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

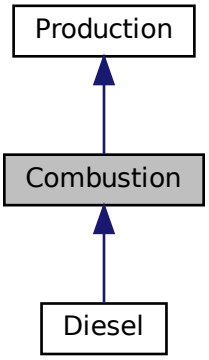
Class Documentation

4.1 Combustion Class Reference

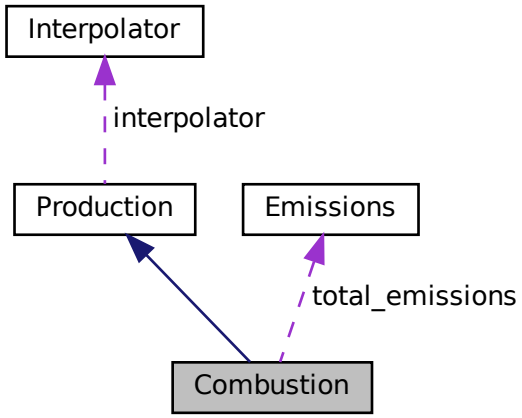
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

`#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)

  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.
- 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.
- Emissions total_emissions

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

```
63 {
64     return;
65 }   /* Combustion() */
```

4.1.2.2 Combustion() [2/2]

```
Combustion::Combustion (
            int n_points,
            double n_years,
            CombustionInputs combustion_inputs )
```

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.

```
93   :
94 Production(
95     n_points,
96     n_years,
97     combustion_inputs.production_inputs
98 )
99 {
100     //  1. check inputs
101     this->__checkInputs(combustion_inputs);
102
```

```
103     //  2. set attributes
104     this->fuel_cost_L = 0;
105     this->nominal_fuel_escalation_annual =
106         combustion_inputs.nominal_fuel_escalation_annual;
107
108     this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
109         combustion_inputs.nominal_fuel_escalation_annual,
110         combustion_inputs.production_inputs.nominal_discount_annual
111     );
112
113     this->linear_fuel_slope_LkWh = 0;
114     this->linear_fuel_intercept_LkWh = 0;
115
116     this->CO2_emissions_intensity_kgL = 0;
117     this->CO_emissions_intensity_kgL = 0;
118     this->NOx_emissions_intensity_kgL = 0;
119     this->SOx_emissions_intensity_kgL = 0;
120     this->CH4_emissions_intensity_kgL = 0;
121     this->PM_emissions_intensity_kgL = 0;
122
123     this->total_fuel_consumed_L = 0;
124
125     this->fuel_consumption_vec_L.resize(this->n_points, 0);
126     this->fuel_cost_vec.resize(this->n_points, 0);
127
128     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
129     this->CO_emissions_vec_kg.resize(this->n_points, 0);
130     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
131     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
132     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
133     this->PM_emissions_vec_kg.resize(this->n_points, 0);
134
135     //  3. construction print
136     if (this->print_flag) {
137         std::cout « "Combustion object constructed at " « this « std::endl;
138     }
139
140     return;
141 }   /* Combustion() */
```

4.1.2.3 ~Combustion()

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

Destructor for the Combustion class.

```
448 {
449     //  1. destruction print
450     if (this->print_flag) {
451         std::cout « "Combustion object at " « this « " destroyed" « std::endl;
452     }
453
454     return;
455 }   /* ~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     // ...
42 
43     return;
44 }   /* __checkInputs() */
```

4.1.3.2 __writeSummary()

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

Reimplemented in Diesel.

```
87 {return;}
```

4.1.3.3 __writeTimeSeries()

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

Reimplemented in Diesel.

```
92          {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.

```
271 {
272     //  1. invoke base class method
273     load_kW = Production :: commit(
274         timestep,
275         dt_hrs,
276         production_kW,
277         load_kW
278     );
279 
280 
281     if (this->is_running) {
282         //  2. compute and record fuel consumption
283         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
284         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
285 
286         //  3. compute and record emissions
287         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
288         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
289         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
290         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
291         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
292         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
293         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
294 
295         //  4. incur fuel costs
296         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
297     }
298 
299     return load_kW;
300 }   /* 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.

```
215 {
216     //  1. account for fuel costs in net present cost
217     double t_hrs = 0;
218     double real_fuel_escalation_scalar = 0;
219 
220     for (int i = 0; i < this->n_points; i++) {
221         t_hrs = time_vec_hrs_ptr->at(i);
222 
223         real_fuel_escalation_scalar = 1.0 / pow(
224             1 + this->real_fuel_escalation_annual,
225             t_hrs / 8760
226         );
227 
228         this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
229     }
```

```
230
231     //  2. invoke base class method
232     Production :: computeEconomics(time_vec_hrs_ptr);
233
234     return;
235 }   /* computeEconomics() */
```

4.1.3.6 computeFuelAndEmissions()

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

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

```
183 {
184     for (int i = 0; i < n_points; i++) {
185         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
186
187         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
188         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
189         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
190         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
191         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
192         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
193     }
194
195     return;
196 }   /* 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.

```
348                                                         {
349     Emissions emissions;
350
351     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
352     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
353     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
354     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
355     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
356     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
357
358     return emissions;
359 }   /* 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].

```
322 {
323     double fuel_consumed_L = (
324         this->linear_fuel_slope_LkWh * production_kW +
325         this->linear_fuel_intercept_LkWh * this->capacity_kW
326     ) * dt_hrs;
327 
328     return fuel_consumed_L;
329 } /* 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.

```
159 {
160     //  1. reset attributes
161     //...
162 
163     //  2. invoke base class method
164     Production :: handleReplacement(timestep);
165 
166     return;
167 }   /* __handleReplacement() */
```

4.1.3.10 requestProductionkW()

```
virtual double Combustion::requestProductionkW (
            int ,
```

```
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Diesel.

```
135 {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.

```
395 {
396     //  1. handle sentinel
397     if (max_lines < 0) {
398         max_lines = this->n_points;
399     }
400 
401     //  2. create subdirectories
402     write_path += "Production/";
403     if (not std::filesystem::is_directory(write_path)) {
404         std::filesystem::create_directory(write_path);
405     }
406 
407     write_path += "Combustion/";
408     if (not std::filesystem::is_directory(write_path)) {
409         std::filesystem::create_directory(write_path);
410     }
411 
412     write_path += this->type_str;
413     write_path += "_";
414     write_path += std::to_string(int(ceil(this->capacity_kW)));
415     write_path += "kW_idx";
416     write_path += std::to_string(combustion_index);
417     write_path += "/";
418     std::filesystem::create_directory(write_path);
419 
420     //  3. write summary
421     this->__writeSummary(write_path);
422 
423     //  4. write time series
424     if (max_lines > this->n_points) {
425         max_lines = this->n_points;
426     }
427 
428     if (max_lines > 0) {
429         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
430     }
431 
432     return;
433 }   /* 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 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.

4.1.4.11 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.

4.1.4.12 nominal_fuel_escalation_annual

```
double Combustion::nominal_fuel_escalation_annual
```

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

4.1.4.13 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.14 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.

4.1.4.15 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.16 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.

4.1.4.17 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.

4.1.4.18 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.19 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.

4.1.4.20 total_emissions

```
Emissions Combustion::total_emissions
```

An Emissions structure for holding total emissions [kg].

4.1.4.21 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

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

4.1.4.22 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

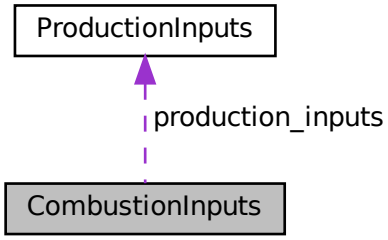
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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.
- double nominal_fuel_escalation_annual = 0.05

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

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 nominal_fuel_escalation_annual

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

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

4.2.2.2 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 *, std::vector< Combustion * > *, 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 *, std::vector< Combustion * > *, 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 *, std::vector< Combustion * > *, 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 *, std::vector< Combustion * > *, 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 *, std::vector< Combustion * > *, 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< Renewable * > *)

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

```
1000 {
1001     return;
1002 }   /* Controller() */
```

4.3.2.2 ~Controller()

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

Destructor for the Controller class.

```
1229 {
1230     this->clear();
1231 
1232     return;
1233 }   /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

```
void Controller::__applyCycleChargingControl_CHARGING (
            int timestep,
            ElectricalLoad * electrical_load_ptr,
            std::vector< Combustion * > * combustion_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.
combustion_ptr_vec_ptr	A pointer to the Combustion 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.

```
403 {
404     //  1. default to load following
405     this->__applyLoadFollowingControl_CHARGING(
406         timestep,
407         electrical_load_ptr,
408         combustion_ptr_vec_ptr,
409         renewable_ptr_vec_ptr,
410         storage_ptr_vec_ptr
411     );
412 
413     return;
414 }   /* __applyCycleChargingControl_CHARGING() */
```

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

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

```
        std::vector< Combustion * > * combustion_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.
combustion_ptr_vec_ptr	A pointer to the Combustion 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

```
453 {
454     //  1. get dt_hrs, net load
455     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
456     double net_load_kW = this->net_load_vec_kW[timestep];
457 
458     //  2. partition Storage assets into depleted and non-depleted
459     std::list<Storage*> depleted_storage_ptr_list;
460     std::list<Storage*> nondepleted_storage_ptr_list;
461 
462     Storage* storage_ptr;
463     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
464         storage_ptr = storage_ptr_vec_ptr->at(i);
465 
466         if (storage_ptr->is_depleted) {
467             depleted_storage_ptr_list.push_back(storage_ptr);
468         }
469 
470         else {
471             nondepleted_storage_ptr_list.push_back(storage_ptr);
472         }
473     }
474 
475     //  3. discharge non-depleted storage assets
476     net_load_kW = this->__handleStorageDischarging(
477         timestep,
478         dt_hrs,
479         net_load_kW,
480         nondepleted_storage_ptr_list
481     );
482 
483     //  4. request optimal production from all Combustion assets
484     //     default to load following if no depleted storage
485     if (depleted_storage_ptr_list.empty()) {
486         net_load_kW = this->__handleCombustionDispatch(
487             timestep,
488             dt_hrs,
489             net_load_kW,
490             combustion_ptr_vec_ptr,
491             false   // is_cycle_charging
492         );
493     }
494 
495     else {
496         net_load_kW = this->__handleCombustionDispatch(
497             timestep,
498             dt_hrs,
499             net_load_kW,
500             combustion_ptr_vec_ptr,
501             true    // is_cycle_charging
502         );
503     }
504 
505     //  5. attempt to charge depleted Storage assets using any and all available
507     //     charge priority is Combustion, then Renewable
508     this->__handleStorageCharging(
509         timestep,
510         dt_hrs,
511         depleted_storage_ptr_list,
512         combustion_ptr_vec_ptr,
513         renewable_ptr_vec_ptr
514     );
```

```
515
516     //  6. record any missed load
517     if (net_load_kW > 1e-6) {
518         this->missed_load_vec_kW[timestep] = net_load_kW;
519     }
520
521     return;
522 }   /* __applyCycleChargingControl_DISCHARGING() */
```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```
void Controller::__applyLoadFollowingControl_CHARGING (
            int timestep,
            ElectricalLoad * electrical_load_ptr,
            std::vector< Combustion * > * combustion_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.
combustion_ptr_vec_ptr	A pointer to the Combustion 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.

