resources() */
```

5.72.2.8 testConstruct_Resources()

```
Resources * testConstruct_Resources (
            void  )
```

A function to construct a Resources object and spot check some post-construction attributes.

Returns

A pointer to a test Resources object.

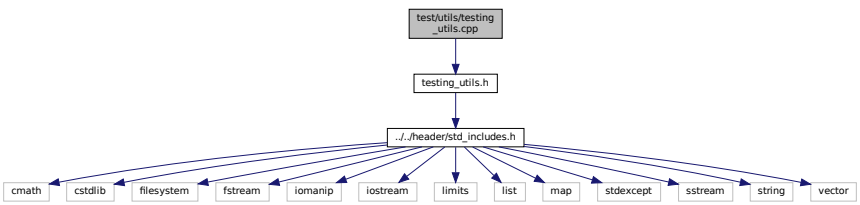
```
39 {
40     Resources* test_resources_ptr = new Resources();
41
42     testFloatEquals(
43         test_resources_ptr->resource_map_1D.size(),
44         0,
45         __FILE__,
46         __LINE__
47     );
48
49     testFloatEquals(
50         test_resources_ptr->path_map_1D.size(),
51         0,
52         __FILE__,
53         __LINE__
54     );
55
56     testFloatEquals(
57         test_resources_ptr->resource_map_2D.size(),
58         0,
59         __FILE__,
60         __LINE__
61     );
62
63     testFloatEquals(
64         test_resources_ptr->path_map_2D.size(),
65         0,
66         __FILE__,
67         __LINE__
68     );
69
70     return test_resources_ptr;
71 }   /* testConstruct_Resources() */
```

5.73 test/utils/testing_utils.cpp File Reference

Implementation file for various PGMcpp testing utilities.

```
#include "testing_utils.h"
```

Include dependency graph for testing_utils.cpp:



Functions

- void printGreen (std::string input_str)

  A function that sends green text to std::cout.
- void printGold (std::string input_str)

  A function that sends gold text to std::cout.
- void printRed (std::string input_str)

  A function that sends red text to std::cout.
- void testFloatEquals (double x, double y, std::string file, int line)

  Tests for the equality of two floating point numbers x and y (to within FLOAT_TOLERANCE).
- void testGreaterThan (double x, double y, std::string file, int line)

  Tests if x > y.
- void testGreaterThanOrEqualTo (double x, double y, std::string file, int line)

  Tests if x >= y.
- void testLessThan (double x, double y, std::string file, int line)

  Tests if x < y.
- void testLessThanOrEqualTo (double x, double y, std::string file, int line)

  Tests if x <= y.
- void testTruth (bool statement, std::string file, int line)

  Tests if the given statement is true.
- void expectedErrorNotDetected (std::string file, int line)

  A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.73.1 Detailed Description

Implementation file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.73.2 Function Documentation

5.73.2.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
            std::string file,
            int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
432 {
433     std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
434     error_str += std::to_string(line);
```

```
435     error_str += " of ";
436     error_str += file;
437
438     #ifdef _WIN32
439         std::cout « error_str « std::endl;
440     #endif
441
442     throw std::runtime_error(error_str);
443     return;
444 }   /* expectedErrorNotDetected() */
```

5.73.2.2 printGold()

```
void printGold (
            std::string input_str )
```

A function that sends gold text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
84 {
85     std::cout « "\x1B[33m" « input_str « "\033[0m";
86     return;
87 }   /* printGold() */
```

5.73.2.3 printGreen()

```
void printGreen (
            std::string input_str )
```

A function that sends green text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
64 {
65     std::cout « "\x1B[32m" « input_str « "\033[0m";
66     return;
67 }   /* printGreen() */
```

5.73.2.4 printRed()

```
void printRed (
            std::string input_str )
```

A function that sends red text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
104 {
105     std::cout « "\x1B[31m" « input_str « "\033[0m";
106     return;
107 }   /* printRed() */
```

5.73.2.5 testFloatEquals()

```
void testFloatEquals (
             double x,
             double y,
             std::string file,
             int line )
```

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT_TOLERANCE).

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142 
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154 
155     #ifdef _WIN32
156         std::cout « error_str « std::endl;
157     #endif
158 
159     throw std::runtime_error(error_str);
160     return;
161 }   /* testFloatEquals() */
```

5.73.2.6 testGreaterThan()

```
void testGreaterThan (
             double x,
             double y,
             std::string file,
             int line )
```