```
246 {
247     //  1. get dt_hrs, set net load
248     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
249     double net_load_kW = 0;
250
251     //  2. request zero production from all Combustion assets
252     this->__handleCombustionDispatch(
253         timestep,
254         dt_hrs,
255         net_load_kW,
256         combustion_ptr_vec_ptr,
257         false   // is_cycle_charging
258     );
259
260     //  3. attempt to charge all Storage assets using any and all available curtailment
261     //     charge priority is Combustion, then Renewable
262     this->__handleStorageCharging(
263         timestep,
264         dt_hrs,
265         storage_ptr_vec_ptr,
266         combustion_ptr_vec_ptr,
267         renewable_ptr_vec_ptr
268     );
269
270     return;
271 }   /* __applyLoadFollowingControl_CHARGING() */
```

4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

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

```
        std::vector< Combustion * > * combustion_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.
combustion_ptr_vec_ptr	A pointer to the Combustion 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

```
309 {
310     //  1. get dt_hrs, net load
311     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
312     double net_load_kW = this->net_load_vec_kW[timestep];
313 
314     //  2. partition Storage assets into depleted and non-depleted
315     std::list<Storage*> depleted_storage_ptr_list;
316     std::list<Storage*> nondepleted_storage_ptr_list;
317 
318     Storage* storage_ptr;
319     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
320         storage_ptr = storage_ptr_vec_ptr->at(i);
321 
322         if (storage_ptr->is_depleted) {
323             depleted_storage_ptr_list.push_back(storage_ptr);
324         }
325 
326         else {
327             nondepleted_storage_ptr_list.push_back(storage_ptr);
328         }
329     }
330 
331     //  3. discharge non-depleted storage assets
332     net_load_kW = this->__handleStorageDischarging(
333         timestep,
334         dt_hrs,
335         net_load_kW,
336         nondepleted_storage_ptr_list
337     );
338 
339     //  4. request optimal production from all Combustion assets
340     net_load_kW = this->__handleCombustionDispatch(
341         timestep,
342         dt_hrs,
343         net_load_kW,
344         combustion_ptr_vec_ptr,
345         false   // is_cycle_charging
346     );
347 
348     //  5. attempt to charge depleted Storage assets using any and all available
350     //     charge priority is Combustion, then Renewable
351     this->__handleStorageCharging(
352         timestep,
353         dt_hrs,
354         depleted_storage_ptr_list,
355         combustion_ptr_vec_ptr,
356         renewable_ptr_vec_ptr
357     );
358 
359     //  6. record any missed load
360     if (net_load_kW > 1e-6) {
361         this->missed_load_vec_kW[timestep] = net_load_kW;
362     }
363 
364     return;
365 }   /* __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.

```
759 {
760     double production_kW = 0;
761
762     switch (renewable_ptr->type) {
763         case (RenewableType :: SOLAR): {
764             production_kW = renewable_ptr->computeProductionkW(
765                 timestep,
766                 dt_hrs,
767                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
768             );
769
770             break;
771         }
772
773         case (RenewableType :: TIDAL): {
774             production_kW = renewable_ptr->computeProductionkW(
775                 timestep,
776                 dt_hrs,
777                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
778             );
779
780             break;
781         }
782
783         case (RenewableType :: WAVE): {
784             production_kW = renewable_ptr->computeProductionkW(
785                 timestep,
786                 dt_hrs,
787                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0],
788                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1]
789             );
790
791             break;
792         }
793
794         case (RenewableType :: WIND): {
795             production_kW = renewable_ptr->computeProductionkW(
796                 timestep,
797                 dt_hrs,
798                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
```

```
799             );
800
801             break;
802         }
803
804         default: {
805             std::string error_str = "ERROR:  Controller::__getRenewableProduction():  ";
806             error_str += "renewable type ";
807             error_str += std::to_string(renewable_ptr->type);
808             error_str += " not recognized";
809
810             #ifdef _WIN32
811                 std::cout « error_str « std::endl;
812             #endif
813
814             throw std::runtime_error(error_str);
815
816             break;
817         }
818     }
819
820     return production_kW;
821 }   /* __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.

```
863 {
864     //  1. get minimal Combustion dispatch
865     double target_production_kW = 1.2 * net_load_kW;
866     double total_capacity_kW = 0;
867
868     std::map<double, std::vector<bool»::iterator iter = this->combustion_map.begin();
869     while (iter != std::prev(this->combustion_map.end(), 1)) {
870         if (target_production_kW <= total_capacity_kW) {
871             break;
872         }
873
874         iter++;
875         total_capacity_kW = iter->first;
```

```
876     }
877
878     //  2. share load proportionally (by rated capacity) over active diesels
879     Combustion* combustion_ptr;
880     double production_kW = 0;
881     double request_kW = 0;
882     double _net_load_kW = net_load_kW;
883
884     for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
885         combustion_ptr = combustion_ptr_vec_ptr->at(i);
886
887         if (total_capacity_kW > 0) {
888             request_kW =
889                 int(this->combustion_map[total_capacity_kW][i]) *
890                 net_load_kW *
891                 (combustion_ptr->capacity_kW / total_capacity_kW);
892         }
893
894         else {
895             request_kW = 0;
896         }
897
898         if (is_cycle_charging and request_kW > 0) {
899             if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
900                 request_kW = 0.85 * combustion_ptr->capacity_kW;
901             }
902         }
903
904         production_kW = combustion_ptr->requestProductionkW(
905             timestep,
906             dt_hrs,
907             request_kW
908         );
909
910         _net_load_kW = combustion_ptr->commit(
911             timestep,
912             dt_hrs,
913             production_kW,
914             _net_load_kW
915         );
916     }
917
918     return _net_load_kW;
919 }   /* __handleCombustionDispatch() */
```

4.3.3.9 __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< 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.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.

```
560 {
561     double acceptable_kW = 0;
562     double curtailment_kW = 0;
563
564     Storage* storage_ptr;
```

```
565     Combustion* combustion_ptr;
566     Renewable* renewable_ptr;
567
568     std::list<Storage*>::iterator iter;
569     for (
570         iter = storage_ptr_list.begin();
571         iter != storage_ptr_list.end();
572         iter++
573     ){
574         storage_ptr = (*iter);
575
576         //  1. attempt to charge from Combustion curtailment first
577         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
578             combustion_ptr = combustion_ptr_vec_ptr->at(i);
579             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
580
581             if (curtailment_kW <= 0) {
582                 continue;
583             }
584
585             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
586
587             if (acceptable_kW > curtailment_kW) {
588                 acceptable_kW = curtailment_kW;
589             }
590
591             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
592             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
593             storage_ptr->power_kW += acceptable_kW;
594         }
595
596         //  2. attempt to charge from Renewable curtailment second
597         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
598             renewable_ptr = renewable_ptr_vec_ptr->at(i);
599             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
600
601             if (curtailment_kW <= 0) {
602                 continue;
603             }
604
605             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
606
607             if (acceptable_kW > curtailment_kW) {
608                 acceptable_kW = curtailment_kW;
609             }
610
611             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
612             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
613             storage_ptr->power_kW += acceptable_kW;
614         }
615
616         //  3. commit charge
617         storage_ptr->commitCharge(
618             timestep,
619             dt_hrs,
620             storage_ptr->power_kW
621         );
622     }
623
624     return;
625 }   /* __handleStorageCharging() */
```

4.3.3.10 __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< 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.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.

```
663 {
664     double acceptable_kW = 0;
665     double curtailment_kW = 0;
666 
667     Storage* storage_ptr;
668     Combustion* combustion_ptr;
669     Renewable* renewable_ptr;
670 
671     for (size_t j = 0; j < storage_ptr_vec_ptr->size(); j++) {
672         storage_ptr = storage_ptr_vec_ptr->at(j);
673 
674         //  1. attempt to charge from Combustion curtailment first
675         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
676             combustion_ptr = combustion_ptr_vec_ptr->at(i);
677             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
678 
679             if (curtailment_kW <= 0) {
680                 continue;
681             }
682 
683             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
684 
685             if (acceptable_kW > curtailment_kW) {
686                 acceptable_kW = curtailment_kW;
687             }
688 
689             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
690             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
691             storage_ptr->power_kW += acceptable_kW;
692         }
693 
694         //  2. attempt to charge from Renewable curtailment second
695         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
696             renewable_ptr = renewable_ptr_vec_ptr->at(i);
697             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
698 
699             if (curtailment_kW <= 0) {
700                 continue;
701             }
702 
703             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
704 
705             if (acceptable_kW > curtailment_kW) {
706                 acceptable_kW = curtailment_kW;
707             }
708 
709             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
710             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
711             storage_ptr->power_kW += acceptable_kW;
712         }
713 
714         //  3. commit charge
715         storage_ptr->commitCharge(
716             timestep,
717             dt_hrs,
718             storage_ptr->power_kW
719         );
720     }
721 
722     return;
723 }   /* __handleStorageCharging() */
```

4.3.3.11 __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.

```
953 {
954     double discharging_kW = 0;
955 
956     Storage* storage_ptr;
957 
958     std::list<Storage*>::iterator iter;
959     for (
960         iter = storage_ptr_list.begin();
961         iter != storage_ptr_list.end();
962         iter++
963     ){
964         storage_ptr = (*iter);
965 
966         discharging_kW = storage_ptr->getAvailablekW(dt_hrs);
967 
968         if (discharging_kW > net_load_kW) {
969             discharging_kW = net_load_kW;
970         }
971 
972         net_load_kW = storage_ptr->commitDischarge(
973             timestep,
974             dt_hrs,
975             discharging_kW,
976             net_load_kW
977         );
978     }
979 
980     return net_load_kW;
981 }   /* __handleStorageDischarging() */
```

4.3.3.12 applyDispatchControl()

```
void Controller::applyDispatchControl (
            ElectricalLoad * electrical_load_ptr,
            std::vector< Combustion * > * combustion_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.
combustion_ptr_vec_ptr	A pointer to the Combustion 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.

```
1124 {
1125     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
1126         switch (this->control_mode) {
1127             case (ControlMode :: LOAD_FOLLOWING): {
1128                 if (this->net_load_vec_kW[i] <= 0) {
1129                     this->__applyLoadFollowingControl_CHARGING(
1130                         i,
1131                         electrical_load_ptr,
1132                         combustion_ptr_vec_ptr,
1133                         renewable_ptr_vec_ptr,
1134                         storage_ptr_vec_ptr
1135                     );
1136                 }
1137 
1138                 else {
1139                     this->__applyLoadFollowingControl_DISCHARGING(
1140                         i,
1141                         electrical_load_ptr,
1142                         combustion_ptr_vec_ptr,
1143                         renewable_ptr_vec_ptr,
1144                         storage_ptr_vec_ptr
1145                     );
1146                 }
1147 
1148                 break;
1149             }
1150 
1151             case (ControlMode :: CYCLE_CHARGING): {
1152                 if (this->net_load_vec_kW[i] <= 0) {
1153                     this->__applyCycleChargingControl_CHARGING(
1154                         i,
1155                         electrical_load_ptr,
1156                         combustion_ptr_vec_ptr,
1157                         renewable_ptr_vec_ptr,
1158                         storage_ptr_vec_ptr
1159                     );
1160                 }
1161 
1162                 else {
1163                     this->__applyCycleChargingControl_DISCHARGING(
1164                         i,
1165                         electrical_load_ptr,
1166                         combustion_ptr_vec_ptr,
1167                         renewable_ptr_vec_ptr,
1168                         storage_ptr_vec_ptr
1169                     );
1170                 }
1171 
1172                 break;
1173             }
1174 
1175             default: {
1176                 std::string error_str = "ERROR:  Controller :: applyDispatchControl():  ";
1177                 error_str += "control mode ";
1178                 error_str += std::to_string(this->control_mode);
1179                 error_str += " not recognized";
1180 
1181                 #ifdef _WIN32
1182                     std::cout « error_str « std::endl;
1183                 #endif
1184 
1185                 throw std::runtime_error(error_str);
1186 
1187                 break;
1188             }
1189         }
1190     }
1191 
1192     return;
1193 }   /* applyDispatchControl() */
```

4.3.3.13 clear()

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

Method to clear all attributes of the Controller object.

```
1208 {
```

```
1209     this->net_load_vec_kW.clear();
1210     this->missed_load_vec_kW.clear();
1211     this->combustion_map.clear();
1212 
1213     return;
1214 }    /* clear() */
```

4.3.3.14 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.

```
1083 {
1084     //  1. compute net load
1085     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);
1086 
1087     //  2. construct Combustion table
1088     this->__constructCombustionMap(combustion_ptr_vec_ptr);
1089 
1090     return;
1091 }    /* init() */
```

4.3.3.15 setControlMode()

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

Parameters

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

```
1017 {
1018     this->control_mode = control_mode;
1019 
1020     switch(control_mode) {
1021         case (ControlMode :: LOAD_FOLLOWING): {
1022             this->control_string = "LOAD_FOLLOWING";
1023 
1024             break;
1025         }
1026 
1027         case (ControlMode :: CYCLE_CHARGING): {
1028             this->control_string = "CYCLE_CHARGING";
1029 
1030             break;
1031         }
1032 
```

```
1033         default: {
1034             std::string error_str = "ERROR:  Controller :: setControlMode():  ";
1035                 error_str += "control mode ";
1036                 error_str += std::to_string(control_mode);
1037                 error_str += " not recognized";
1038 
1039                 #ifdef _WIN32
1040                     std::cout « error_str « std::endl;
1041                 #endif
1042 
1043                 throw std::runtime_error(error_str);
1044 
1045             break;
1046         }
1047     }
1048 
1049     return;
1050 }   /* 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:

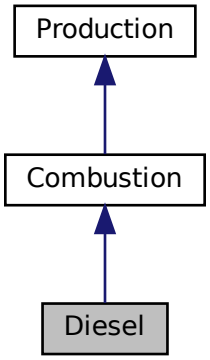
- header/Controller.h
- source/Controller.cpp

4.4 Diesel Class Reference

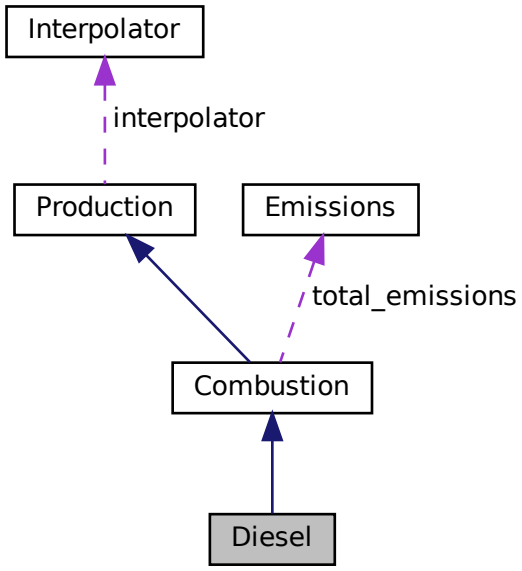
A derived class of the Combustion branch of Production which models production using a diesel generator.

```
#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)

  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.

```
575 {
576     return;
577 }   /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

```
Diesel::Diesel (
            int n_points,
            double n_years,
            DieselInputs diesel_inputs )
```

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.

```
605   :
606 Combustion(
607     n_points,
608     n_years,
609     diesel_inputs.combustion_inputs
610 )
611 {
612     //  1. check inputs
613     this->__checkInputs(diesel_inputs);
614
615     //  2. set attributes
616     this->type = CombustionType :: DIESEL;
617     this->type_str = "DIESEL";
618
619     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
620
621     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
622
623     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
624     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
625     this->time_since_last_start_hrs = 0;
626
627     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
628     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
629     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
630     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
631     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
632     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
633
634     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
635         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
636     }
637
638     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
639         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
640     }
641
642     if (diesel_inputs.capital_cost < 0) {
643         this->capital_cost = this->__getGenericCapitalCost();
644     }
645
646     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
647         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
648     }
649
650     if (not this->is_sunk) {
651         this->capital_cost_vec[0] = this->capital_cost;
652     }
653
654     //  3. construction print
655     if (this->print_flag) {
656         std::cout « "Diesel object constructed at " « this « std::endl;
657     }
658
659     return;
660 }   /* Diesel() */
```

4.4.2.3 ~Diesel()

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

Destructor for the Diesel class.

```
815 {
816     //  1. destruction print
817     if (this->print_flag) {
818         std::cout « "Diesel object at " « this « " destroyed" « std::endl;
819     }
820
821     return;
822 }   /* ~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 « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
365     ofs « "Capital Cost: " « this->capital_cost « "  \n";
366     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
367         « " per kWh produced  \n";
368     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
369         « "  \n";
370     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
371         « "  \n";
372     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
373     ofs « "\n";
374 
375     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
376     ofs « "\n--------\n\n";
377 
378     //  2.2. Combustion attributes
379     ofs « "## Combustion Attributes\n";
380     ofs « "\n";
381 
382     ofs « "Fuel Cost: " « this->fuel_cost_L « " per L  \n";
383     ofs « "Nominal Fuel Escalation Rate (annual): "
384         « this->nominal_fuel_escalation_annual « "  \n";
385     ofs « "Real Fuel Escalation Rate (annual): "
386         « this->real_fuel_escalation_annual « "  \n";
387     ofs « "\n";
388 
389     ofs « "Linear Fuel Slope: " « this->linear_fuel_slope_LkWh « " L/kWh  \n";
390     ofs « "Linear Fuel Intercept Coefficient: " « this->linear_fuel_intercept_LkWh
391         « " L/kWh  \n";
392     ofs « "\n";
393 
394     ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
395         « this->CO2_emissions_intensity_kgL « " kg/L  \n";
396 
397     ofs « "Carbon Monoxide (CO) Emissions Intensity: "
398         « this->CO_emissions_intensity_kgL « " kg/L  \n";
399 
400     ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
```

```
401         « this->NOx_emissions_intensity_kgL « " kg/L  \n";
402
403     ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
404         « this->SOx_emissions_intensity_kgL « " kg/L  \n";
405
406     ofs « "Methane (CH4) Emissions Intensity: "
407         « this->CH4_emissions_intensity_kgL « " kg/L  \n";
408
409     ofs « "Particulate Matter (PM) Emissions Intensity: "
410         « this->PM_emissions_intensity_kgL « " kg/L  \n";
411
412     ofs « "\n--------\n\n";
413
414     //  2.3. Diesel attributes
415     ofs « "## Diesel Attributes\n";
416     ofs « "\n";
417
418     ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « "  \n";
419     ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs  \n";
420
421     ofs « "\n--------\n\n";
422
423     //  2.4. Diesel Results
424     ofs « "## Results\n";
425     ofs « "\n";
426
427     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
428     ofs « "\n";
429
430     ofs « "Total Dispatch: " « this->total_dispatch_kWh
431         « " kWh  \n";
432
433     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
434         « " per kWh dispatched  \n";
435     ofs « "\n";
436
437     ofs « "Running Hours: " « this->running_hours « "  \n";
438     ofs « "Starts: " « this->n_starts « "  \n";
439     ofs « "Replacements: " « this->n_replacements « "  \n";
440
441     ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
442         « "(Annual Average: " « this->total_fuel_consumed_L / this->n_years
443         « " L/yr)  \n";
444     ofs « "\n";
445
446     ofs « "Total Carbon Dioxide (CO2) Emissions: " «
447         this->total_emissions.CO2_kg « " kg "
448         « "(Annual Average: " «  this->total_emissions.CO2_kg / this->n_years
449         « " kg/yr)  \n";
450
451     ofs « "Total Carbon Monoxide (CO) Emissions: " «
452         this->total_emissions.CO_kg « " kg "
453         « "(Annual Average: " «  this->total_emissions.CO_kg / this->n_years
454         « " kg/yr)  \n";
455
456     ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
457         this->total_emissions.NOx_kg « " kg "
458         « "(Annual Average: " «  this->total_emissions.NOx_kg / this->n_years
459         « " kg/yr)  \n";
460
461     ofs « "Total Sulfur Oxides (SOx) Emissions: " «
462         this->total_emissions.SOx_kg « " kg "
463         « "(Annual Average: " «  this->total_emissions.SOx_kg / this->n_years
464         « " kg/yr)  \n";
465
466     ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
467         « "(Annual Average: " «  this->total_emissions.CH4_kg / this->n_years
468         « " kg/yr)  \n";
469
470     ofs « "Total Particulate Matter (PM) Emissions: " «
471         this->total_emissions.PM_kg « " kg "
472         « "(Annual Average: " «  this->total_emissions.PM_kg / this->n_years
473         « " kg/yr)  \n";
474
475     ofs « "\n--------\n\n";
476
477     ofs.close();
478     return;
479 }   /* __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.