Tests if x $>$ y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout « error_str « std::endl;
208     #endif
209
210     throw std::runtime_error(error_str);
211     return;
212 }   /* testGreaterThan() */
```

5.73.2.7 testGreaterThanOrEqualTo()

```
void testGreaterThanOrEqualTo (
              double x,
              double y,
              std::string file,
              int line )
```

Tests if x >= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
242 {
243     if (x >= y) {
244         return;
245     }
246
247     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
248     error_str += file;
249     error_str += "\tline ";
250     error_str += std::to_string(line);
251     error_str += ":\t\n";
252     error_str += std::to_string(x);
253     error_str += " is not greater than or equal to ";
254     error_str += std::to_string(y);
255     error_str += "\n";
256
257     #ifdef _WIN32
258         std::cout « error_str « std::endl;
259     #endif
260
261     throw std::runtime_error(error_str);
```

```
262     return;
263 }   /* testGreaterThanOrEqualTo() */
```

5.73.2.8 testLessThan()

```
void testLessThan (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x < y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
293 {
294     if (x < y) {
295         return;
296     }
297 
298     std::string error_str = "ERROR: testLessThan():\t in ";
299     error_str += file;
300     error_str += "\tline ";
301     error_str += std::to_string(line);
302     error_str += ":\t\n";
303     error_str += std::to_string(x);
304     error_str += " is not less than ";
305     error_str += std::to_string(y);
306     error_str += "\n";
307 
308     #ifdef _WIN32
309         std::cout « error_str « std::endl;
310     #endif
311 
312     throw std::runtime_error(error_str);
313     return;
314 }   /* testLessThan() */
```

5.73.2.9 testLessThanOrEqualTo()

```
void testLessThanOrEqualTo (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x <= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
344 {
345     if (x <= y) {
346         return;
347     }
348 
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358 
359     #ifdef _WIN32
360         std::cout « error_str « std::endl;
361     #endif
362 
363     throw std::runtime_error(error_str);
364     return;
365 }   /* testLessThanOrEqualTo() */
```

5.73.2.10 testTruth()

```
void testTruth (
              bool statement,
              std::string file,
              int line )
```

Tests if the given statement is true.

Parameters

statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

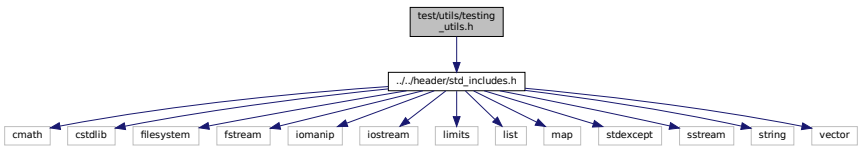
```
392 {
393     if (statement) {
394         return;
395     }
396 
397     std::string error_str = "ERROR: testTruth():\t in ";
398     error_str += file;
399     error_str += "\tline ";
400     error_str += std::to_string(line);
401     error_str += ":\t\n";
402     error_str += "Given statement is not true";
403 
404     #ifdef _WIN32
405         std::cout « error_str « std::endl;
406     #endif
407 
408     throw std::runtime_error(error_str);
409     return;
410 }   /* testTruth() */
```

5.74 test/utils/testing_utils.h File Reference

Header file for various PGMcpp testing utilities.

```
#include "../../header/std_includes.h"
```

Include dependency graph for testing_utils.h:



This graph shows which files directly or indirectly include this file:

Macros

- #define FLOAT_TOLERANCE 1e-6

  A tolerance for application to floating point equality tests.

Functions

- void printGreen (std::string)

  A function that sends green text to std::cout.
- void printGold (std::string)

  A function that sends gold text to std::cout.
- void printRed (std::string)

  A function that sends red text to std::cout.
- void testFloatEquals (double, double, std::string, int)

  Tests for the equality of two floating point numbers x and y (to within FLOAT_TOLERANCE).
- void testGreaterThan (double, double, std::string, int)

  Tests if x > y.
- void testGreaterThanOrEqualTo (double, double, std::string, int)

  Tests if x >= y.
- void testLessThan (double, double, std::string, int)

  Tests if x < y.
- void testLessThanOrEqualTo (double, double, std::string, int)

  Tests if x <= y.
- void testTruth (bool, std::string, int)

  Tests if the given statement is true.
- void expectedErrorNotDetected (std::string, int)

  A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.74.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.74.2 Macro Definition Documentation

5.74.2.1 FLOAT_TOLERANCE

```
#define FLOAT_TOLERANCE 1e-6
```

A tolerance for application to floating point equality tests.