```
509 {
510     //  1. create filestream
511     write_path += "time_series_results.csv";
512     std::ofstream ofs;
513     ofs.open(write_path, std::ofstream::out);
514 
515     //  2. write time series results (comma separated value)
516     ofs « "Time (since start of data) [hrs],";
517     ofs « "Production [kW],";
518     ofs « "Dispatch [kW],";
519     ofs « "Storage [kW],";
520     ofs « "Curtailment [kW],";
521     ofs « "Is Running (N = 0 / Y = 1),";
522     ofs « "Fuel Consumption [L],";
523     ofs « "Fuel Cost (actual),";
524     ofs « "Carbon Dioxide (CO2) Emissions [kg],";
525     ofs « "Carbon Monoxide (CO) Emissions [kg],";
526     ofs « "Nitrogen Oxides (NOx) Emissions [kg],";
527     ofs « "Sulfur Oxides (SOx) Emissions [kg],";
528     ofs « "Methane (CH4) Emissions [kg],";
529     ofs « "Particulate Matter (PM) Emissions [kg],";
530     ofs « "Capital Cost (actual),";
531     ofs « "Operation and Maintenance Cost (actual),";
532     ofs « "\n";
533 
534     for (int i = 0; i < max_lines; i++) {
535         ofs « time_vec_hrs_ptr->at(i) « ",";
536         ofs « this->production_vec_kW[i] « ",";
537         ofs « this->dispatch_vec_kW[i] « ",";
538         ofs « this->storage_vec_kW[i] « ",";
539         ofs « this->curtailment_vec_kW[i] « ",";
540         ofs « this->is_running_vec[i] « ",";
541         ofs « this->fuel_consumption_vec_L[i] « ",";
542         ofs « this->fuel_cost_vec[i] « ",";
543         ofs « this->CO2_emissions_vec_kg[i] « ",";
544         ofs « this->CO_emissions_vec_kg[i] « ",";
545         ofs « this->NOx_emissions_vec_kg[i] « ",";
546         ofs « this->SOx_emissions_vec_kg[i] « ",";
547         ofs « this->CH4_emissions_vec_kg[i] « ",";
548         ofs « this->PM_emissions_vec_kg[i] « ",";
549         ofs « this->capital_cost_vec[i] « ",";
550         ofs « this->operation_maintenance_cost_vec[i] « ",";
551         ofs « "\n";
552     }
553 
554     ofs.close();
555     return;
556 }   /* __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.

```
773 {
774     //  1. handle start/stop, enforce minimum runtime constraint
775     this->__handleStartStop(timestep, dt_hrs, production_kW);
776 
777     //  2. invoke base class method
778     load_kW = Combustion :: commit(
779         timestep,
780         dt_hrs,
781         production_kW,
782         load_kW
783     );
784 
785     if (this->is_running) {
786         //  3. log time since last start
787         this->time_since_last_start_hrs += dt_hrs;
788 
789         //  4. correct operation and maintenance costs (should be non-zero if idling)
790         if (production_kW <= 0) {
791             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
792 
793             double operation_maintenance_cost =
794                 this->operation_maintenance_cost_kWh * produced_kWh;
795             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
796         }
797     }
798 
799     return load_kW;
800 }   /* 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.

```
678 {
679     //  1. reset attributes
680     this->time_since_last_start_hrs = 0;
681 
682     //  2. invoke base class method
683     Combustion :: handleReplacement(timestep);
684 
685     return;
686 }   /* __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.

```
718 {
719     //  1. return on request of zero
720     if (request_kW <= 0) {
721         return 0;
722     }
723 
724     double deliver_kW = request_kW;
725 
726     //  2. enforce capacity constraint
727     if (deliver_kW > this->capacity_kW) {
728         deliver_kW = this->capacity_kW;
729     }
730 
731     //  3. enforce minimum load ratio
732     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
733         deliver_kW = this->minimum_load_ratio * this->capacity_kW;
734     }
735 
736     return deliver_kW;
737 }   /* 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:

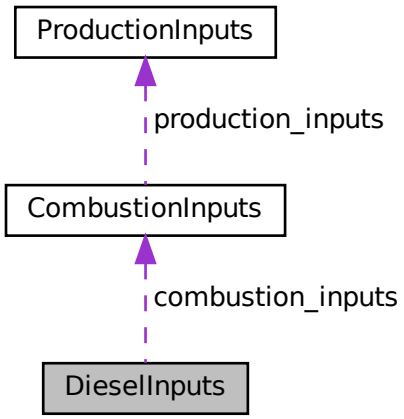
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

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 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.

4.8.1 Detailed Description

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

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Interpolator()

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

Constructor for the Interpolator class.

```
670 {
671     //...
672
673     return;
674 }   /* Interpolator() */
```

4.8.2.2 ~Interpolator()

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

Destructor for the Interpolator class.

```
856 {
857     //...
858
859     return;
860 }   /* ~Interpolator() */
```

4.8.3 Member Function Documentation

4.8.3.1 __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_x	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
133         #ifdef _WIN32
134             std::cout « error_str « std::endl;
135         #endif
136
137         throw std::invalid_argument(error_str);
138     }
139
140     return;
141 }   /* __checkBounds1D() */
```

4.8.3.2 __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.

```
164 {
165     //  1. key error
166     if (this->interp_map_2D.count(data_key) == 0) {
167         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
168         error_str += "data key ";
169         error_str += std::to_string(data_key);
170         error_str += " has not been registered";
171
172         #ifdef _WIN32
173             std::cout « error_str « std::endl;
174         #endif
175
176         throw std::invalid_argument(error_str);
177     }
178
179     //  2. bounds error (x_interp)
180     if (
181         interp_x < this->interp_map_2D[data_key].min_x or
182         interp_x > this->interp_map_2D[data_key].max_x
183     ) {
184         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
185         error_str += "interpolation value interp_x = ";
186         error_str += std::to_string(interp_x);
187         error_str += " is outside of the given interpolation data domain";
188
189         #ifdef _WIN32
190             std::cout « error_str « std::endl;
191         #endif
192
193         throw std::invalid_argument(error_str);
194     }
195
```

```
196     //  2. bounds error (y_interp)
197     if (
198         interp_y < this->interp_map_2D[data_key].min_y or
199         interp_y > this->interp_map_2D[data_key].max_y
200     ) {
201         std::string error_str = "ERROR:  Interpolator::interp2D()  ";
202         error_str += "interpolation value interp_y = ";
203         error_str += std::to_string(interp_y);
204         error_str += " is outside of the given interpolation data domain";
205 
206         #ifdef _WIN32
207             std::cout « error_str « std::endl;
208         #endif
209 
210         throw std::invalid_argument(error_str);
211     }
212 
213     return;
214 }   /* __checkBounds2D() */
```

4.8.3.3 __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() */
```

4.8.3.4 __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() */
```

4.8.3.5 __getDataStringMatrix()

```
std::vector< std::vector< std::string > > Interpolator::__getDataStringMatrix (
              std::string path_2_data )  [private]
389 {
390     //  1. create input file stream
391     std::ifstream ifs;
392     ifs.open(path_2_data);
393
394     //  2. check that open() worked
395     if (not ifs.is_open()) {
396         std::string error_str = "ERROR:  Interpolator::__getDataStringMatrix()  ";
397         error_str += " failed to open ";
398         error_str += path_2_data;
399
400         #ifdef _WIN32
401             std::cout « error_str « std::endl;
402         #endif
403
404         throw std::invalid_argument(error_str);
405     }
406
407     //  3. read file line by line
408     bool is_header = true;
409     std::string line;
410     std::vector<std::string> line_split_vec;
411     std::vector<std::vector<std::string» string_matrix;
412
413     while (not ifs.eof()) {
414         std::getline(ifs, line);
415
416         if (is_header) {
417             is_header = false;
418             continue;
419         }
420
421         line_split_vec = this->__splitCommaSeparatedString(line);
422
423         if (not line_split_vec.empty()) {
424             string_matrix.push_back(line_split_vec);
425         }
426     }
427
428     ifs.close();
429     return string_matrix;
430 }   /* __getDataStringMatrix() */
```

4.8.3.6 __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.

```
306 {
307     int idx = 0;
308     while (
309         not (interp_x >= x_vec_ptr->at(idx) and interp_x <= x_vec_ptr->at(idx + 1))
310     ) {
311         idx++;
312     }
313 
314     return idx;
315 }   /* __getInterpolationIndex() */
```

4.8.3.7 __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.

```
271 {
272     for (size_t i = 0; i < str.size(); i++) {;
273         if (isalpha(str[i])) {
274             return true;
275         }
276     }
277 
278     return false;
279 }   /* __isAlpha() */
```

4.8.3.8 __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.

```
450 {
451     //  1. get string matrix
452     std::vector<std::vector<std::string» string_matrix =
453         this->__getDataStringMatrix(path_2_data);
454 
455     //  2. read string matrix contents into 1D interpolation struct
456     InterpolatorStruct1D interp_struct_1D;
457 
458     interp_struct_1D.n_points = string_matrix.size();
459     interp_struct_1D.x_vec.resize(interp_struct_1D.n_points, 0);
460     interp_struct_1D.y_vec.resize(interp_struct_1D.n_points, 0);
461 
462     for (int i = 0; i < interp_struct_1D.n_points; i++) {
463         try {
464             interp_struct_1D.x_vec[i] = std::stod(string_matrix[i][0]);
465             interp_struct_1D.y_vec[i] = std::stod(string_matrix[i][1]);
466         }
467 
468         catch (...) {
469             this->__throwReadError(path_2_data, 1);
470         }
471     }
472 
473     interp_struct_1D.min_x = interp_struct_1D.x_vec[0];
474     interp_struct_1D.max_x = interp_struct_1D.x_vec[interp_struct_1D.n_points - 1];
475 
476     //  3. write struct to map
477     this->interp_map_1D.insert(
478         std::pair<int, InterpolatorStruct1D>(data_key, interp_struct_1D)
479     );
480 
481     /*
482     // ==== TEST PRINT ==== //
483     std::cout « std::endl;
484     std::cout « path_2_data « std::endl;
485     std::cout « "--------" « std::endl;
486 
487     std::cout « "n_points: " « this->interp_map_1D[data_key].n_points « std::endl;
488 
489     std::cout « "x_vec: [";
490     for (
491         int i = 0;
492         i < this->interp_map_1D[data_key].n_points;
493         i++
494     ) {
495         std::cout « this->interp_map_1D[data_key].x_vec[i] « ", ";
496     }
497     std::cout « "]" « std::endl;
498 
499     std::cout « "y_vec: [";
500     for (
501         int i = 0;
502         i < this->interp_map_1D[data_key].n_points;
503         i++
504     ) {
505         std::cout « this->interp_map_1D[data_key].y_vec[i] « ", ";
506     }
507     std::cout « "]" « std::endl;
508 
509     std::cout « std::endl;
510     // ==== END TEST PRINT ==== //
511     //*/
512 
513     return;
514 }   /* __readData1D() */
```

4.8.3.9 __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.

```
534 {
535     //  1. get string matrix
536     std::vector<std::vector<std::string» string_matrix =
537         this->__getDataStringMatrix(path_2_data);
538 
539     //  2. read string matrix contents into 2D interpolation map
540     InterpolatorStruct2D interp_struct_2D;
541 
542     interp_struct_2D.n_rows = string_matrix.size() - 1;
543     interp_struct_2D.n_cols = string_matrix[0].size() - 1;
544 
545     interp_struct_2D.x_vec.resize(interp_struct_2D.n_cols, 0);
546     interp_struct_2D.y_vec.resize(interp_struct_2D.n_rows, 0);
547 
548     interp_struct_2D.z_matrix.resize(interp_struct_2D.n_rows, {});
549 
550     for (int i = 0; i < interp_struct_2D.n_rows; i++) {
551         interp_struct_2D.z_matrix[i].resize(interp_struct_2D.n_cols, 0);
552     }
553 
554     for (size_t i = 1; i < string_matrix[0].size(); i++) {
555         try {
556             interp_struct_2D.x_vec[i - 1] = std::stod(string_matrix[0][i]);
557         }
558 
559         catch (...) {
560             this->__throwReadError(path_2_data, 2);
561         }
562     }
563 
564     interp_struct_2D.min_x = interp_struct_2D.x_vec[0];
565     interp_struct_2D.max_x = interp_struct_2D.x_vec[interp_struct_2D.n_cols - 1];
566 
567     for (size_t i = 1; i < string_matrix.size(); i++) {
568         try {
569             interp_struct_2D.y_vec[i - 1] = std::stod(string_matrix[i][0]);
570         }
571 
572         catch (...) {
573             this->__throwReadError(path_2_data, 2);
574         }
575     }
576 
577     interp_struct_2D.min_y = interp_struct_2D.y_vec[0];
578     interp_struct_2D.max_y = interp_struct_2D.y_vec[interp_struct_2D.n_rows - 1];
579 
580     for (size_t i = 1; i < string_matrix.size(); i++) {
581         for (size_t j = 1; j < string_matrix[0].size(); j++) {
582             try {
583                 interp_struct_2D.z_matrix[i - 1][j - 1] = std::stod(string_matrix[i][j]);
584             }
585 
586             catch (...) {
587                 this->__throwReadError(path_2_data, 2);
588             }
589         }
590     }
591 
592     //  3. write struct to map
593     this->interp_map_2D.insert(
594         std::pair<int, InterpolatorStruct2D>(data_key, interp_struct_2D)
595     );
596 
597     /*
598     // ==== TEST PRINT ==== //
599     std::cout « std::endl;
600     std::cout « path_2_data « std::endl;
601     std::cout « "--------" « std::endl;
602 
603     std::cout « "n_rows: " « this->interp_map_2D[data_key].n_rows « std::endl;
604     std::cout « "n_cols: " « this->interp_map_2D[data_key].n_cols « std::endl;
605 
606     std::cout « "x_vec: [";
607     for (
608         int i = 0;
609         i < this->interp_map_2D[data_key].n_cols;
610         i++
611     ) {
612         std::cout « this->interp_map_2D[data_key].x_vec[i] « ", ";
613     }
```

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

4.8.3.10 __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.

```
344 {
345     std::vector<std::string> str_split_vec;
346
347     size_t idx = 0;
348     std::string substr;
349
350     while ((idx = str.find(',')) != std::string::npos) {
351         substr = str.substr(0, idx);
352
353         if (substr == break_str) {
354             break;
355         }
356
```

```
357         str_split_vec.push_back(substr);
358
359         str.erase(0, idx + 1);
360     }
361
362     return str_split_vec;
363 }   /* __splitCommaSeparatedString() */
```

4.8.3.11 __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.

```
235 {
236     std::string error_str = "ERROR:  Interpolator::addData";
237     error_str += std::to_string(dimensions);
238     error_str += "D()  ";
239     error_str += " failed to read ";
240     error_str += path_2_data;
241     error_str += " (this is probably a std::stod() error; is there non-numeric ";
242     error_str += "data where only numeric data should be?)";
243
244     #ifdef _WIN32
245         std::cout « error_str « std::endl;
246     #endif
247
248     throw std::runtime_error(error_str);
249
250     return;
251 }   /* __throwReadError() */
```

4.8.3.12 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.

```
694 {
695     //  1. check key
696     this->__checkDataKey1D(data_key);
697
698     //  2. read data into map
699     this->__readData1D(data_key, path_2_data);
```

```
700
701     //  3. record path
702     this->path_map_1D.insert(std::pair<int, std::string>(data_key, path_2_data));
703
704     return;
705 }   /* addData1D() */
```

4.8.3.13 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.

```
725 {
726     //  1. check key
727     this->__checkDataKey2D(data_key);
728
729     //  2. read data into map
730     this->__readData2D(data_key, path_2_data);
731
732     //  3. record path
733     this->path_map_2D.insert(std::pair<int, std::string>(data_key, path_2_data));
734
735     return;
736 }   /* addData2D() */
```

4.8.3.14 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.

```
758 {
759     //  1. check bounds
760     this->__checkBounds1D(data_key, interp_x);
761
762     //  2. get interpolation index
```

```
763     int idx = this->__getInterpolationIndex(
764         interp_x,
765         &(this->interp_map_1D[data_key].x_vec)
766     );
767 
768     //  3. perform interpolation
769     double x_0 = this->interp_map_1D[data_key].x_vec[idx];
770     double x_1 = this->interp_map_1D[data_key].x_vec[idx + 1];
771 
772     double y_0 = this->interp_map_1D[data_key].y_vec[idx];
773     double y_1 = this->interp_map_1D[data_key].y_vec[idx + 1];
774 
775     double interp_y = ((y_1 - y_0) / (x_1 - x_0)) * (interp_x - x_0) + y_0;
776 
777     return interp_y;
778 }   /* interp1D() */
```

4.8.3.15 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.

```
803 {
804     //  1. check bounds
805     this->__checkBounds2D(data_key, interp_x, interp_y);
806 
807     //  2. get interpolation indices
808     int idx_x = this->__getInterpolationIndex(
809         interp_x,
810         &(this->interp_map_2D[data_key].x_vec)
811     );
812 
813     int idx_y = this->__getInterpolationIndex(
814         interp_y,
815         &(this->interp_map_2D[data_key].y_vec)
816     );
817 
818     //  3. perform first horizontal interpolation
819     double x_0 = this->interp_map_2D[data_key].x_vec[idx_x];
820     double x_1 = this->interp_map_2D[data_key].x_vec[idx_x + 1];
821 
822     double z_0 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x];
823     double z_1 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x + 1];
824 
825     double interp_z_0 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
826 
827     //  4. perform second horizontal interpolation
828     z_0 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x];
829     z_1 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x + 1];
830 
831     double interp_z_1 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
832 
```

```
833     //  5. perform vertical interpolation
834     double y_0 = this->interp_map_2D[data_key].y_vec[idx_y];
835     double y_1 = this->interp_map_2D[data_key].y_vec[idx_y + 1];
836
837     double interp_z =
838         ((interp_z_1 - interp_z_0) / (y_1 - y_0)) * (interp_y - y_0) + interp_z_0;
839
840     return interp_z;
841 }   /* interp2D() */
```

4.8.4 Member Data Documentation

4.8.4.1 interp_map_1D

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

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

4.8.4.2 interp_map_2D

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

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

4.8.4.3 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.

4.8.4.4 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

4.9 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.

4.9.1 Detailed Description

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

4.9.2 Member Data Documentation

4.9.2.1 max_x

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

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

4.9.2.2 min_x

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

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

4.9.2.3 n_points

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

The number of data points in each parallel vector.

4.9.2.4 x_vec

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

A vector of independent data.

4.9.2.5 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

4.10 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.

4.10.1 Detailed Description

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

4.10.2 Member Data Documentation

4.10.2.1 max_x

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

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

4.10.2.2 max_y

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

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

4.10.2.3 min_x

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

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

4.10.2.4 min_y

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

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

4.10.2.5 n_cols

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

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

4.10.2.6 n_rows

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

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

4.10.2.7 x_vec

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

A vector of independent data (columns).

4.10.2.8 y_vec

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

A vector of independent data (rows).

4.10.2.9 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:

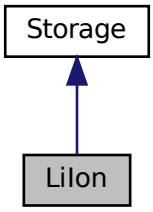
- header/Interpolator.h

4.11 LiIon Class Reference

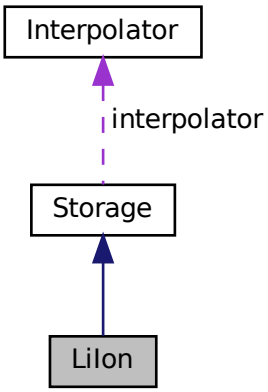
A derived class of Storage which models energy storage by way of lithium-ion batteries.

`#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

- double dynamic_energy_capacity_kWh

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

4.11.1 Detailed Description

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

4.11.2 Constructor & Destructor Documentation

4.11.2.1 LiIon() [1/2]

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

Constructor (dummy) for the LiIon class.

```
646 {
647     return;
648 }   /* LiIon() */
```

4.11.2.2 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.

```
676   :
677 Storage(
678     n_points,
679     n_years,
680     liion_inputs.storage_inputs
681 )
682 {
683     //  1. check inputs
684     this->__checkInputs(liion_inputs);
685 
686     //  2. set attributes
687     this->type = StorageType :: LIION;
688     this->type_str = "LIION";
689 
690     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
691     this->SOH = 1;
692     this->replace_SOH = liion_inputs.replace_SOH;
693 
694     this->degradation_alpha = liion_inputs.degradation_alpha;
695     this->degradation_beta = liion_inputs.degradation_beta;
696     this->degradation_B_hat_cal_0 = liion_inputs.degradation_B_hat_cal_0;
697     this->degradation_r_cal = liion_inputs.degradation_r_cal;
698     this->degradation_Ea_cal_0 = liion_inputs.degradation_Ea_cal_0;
699     this->degradation_a_cal = liion_inputs.degradation_a_cal;
700     this->degradation_s_cal = liion_inputs.degradation_s_cal;
701     this->gas_constant_JmolK = liion_inputs.gas_constant_JmolK;
702     this->temperature_K = liion_inputs.temperature_K;
703 
704     this->init_SOC = liion_inputs.init_SOC;
705     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
706 
707     this->min_SOC = liion_inputs.min_SOC;
708     this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
709     this->max_SOC = liion_inputs.max_SOC;
710 
711     this->charging_efficiency = liion_inputs.charging_efficiency;
712     this->discharging_efficiency = liion_inputs.discharging_efficiency;
713 
714     if (liion_inputs.capital_cost < 0) {
715         this->capital_cost = this->__getGenericCapitalCost();
716     }
717 
718     if (liion_inputs.operation_maintenance_cost_kWh < 0) {
719         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
720     }
721 
722     if (not this->is_sunk) {
723         this->capital_cost_vec[0] = this->capital_cost;
724     }
725 
726     this->SOH_vec.resize(this->n_points, 0);
727 
728     //  3. construction print
729     if (this->print_flag) {
730         std::cout « "LiIon object constructed at " « this « std::endl;
731     }
732 
733     return;
734 }   /* LiIon() */
```

4.11.2.3 ~LiIon()

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

Destructor for the LiIon class.

```
990 {
991     //  1. destruction print
```

```
992     if (this->print_flag) {
993         std::cout « "LiIon object at " « this « " destroyed" « std::endl;
994     }
995
996     return;
997 }   /* ~LiIon() */
```

4.11.3 Member Function Documentation

4.11.3.1 __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() */
```

4.11.3.2 __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 [2023]

Parameters

SOC	The current state of charge of the asset.

Returns

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

```
427 {
428     double B_cal = this->degradation_B_hat_cal_0 *
429         exp(this->degradation_r_cal * SOC);
430 
431     return B_cal;
432 }   /* __getBcal() */
```

4.11.3.3 __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 [2023]

Parameters

SOC	The current state of charge of the asset.

Returns

The activation energy value for the given state of charge.

```
454 {
455     double Ea_cal = this->degradation_Ea_cal_0;
456 
457     Ea_cal -= this->degradation_a_cal *
458         (exp(this->degradation_s_cal * SOC) - 1);
459 
460     return Ea_cal;
461 }   /* __getEacal( */
```

4.11.3.4 __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() */
```

4.11.3.5 __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() */
```

4.11.3.6 __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     return;
357 }   /* __handleDegradation() */
```

4.11.3.7 __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 [2023]

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.

```
380 {
381     //  1. compute SOC
382     double SOC = this->charge_kWh / this->energy_capacity_kWh;
383 
384     //  2. compute C-rate and corresponding acceleration factor
385     double C_rate = charging_discharging_kW / this->power_capacity_kW;
386 
387     double C_acceleration_factor =
388         1 + this->degradation_alpha * pow(C_rate, this->degradation_beta);
389 
390     //  3. compute dSOH / dt
391     double B_cal = __getBcal(SOC);
392     double Ea_cal = __getEacal(SOC);
393 
394     double dSOH_dt = B_cal *
395         exp((-1 * Ea_cal) / (this->gas_constant_JmolK * this->temperature_K));
396 
397     dSOH_dt *= dSOH_dt;
398     dSOH_dt *= 1 / (2 * this->SOH);
399     dSOH_dt *= C_acceleration_factor;
400 
401     //  4. update state of health
402     this->SOH -= dSOH_dt * dt_hrs;
403 
404     return;
405 }   /* __modelDegradation() */
```

4.11.3.8 __toggleDepleted()

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

Helper method to toggle the is_depleted attribute of Lilon.

```
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() */
```

4.11.3.9 __writeSummary()

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

Helper method to write summary results for Lilon.