5.74.3 Function Documentation

5.74.3.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
            std::string file,
            int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
432 {
433     std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
434     error_str += std::to_string(line);
435     error_str += " of ";
436     error_str += file;
437 
438     #ifdef _WIN32
439         std::cout « error_str « std::endl;
440     #endif
441 
442     throw std::runtime_error(error_str);
443     return;
444 }   /* expectedErrorNotDetected() */
```

5.74.3.2 printGold()

```
void printGold (
            std::string input_str )
```

A function that sends gold text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
84 {
85     std::cout « "\x1B[33m" « input_str « "\033[0m";
86     return;
87 }   /* printGold() */
```

5.74.3.3 printGreen()

```
void printGreen (
             std::string input_str )
```

A function that sends green text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
64 {
65     std::cout « "\x1B[32m" « input_str « "\033[0m";
66     return;
67 }   /* printGreen() */
```

5.74.3.4 printRed()

```
void printRed (
             std::string input_str )
```

A function that sends red text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
104 {
105     std::cout « "\x1B[31m" « input_str « "\033[0m";
106     return;
107 }   /* printRed() */
```

5.74.3.5 testFloatEquals()

```
void testFloatEquals (
             double x,
             double y,
             std::string file,
             int line )
```

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT_TOLERANCE).

Parameters

x	The first of two numbers to test.

Parameters

y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142 
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154 
155     #ifdef _WIN32
156         std::cout << error_str << std::endl;
157     #endif
158 
159     throw std::runtime_error(error_str);
160     return;
161 }   /* testFloatEquals() */
```

5.74.3.6 testGreaterThan()

```
void testGreaterThan (
              double x,
              double y,
              std::string file,
              int line )
```

Tests if x > y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
191 {
192     if (x > y) {
193         return;
194     }
195 
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205 
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209 
```

```
210     throw std::runtime_error(error_str);
211     return;
212 }   /* testGreaterThan() */
```

5.74.3.7 testGreaterThanOrEqualTo()

```
void testGreaterThanOrEqualTo (
             double x,
             double y,
             std::string file,
             int line )
```

Tests if x >= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
242 {
243     if (x >= y) {
244         return;
245     }
246
247     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
248     error_str += file;
249     error_str += "\tline ";
250     error_str += std::to_string(line);
251     error_str += ":\t\n";
252     error_str += std::to_string(x);
253     error_str += " is not greater than or equal to ";
254     error_str += std::to_string(y);
255     error_str += "\n";
256
257     #ifdef _WIN32
258         std::cout << error_str << std::endl;
259     #endif
260
261     throw std::runtime_error(error_str);
262     return;
263 }   /* testGreaterThanOrEqualTo() */
```

5.74.3.8 testLessThan()

```
void testLessThan (
             double x,
             double y,
             std::string file,
             int line )
```

Tests if x < y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
293 {
294     if (x < y) {
295         return;
296     }
297 
298     std::string error_str = "ERROR: testLessThan():\t in ";
299     error_str += file;
300     error_str += "\tline ";
301     error_str += std::to_string(line);
302     error_str += ":\t\n";
303     error_str += std::to_string(x);
304     error_str += " is not less than ";
305     error_str += std::to_string(y);
306     error_str += "\n";
307 
308     #ifdef _WIN32
309         std::cout « error_str « std::endl;
310     #endif
311 
312     throw std::runtime_error(error_str);
313     return;
314 }   /* testLessThan() */
```

5.74.3.9 testLessThanOrEqualTo()

```
void testLessThanOrEqualTo (
              double x,
              double y,
              std::string file,
              int line )
```

Tests if x <= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
344 {
345     if (x <= y) {
346         return;
347     }
348 
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358 
359     #ifdef _WIN32
360         std::cout « error_str « std::endl;
361     #endif
362 
363     throw std::runtime_error(error_str);
364     return;
365 }   /* testLessThanOrEqualTo() */
```

5.74.3.10 testTruth()