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.

```
479 {
480     //  1. create filestream
481     write_path += "summary_results.md";
482     std::ofstream ofs;
483     ofs.open(write_path, std::ofstream::out);
484 
485     //  2. write summary results (markdown)
486     ofs « "# ";
487     ofs « std::to_string(int(ceil(this->power_capacity_kW)));
488     ofs « " kW ";
489     ofs « std::to_string(int(ceil(this->energy_capacity_kWh)));
490     ofs « " kWh LIION Summary Results\n";
491     ofs « "\n--------\n\n";
492 
493     //  2.1. Storage attributes
494     ofs « "## Storage Attributes\n";
495     ofs « "\n";
496     ofs « "Power Capacity: " « this->power_capacity_kW « "kW  \n";
497     ofs « "Energy Capacity: " « this->energy_capacity_kWh « "kWh  \n";
498     ofs « "\n";
499 
500     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
501     ofs « "Capital Cost: " « this->capital_cost « "  \n";
502     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
503         « " per kWh charged/discharged  \n";
504     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
505         « "  \n";
506     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
507         « "  \n";
508     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
509 
510     ofs « "\n--------\n\n";
511 
512     //  2.2. LiIon attributes
513     ofs « "## LiIon Attributes\n";
514     ofs « "\n";
515 
516     ofs « "Charging Efficiency: " « this->charging_efficiency « "  \n";
517     ofs « "Discharging Efficiency: " « this->discharging_efficiency « "  \n";
518     ofs « "\n";
519 
520     ofs « "Initial State of Charge: " « this->init_SOC « "  \n";
521     ofs « "Minimum State of Charge: " « this->min_SOC « "  \n";
522     ofs « "Hyteresis State of Charge: " « this->hysteresis_SOC « "  \n";
523     ofs « "Maximum State of Charge: " « this->max_SOC « "  \n";
524     ofs « "\n";
525 
526     ofs « "Replacement State of Health: " « this->replace_SOH « "  \n";
527     ofs « "\n";
528 
529     ofs « "Degradation Acceleration Coeff.: " « this->degradation_alpha « "  \n";
530     ofs « "Degradation Acceleration Exp.: " « this->degradation_beta « "  \n";
531     ofs « "Degradation Base Pre-Exponential Factor: "
532         « this->degradation_B_hat_cal_0 « " 1/sqrt(hrs)  \n";
533     ofs « "Degradation Dimensionless Constant (r_cal): "
534         « this->degradation_r_cal « "  \n";
535     ofs « "Degradation Base Activation Energy: "
536         « this->degradation_Ea_cal_0 « " J/mol  \n";
537     ofs « "Degradation Pre-Exponential Factor: "
538         « this->degradation_a_cal « " J/mol  \n";
539     ofs « "Degradation Dimensionless Constant (s_cal): "
540         « this->degradation_s_cal « "  \n";
541     ofs « "Universal Gas Constant: " « this->gas_constant_JmolK
542         « " J/mol.K  \n";
543     ofs « "Absolute Environmental Temperature: " « this->temperature_K « " K  \n";
544     ofs « "\n";
545 
546     ofs « "\n--------\n\n";
547 
548     //  2.3. LiIon Results
549     ofs « "## Results\n";
550     ofs « "\n";
551 
552     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
553     ofs « "\n";
554 
555     ofs « "Total Discharge: " « this->total_discharge_kWh
```

```
556         « " kWh  \n";
557
558     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
559         « " per kWh dispatched  \n";
560     ofs « "\n";
561
562     ofs « "Replacements: " « this->n_replacements « "  \n";
563
564     ofs « "\n--------\n\n";
565     ofs.close();
566     return;
567 }   /* __writeSummary() */
```

4.11.3.10 __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.

```
598 {
599     //  1. create filestream
600     write_path += "time_series_results.csv";
601     std::ofstream ofs;
602     ofs.open(write_path, std::ofstream::out);
603
604     //  2. write time series results (comma separated value)
605     ofs « "Time (since start of data) [hrs],";
606     ofs « "Charging Power [kW],";
607     ofs « "Discharging Power [kW],";
608     ofs « "Charge (at end of timestep) [kWh],";
609     ofs « "State of Health (at end of timestep) [ ],";
610     ofs « "Capital Cost (actual),";
611     ofs « "Operation and Maintenance Cost (actual),";
612     ofs « "\n";
613
614     for (int i = 0; i < max_lines; i++) {
615         ofs « time_vec_hrs_ptr->at(i) « ",";
616         ofs « this->charging_power_vec_kW[i] « ",";
617         ofs « this->discharging_power_vec_kW[i] « ",";
618         ofs « this->charge_vec_kWh[i] « ",";
619         ofs « this->SOH_vec[i] « ",";
620         ofs « this->capital_cost_vec[i] « ",";
621         ofs « this->operation_maintenance_cost_vec[i] « ",";
622         ofs « "\n";
623     }
624
625     ofs.close();
626     return;
627 }   /* __writeTimeSeries() */
```

4.11.3.11 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.

```
881 {
882     //  1. record charging power
883     this->charging_power_vec_kW[timestep] = charging_kW;
884 
885     //  2. update charge and record
886     this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
887     this->charge_vec_kWh[timestep] = this->charge_kWh;
888 
889     //  3. toggle depleted flag (if applicable)
890     this->__toggleDepleted();
891 
892     //  4. model degradation
893     this->__handleDegradation(timestep, dt_hrs, charging_kW);
894 
895     //  5. trigger replacement (if applicable)
896     if (this->SOH <= this->replace_SOH) {
897         this->handleReplacement(timestep);
898     }
899 
900     //  6. capture operation and maintenance costs (if applicable)
901     if (charging_kW > 0) {
902         this->operation_maintenance_cost_vec[timestep] = charging_kW * dt_hrs *
903             this->operation_maintenance_cost_kWh;
904     }
905 
906     this->power_kW= 0;
907     return;
908 }   /* commitCharge() */
```

4.11.3.12 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.

```
944 {
945     //  1. record discharging power, update total
946     this->discharging_power_vec_kW[timestep] = discharging_kW;
947     this->total_discharge_kWh += discharging_kW * dt_hrs;
948 
949     //  2. update charge and record
950     this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
951     this->charge_vec_kWh[timestep] = this->charge_kWh;
952 
953     //  3. update load
954     load_kW -= discharging_kW;
955 
956     //  4. toggle depleted flag (if applicable)
957     this->__toggleDepleted();
958 
959     //  5. model degradation
960     this->__handleDegradation(timestep, dt_hrs, discharging_kW);
961 
962     //  6. trigger replacement (if applicable)
963     if (this->SOH <= this->replace_SOH) {
964         this->handleReplacement(timestep);
965     }
966 
967     //  7. capture operation and maintenance costs (if applicable)
968     if (discharging_kW > 0) {
969         this->operation_maintenance_cost_vec[timestep] = discharging_kW * dt_hrs *
970             this->operation_maintenance_cost_kWh;
971     }
972 
973     this->power_kW = 0;
974     return load_kW;
975 }   /* commitDischarge() */
```

4.11.3.13 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.

```
825 {
826     //  1. get max charge
827     double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;
828 
829     if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
830         max_charge_kWh = this->dynamic_energy_capacity_kWh;
831     }
832 
833     //  2. compute acceptable power
834     //     (accounting for the power currently being charged/discharged by the asset)
835     double acceptable_kW =
836         (max_charge_kWh - this->charge_kWh) /
837         (this->charging_efficiency * dt_hrs);
```

```
838
839     acceptable_kW -= this->power_kW;
840
841     if (acceptable_kW <= 0) {
842         return 0;
843     }
844
845     //  3. apply power constraint
846     if (acceptable_kW > this->power_capacity_kW) {
847         acceptable_kW = this->power_capacity_kW;
848     }
849
850     return acceptable_kW;
851 }   /* getAcceptablekW( */
```

4.11.3.14 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.

```
784 {
785     //  1. get min charge
786     double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
787
788     //  2. compute available power
789     //     (accounting for the power currently being charged/discharged by the asset)
790     double available_kW =
791         ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
792         dt_hrs;
793
794     available_kW -= this->power_kW;
795
796     if (available_kW <= 0) {
797         return 0;
798     }
799
800     //  3. apply power constraint
801     if (available_kW > this->power_capacity_kW) {
802         available_kW = this->power_capacity_kW;
803     }
804
805     return available_kW;
806 }   /* getAvailablekW() */
```

4.11.3.15 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.

```
752 {
753     //  1. reset attributes
754     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
755     this->SOH = 1;
756 
757     // 2. invoke base class method
758     Storage::handleReplacement(timestep);
759 
760     //  3. correct attributes
761     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
762     this->is_depleted = false;
763 
764     return;
765 }   /* __handleReplacement() */
```

4.11.4 Member Data Documentation

4.11.4.1 charging_efficiency

```
double LiIon::charging_efficiency
```

The charging efficiency of the asset.

4.11.4.2 degradation_a_cal

```
double LiIon::degradation_a_cal
```

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

4.11.4.3 degradation_alpha

```
double LiIon::degradation_alpha
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.11.4.4 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.

4.11.4.5 degradation_beta

```
double LiIon::degradation_beta
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.11.4.6 degradation_Ea_cal_0

```
double LiIon::degradation_Ea_cal_0
```

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

4.11.4.7 degradation_r_cal

```
double LiIon::degradation_r_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.11.4.8 degradation_s_cal

```
double LiIon::degradation_s_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.11.4.9 discharging_efficiency

```
double LiIon::discharging_efficiency
```

The discharging efficiency of the asset.

4.11.4.10 dynamic_energy_capacity_kWh

```
double LiIon::dynamic_energy_capacity_kWh
```

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

4.11.4.11 gas_constant_JmolK

```
double LiIon::gas_constant_JmolK
```

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

4.11.4.12 hysteresis_SOC

```
double LiIon::hysteresis_SOC
```

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

4.11.4.13 init_SOC

```
double LiIon::init_SOC
```

The initial state of charge of the asset.

4.11.4.14 max_SOC

```
double LiIon::max_SOC
```

The maximum state of charge of the asset.

4.11.4.15 min_SOC

```
double LiIon::min_SOC
```

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

4.11.4.16 replace_SOH

```
double LiIon::replace_SOH
```

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

4.11.4.17 SOH

```
double LiIon::SOH
```

The state of health of the asset.

4.11.4.18 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.

4.11.4.19 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:

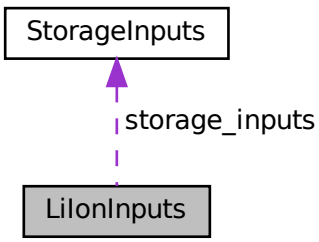
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

4.12 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.
- 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.

4.12.1 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 [2023]

4.12.2 Member Data Documentation

4.12.2.1 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].

4.12.2.2 charging_efficiency

```
double LiIonInputs::charging_efficiency = 0.9
```

The charging efficiency of the asset.

4.12.2.3 degradation_a_cal

```
double LiIonInputs::degradation_a_cal = 100
```

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

4.12.2.4 degradation_alpha

```
double LiIonInputs::degradation_alpha = 8.935
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.12.2.5 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.

4.12.2.6 degradation_beta

```
double LiIonInputs::degradation_beta = 1
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.12.2.7 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.

4.12.2.8 degradation_r_cal

```
double LiIonInputs::degradation_r_cal = 0.4361
```

A dimensionless constant used in modelling energy capacity degradation.

4.12.2.9 degradation_s_cal

```
double LiIonInputs::degradation_s_cal = 2
```

A dimensionless constant used in modelling energy capacity degradation.

4.12.2.10 discharging_efficiency

```
double LiIonInputs::discharging_efficiency = 0.9
```

The discharging efficiency of the asset.

4.12.2.11 gas_constant_JmolK

```
double LiIonInputs::gas_constant_JmolK = 8.31446
```

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

4.12.2.12 hysteresis_SOC

```
double LiIonInputs::hysteresis_SOC = 0.5
```

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

4.12.2.13 init_SOC

```
double LiIonInputs::init_SOC = 0.5
```

The initial state of charge of the asset.

4.12.2.14 max_SOC

```
double LiIonInputs::max_SOC = 0.9
```

The maximum state of charge of the asset.

4.12.2.15 min_SOC

```
double LiIonInputs::min_SOC = 0.15
```

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

4.12.2.16 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].

4.12.2.17 replace_SOH

```
double LiIonInputs::replace_SOH = 0.8
```

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

4.12.2.18 storage_inputs

```
StorageInputs LiIonInputs::storage_inputs
```

An encapsulated StorageInputs instance.

4.12.2.19 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:

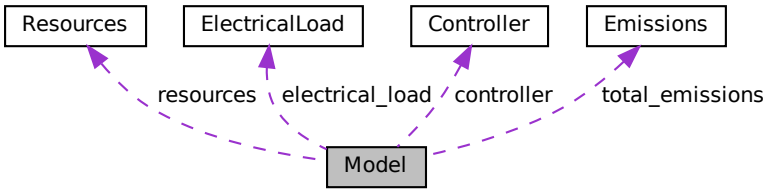
- header/Storage/LiIon.h

4.13 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 (RenewableType, std::string, int)

  A method to add a renewable resource time series 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_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< 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.
- 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.

4.13.1 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.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 Model() [1/2]

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

Constructor (dummy) for the Model class.

```
507 {
508     return;
509 }   /* Model() */
```

4.13.2.2 Model() [2/2]

```
Model::Model (
            ModelInputs model_inputs )
```

Constructor (intended) for the Model class.

Parameters

model_inputs	A structure of Model constructor inputs.

```
526 {
527     //  1. check inputs
528     this->__checkInputs(model_inputs);
529 
530     //  2. read in electrical load data
531     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
532 
533     //  3. set control mode
534     this->controller.setControlMode(model_inputs.control_mode);
535 
536     //  4. set public attributes
537     this->total_fuel_consumed_L = 0;
538     this->net_present_cost = 0;
539     this->total_dispatch_discharge_kWh = 0;
540     this->levellized_cost_of_energy_kWh = 0;
541 
542     return;
543 }   /* Model() */
```

4.13.2.3 ~Model()

```
Model::~Model (
              void  )
```

Destructor for the Model class.

```
970 {
971     this->clear();
972     return;
973 }   /* ~Model() */
```

4.13.3 Member Function Documentation

4.13.3.1 __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() */
```

4.13.3.2 __computeEconomics()

```
void Model::__computeEconomics (
              void  )  [private]
```

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

```
216 {
217     this->__computeNetPresentCost();
218     this->__computeLevellizedCostOfEnergy();
219
220     return;
221 }   /* __computeEconomics() */
```

4.13.3.3 __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() */
```

4.13.3.4 __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.

```
170 {
171     //  1. account for Combustion economics in levellized cost of energy
172     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
173         this->levellized_cost_of_energy_kWh +=
174             (
175                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
176                 this->combustion_ptr_vec[i]->total_dispatch_kWh
177             ) / this->total_dispatch_discharge_kWh;
178     }
179
180     //  2. account for Renewable economics in levellized cost of energy
181     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
182         this->levellized_cost_of_energy_kWh +=
183             (
184                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
185                 this->renewable_ptr_vec[i]->total_dispatch_kWh
186             ) / this->total_dispatch_discharge_kWh;
187     }
188
189     //  3. account for Storage economics in levellized cost of energy
190     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
191         /*
192         this->levellized_cost_of_energy_kWh +=
193             (
194                 this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
195                 this->storage_ptr_vec[i]->total_discharge_kWh
196             ) / this->total_dispatch_discharge_kWh;
197         */
198     }
199
200     return;
201 }   /* __computeLevellizedCostOfEnergy() */
```

4.13.3.5 __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.

```
113 {
114     //  1. account for Combustion economics in net present cost
115     //     increment total dispatch
116     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
117         this->combustion_ptr_vec[i]->computeEconomics(
118             &(this->electrical_load.time_vec_hrs)
119         );
120
121         this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
122
123         this->total_dispatch_discharge_kWh +=
124             this->combustion_ptr_vec[i]->total_dispatch_kWh;
125     }
126
127     //  2. account for Renewable economics in net present cost,
128     //     increment total dispatch
129     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
130         this->renewable_ptr_vec[i]->computeEconomics(
131             &(this->electrical_load.time_vec_hrs)
132         );
133
134         this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
135
136         this->total_dispatch_discharge_kWh +=
137             this->renewable_ptr_vec[i]->total_dispatch_kWh;
138     }
139
140     //  3. account for Storage economics in net present cost
```

```
141     //     increment total dispatch
142     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
143         this->storage_ptr_vec[i]->computeEconomics(
144             &(this->electrical_load.time_vec_hrs)
145         );
146 
147         this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
148 
149         this->total_dispatch_discharge_kWh +=
150             this->storage_ptr_vec[i]->total_discharge_kWh;
151     }
152 
153     return;
154 }   /* __computeNetPresentCost() */
```

4.13.3.6 __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.

```
239 {
240     //  1. create subdirectory
241     write_path += "Model/";
242     std::filesystem::create_directory(write_path);
243 
244     //  2. create filestream
245     write_path += "summary_results.md";
246     std::ofstream ofs;
247     ofs.open(write_path, std::ofstream::out);
248 
249     //  3. write summary results (markdown)
250     ofs « "# Model Summary Results\n";
251     ofs « "\n--------\n\n";
252 
253     //  3.1. ElectricalLoad
254     ofs « "## Electrical Load\n";
255     ofs « "\n";
256     ofs « "Path: " «
257         this->electrical_load.path_2_electrical_load_time_series « "  \n";
258     ofs « "Data Points: " « this->electrical_load.n_points « "  \n";
259     ofs « "Years: " « this->electrical_load.n_years « "  \n";
260     ofs « "Min: " « this->electrical_load.min_load_kW « " kW  \n";
261     ofs « "Mean: " « this->electrical_load.mean_load_kW « " kW  \n";
262     ofs « "Max: " « this->electrical_load.max_load_kW « " kW  \n";
263     ofs « "\n--------\n\n";
264 
265     //  3.2. Controller
266     ofs « "## Controller\n";
267     ofs « "\n";
268     ofs « "Control Mode: " « this->controller.control_string « "  \n";
269     ofs « "\n--------\n\n";
270 
271     //  3.3. Resources (1D)
272     ofs « "## 1D Renewable Resources\n";
273     ofs « "\n";
274 
275     std::map<int, std::string>::iterator string_map_1D_iter =
276         this->resources.string_map_1D.begin();
277     std::map<int, std::string>::iterator path_map_1D_iter =
278         this->resources.path_map_1D.begin();
279 
280     while (
281         string_map_1D_iter != this->resources.string_map_1D.end() and
282         path_map_1D_iter != this->resources.path_map_1D.end()
283     ) {
284         ofs « "Resource Key: " « string_map_1D_iter->first « "  \n";
285         ofs « "Type: " « string_map_1D_iter->second « "  \n";
```

```
286         ofs « "Path: " « path_map_1D_iter->second « "  \n";
287         ofs « "\n";
288 
289         string_map_1D_iter++;
290         path_map_1D_iter++;
291     }
292 
293     ofs « "\n--------\n\n";
294 
295     //  3.4. Resources (2D)
296     ofs « "## 2D Renewable Resources\n";
297     ofs « "\n";
298 
299     std::map<int, std::string>::iterator string_map_2D_iter =
300         this->resources.string_map_2D.begin();
301     std::map<int, std::string>::iterator path_map_2D_iter =
302         this->resources.path_map_2D.begin();
303 
304     while (
305         string_map_2D_iter != this->resources.string_map_2D.end() and
306         path_map_2D_iter != this->resources.path_map_2D.end()
307     ) {
308         ofs « "Resource Key: " « string_map_2D_iter->first « "  \n";
309         ofs « "Type: " « string_map_2D_iter->second « "  \n";
310         ofs « "Path: " « path_map_2D_iter->second « "  \n";
311         ofs « "\n";
312 
313         string_map_2D_iter++;
314         path_map_2D_iter++;
315     }
316 
317     ofs « "\n--------\n\n";
318 
319     //  3.5. Combustion
320     ofs « "## Combustion Assets\n";
321     ofs « "\n";
322 
323     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
324         ofs « "Asset Index: " « i « "  \n";
325         ofs « "Type: " « this->combustion_ptr_vec[i]->type_str « "  \n";
326         ofs « "Capacity: " « this->combustion_ptr_vec[i]->capacity_kW « " kW  \n";
327         ofs « "\n";
328     }
329 
330     ofs « "\n--------\n\n";
331 
332     //  3.6. Renewable
333     ofs « "## Renewable Assets\n";
334     ofs « "\n";
335 
336     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
337         ofs « "Asset Index: " « i « "  \n";
338         ofs « "Type: " « this->renewable_ptr_vec[i]->type_str « "  \n";
339         ofs « "Capacity: " « this->renewable_ptr_vec[i]->capacity_kW « " kW  \n";
340         ofs « "\n";
341     }
342 
343     ofs « "\n--------\n\n";
344 
345     //  3.7. Storage
346     ofs « "## Storage Assets\n";
347     ofs « "\n";
348 
349     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
350         //...
351     }
352 
353     ofs « "\n--------\n\n";
354 
355     //  3.8. Model Results
356     ofs « "## Results\n";
357     ofs « "\n";
358 
359     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
360     ofs « "\n";
361 
362     ofs « "Total Dispatch + Discharge: " « this->total_dispatch_discharge_kWh
363         « " kWh  \n";
364 
365     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
366         « " per kWh dispatched/discharged  \n";
367     ofs « "\n";
368 
369     ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
370         « "(Annual Average: " «
371             this->total_fuel_consumed_L / this->electrical_load.n_years
372         « " L/yr)  \n";
```

```
373     ofs « "\n";
374
375     ofs « "Total Carbon Dioxide (CO2) Emissions: " «
376         this->total_emissions.CO2_kg « " kg "
377         « "(Annual Average: " «
378             this->total_emissions.CO2_kg / this->electrical_load.n_years
379         « " kg/yr)  \n";
380
381     ofs « "Total Carbon Monoxide (CO) Emissions: " «
382         this->total_emissions.CO_kg « " kg "
383         « "(Annual Average: " «
384             this->total_emissions.CO_kg / this->electrical_load.n_years
385         « " kg/yr)  \n";
386
387     ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
388         this->total_emissions.NOx_kg « " kg "
389         « "(Annual Average: " «
390             this->total_emissions.NOx_kg / this->electrical_load.n_years
391         « " kg/yr)  \n";
392
393     ofs « "Total Sulfur Oxides (SOx) Emissions: " «
394         this->total_emissions.SOx_kg « " kg "
395         « "(Annual Average: " «
396             this->total_emissions.SOx_kg / this->electrical_load.n_years
397         « " kg/yr)  \n";
398
399     ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
400         « "(Annual Average: " «
401             this->total_emissions.CH4_kg / this->electrical_load.n_years
402         « " kg/yr)  \n";
403
404     ofs « "Total Particulate Matter (PM) Emissions: " «
405         this->total_emissions.PM_kg « " kg "
406         « "(Annual Average: " «
407             this->total_emissions.PM_kg / this->electrical_load.n_years
408         « " kg/yr)  \n";
409
410     ofs « "\n--------\n\n";
411
412     ofs.close();
413     return;
414 }   /* __writeSummary() */
```