```
void testTruth (
```

```
            bool statement,
            std::string file,
            int line )
```

Tests if the given statement is true.

Parameters

statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
392 {
393     if (statement) {
394         return;
395     }
396
397     std::string error_str = "ERROR: testTruth():\t in ";
398     error_str += file;
399     error_str += "\tline ";
400     error_str += std::to_string(line);
401     error_str += ":\t\n";
402     error_str += "Given statement is not true";
403
404     #ifdef _WIN32
405         std::cout « error_str « std::endl;
406     #endif
407
408     throw std::runtime_error(error_str);
409     return;
410 }   /* testTruth() */
```

Bibliography

G.S. Bir, M.J. Lawson, and Y. Li. Structural Design of a Horizontal-Axis Tidal Current Turbine Composite Blade. *NREL*, 2011. URL https://www.researchgate.net/publication/239886961_Structural_Design_of_a_Horizontal-Axis_Tidal_Current_Turbine_Composite_Blade. 233, 234

Dr. B. Buckham, Dr. C. Crawford, Dr. I. Beya Marshall, and Dr. B. Whitby. Wei Wai Kum Tidal Prefeasibility Study - Tidal Resource Assessment. Technical report, PRIMED, 2023. Internal: P2202E_BRKLYG+WEI WAI KUM_R01_V20230613v3. 234

CIMAC. Guide to Diesel Exhaust Emissions Control of NOx, SOx, Particulates, Smoke, and CO2. Technical report, Conseil International des Machines à Combustion, 2008. Included: docs/refs/diesel_emissions_ref_2.pdf. 60

HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 171, 220

HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 16, 158, 171, 172, 219, 220

HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 51, 60

HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 51, 60

HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 51, 60

HOMER. How HOMER Calculates the PV Array Power Output, 2023f. URL https://www.homerenergy.com/products/pro/docs/latest/how_homer_calculates_the_pv_array_power_output.html. 210

HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 171, 220

HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 172, 219

HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 171, 220

W. Jakob. pybind11 — Seamless operability between C++11 and Python, 2023. URL https://pybind11.readthedocs.io/en/stable/. 305, 306, 308, 311, 314, 316, 321, 322, 324, 326, 328, 330, 332, 333, 335, 336, 338, 341

M. Lewis, R.O. Murray, S. Fredriksson, J. Maskell, A. de Fockert, S.P. Neill, and P.E. Robins. A standardised tidal-stream power curve, optimised for the global resource. *Renewable Energy*, 2021. doi: 10.1016/j.renene.2021.02.032. URL https://www.researchgate.net/publication/349341552_A_standardised_tidal-stream_power_curve_optimised_for_the_global_resource. 233, 234

Dr. S.L. MacDougall. Commercial Potential of Marine Renewables in British Columbia. Technical report, S.L. MacDougall Research & Consulting, 2019. Submitted to Natural Resources Canada. 236, 251

P. Milan, M. Wächter, S. Barth, and J. Peinke. Power curves for wind turbines. *Wind Power Generation and Wind Turbine Design*, page 595–612, 2010. doi: $10.2495/978\text{-}1\text{-}84564\text{-}205\text{-}1/18$. 265

NRCan. Auto$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 60

Dr. B. Robertson, Dr. H. Bailey, M. Leary, and Dr. B. Buckham. A methodology for architecture agnostic and time flexible representations of wave energy converter performance. *Applied Energy*, 287, 2021. doi:10.1016/j.apenergy.2021.116588. 250

A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023a. Included: docs/refs/battery_degradation.pdf. 116, 118, 129

A. Truelove. Hydro Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023b. Included: docs/refs/hydro.pdf. 75, 77, 78, 79, 81

A. Truelove, Dr. B. Buckham, Dr. C. Crawford, and C. Hiles. Scaling Technology Models for HOMER Pro: Wind, Tidal Stream, and Wave. Technical report, PRIMED, 2019. Included: docs/refs/wind_tidal_wave.pdf. 234, 249, 266

D. van Heesch. Doxygen: Generate documentation from source code, 2023. URL https://www.doxygen.nl. 278

B. Whitby and C.E. Ugalde-Loo. Performance of Pitch and Stall Regulated Tidal Stream Turbines. *IEEE Transactions on Sustainable Energy*, 5(1), 2013. doi: $10.1109/\mathrm{TSTE}.2013.2272653$. 234

U. Zafar. Literature Review of Wind Turbines. *Bauhaus Universität*, 2018. URL https://www.researchgate.net/publication/329680977_Literature_Review_of_Wind_Turbines. 265

Index