4.13.3.7 __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.

```
434 {
435     //  1. create filestream
436     write_path += "Model/time_series_results.csv";
437     std::ofstream ofs;
438     ofs.open(write_path, std::ofstream::out);
439
440     //  2. write time series results header (comma separated value)
441     ofs « "Time (since start of data) [hrs],";
442     ofs « "Electrical Load [kW],";
443     ofs « "Net Load [kW],";
444     ofs « "Missed Load [kW],";
445
446     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
447         ofs « this->renewable_ptr_vec[i]->capacity_kW « " kW "
448             « this->renewable_ptr_vec[i]->type_str « " Dispatch [kW],";
449     }
```

```
450
451     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
452         //...
453     }
454
455     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
456         ofs « this->combustion_ptr_vec[i]->capacity_kW « " kW "
457             « this->combustion_ptr_vec[i]->type_str « " Dispatch [kW],";
458     }
459
460     ofs « "\n";
461
462     //  3. write time series results values (comma separated value)
463     for (int i = 0; i < max_lines; i++) {
464         //  3.1. load values
465         ofs « this->electrical_load.time_vec_hrs[i] « ",";
466         ofs « this->electrical_load.load_vec_kW[i] « ",";
467         ofs « this->controller.net_load_vec_kW[i] « ",";
468         ofs « this->controller.missed_load_vec_kW[i] « ",";
469
470         //  3.2. asset-wise dispatch/discharge
471         for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
472             ofs « this->renewable_ptr_vec[j]->dispatch_vec_kW[i] « ",";
473         }
474
475         for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
476             //...
477         }
478
479         for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
480             ofs « this->combustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
481         }
482
483         ofs « "\n";
484     }
485
486     ofs.close();
487     return;
488 }   /* __writeTimeSeries() */
```

4.13.3.8 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.

```
560 {
561     Combustion* diesel_ptr = new Diesel(
562         this->electrical_load.n_points,
563         this->electrical_load.n_years,
564         diesel_inputs
565     );
566
567     this->combustion_ptr_vec.push_back(diesel_ptr);
568
569     return;
570 }   /* addDiesel() */
```

4.13.3.9 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.

```
733 {
734     Storage* liion_ptr = new LiIon(
735         this->electrical_load.n_points,
736         this->electrical_load.n_years,
737         liion_inputs
738     );
739 
740     this->storage_ptr_vec.push_back(liion_ptr);
741 
742     return;
743 }   /* addLiIon() */
```

4.13.3.10 addResource()

```
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.

```
599 {
600     resources.addResource(
601         renewable_type,
602         path_2_resource_data,
603         resource_key,
604         &(this->electrical_load)
605     );
606 
607     return;
608 }   /* addResource() */
```

4.13.3.11 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.

```
625 {
626     Renewable* solar_ptr = new Solar(
```

```
627         this->electrical_load.n_points,
628         this->electrical_load.n_years,
629         solar_inputs
630     );
631
632     this->renewable_ptr_vec.push_back(solar_ptr);
633
634     return;
635 }   /* addSolar() */
```

4.13.3.12 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.

```
652 {
653     Renewable* tidal_ptr = new Tidal(
654         this->electrical_load.n_points,
655         this->electrical_load.n_years,
656         tidal_inputs
657     );
658
659     this->renewable_ptr_vec.push_back(tidal_ptr);
660
661     return;
662 }   /* addTidal() */
```

4.13.3.13 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.

```
679 {
680     Renewable* wave_ptr = new Wave(
681         this->electrical_load.n_points,
682         this->electrical_load.n_years,
683         wave_inputs
684     );
685
686     this->renewable_ptr_vec.push_back(wave_ptr);
687
688     return;
689 }   /* addWave() */
```

4.13.3.14 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.

```
706 {
707     Renewable* wind_ptr = new Wind(
708         this->electrical_load.n_points,
709         this->electrical_load.n_years,
710         wind_inputs
711     );
712
713     this->renewable_ptr_vec.push_back(wind_ptr);
714
715     return;
716 }   /* addWind() */
```

4.13.3.15 clear()

```
void Model::clear (
            void  )
```

Method to clear all attributes of the Model object.

```
849 {
850     //  1. reset
851     this->reset();
852
853     //  2. clear components
854     controller.clear();
855     electrical_load.clear();
856     resources.clear();
857
858     return;
859 }   /* clear() */
```

4.13.3.16 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.

```
800 {
801     //  1. clear combustion_ptr_vec
802     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
803         delete this->combustion_ptr_vec[i];
804     }
805     this->combustion_ptr_vec.clear();
806
807     //  2. clear renewable_ptr_vec
808     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
809         delete this->renewable_ptr_vec[i];
810     }
811     this->renewable_ptr_vec.clear();
812
```

```
813     //  3. clear storage_ptr_vec
814     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
815         delete this->storage_ptr_vec[i];
816     }
817     this->storage_ptr_vec.clear();
818 
819     //  4. reset attributes
820     this->total_fuel_consumed_L = 0;
821 
822     this->total_emissions.CO2_kg = 0;
823     this->total_emissions.CO_kg = 0;
824     this->total_emissions.NOx_kg = 0;
825     this->total_emissions.SOx_kg = 0;
826     this->total_emissions.CH4_kg = 0;
827     this->total_emissions.PM_kg = 0;
828 
829     this->net_present_cost = 0;
830     this->total_dispatch_discharge_kWh = 0;
831     this->levellized_cost_of_energy_kWh = 0;
832 
833     return;
834 }   /* reset() */
```

4.13.3.17 run()

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

A method to run the Model.

```
758 {
759     // 1. init Controller
760     this->controller.init(
761         &(this->electrical_load),
762         &(this->renewable_ptr_vec),
763         &(this->resources),
764         &(this->combustion_ptr_vec)
765     );
766 
767     //  2. apply dispatch control
768     this->controller.applyDispatchControl(
769         &(this->electrical_load),
770         &(this->combustion_ptr_vec),
771         &(this->renewable_ptr_vec),
772         &(this->storage_ptr_vec)
773     );
774 
775     //  3. compute total fuel consumption and emissions
776     this->__computeFuelAndEmissions();
777 
778     //  4. compute key economic metrics
779     this->__computeEconomics();
780 
781     return;
782 }   /* run() */
```

4.13.3.18 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.

```
887 {
888     //  1. handle sentinel
889     if (max_lines < 0) {
890         max_lines = this->electrical_load.n_points;
891     }
892 
893     //  2. check for pre-existing, warn (and remove), then create
894     if (write_path.back() != '/') {
895         write_path += '/';
896     }
897 
898     if (std::filesystem::is_directory(write_path)) {
899         std::string warning_str = "WARNING:  Model::writeResults():  ";
900         warning_str += write_path;
901         warning_str += " already exists, contents will be overwritten!";
902 
903         std::cout « warning_str « std::endl;
904 
905         std::filesystem::remove_all(write_path);
906     }
907 
908     std::filesystem::create_directory(write_path);
909 
910     //  3. write summary
911     this->__writeSummary(write_path);
912 
913     //  4. write time series
914     if (max_lines > this->electrical_load.n_points) {
915         max_lines = this->electrical_load.n_points;
916     }
917 
918     if (max_lines > 0) {
919         this->__writeTimeSeries(write_path, max_lines);
920     }
921 
922     //  5. call out to Combustion :: writeResults()
923     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
924         this->combustion_ptr_vec[i]->writeResults(
925             write_path,
926             &(this->electrical_load.time_vec_hrs),
927             i,
928             max_lines
929         );
930     }
931 
932     //  6. call out to Renewable :: writeResults()
933     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
934         this->renewable_ptr_vec[i]->writeResults(
935             write_path,
936             &(this->electrical_load.time_vec_hrs),
937             &(this->resources.resource_map_1D),
938             &(this->resources.resource_map_2D),
939             i,
940             max_lines
941         );
942     }
943 
944     //  7. call out to Storage :: writeResults()
945     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
946         this->storage_ptr_vec[i]->writeResults(
947             write_path,
948             &(this->electrical_load.time_vec_hrs),
949             i,
950             max_lines
951         );
952     }
953 
954     return;
955 }   /* writeResults() */
```

4.13.4 Member Data Documentation

4.13.4.1 combustion_ptr_vec

```
std::vector<Combustion*> Model::combustion_ptr_vec
```

A vector of pointers to the various Combustion assets in the Model.

4.13.4.2 controller

```
Controller Model::controller
```

Controller component of Model.

4.13.4.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

ElectricalLoad component of Model.

4.13.4.4 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).

4.13.4.5 net_present_cost

```
double Model::net_present_cost
```

The net present cost of the Model (undefined currency).

4.13.4.6 renewable_ptr_vec

```
std::vector<Renewable*> Model::renewable_ptr_vec
```

A vector of pointers to the various Renewable assets in the Model.

4.13.4.7 resources

```
Resources Model::resources
```

Resources component of Model.

4.13.4.8 storage_ptr_vec

```
std::vector<Storage*> Model::storage_ptr_vec
```

A vector of pointers to the various Storage assets in the Model.

4.13.4.9 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

The total energy dispatched/discharged [kWh] over the Model run.

4.13.4.10 total_emissions

```
Emissions Model::total_emissions
```

An Emissions structure for holding total emissions [kg].

4.13.4.11 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

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

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

- header/Model.h
- source/Model.cpp

4.14 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.

4.14.1 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).

4.14.2 Member Data Documentation

4.14.2.1 control_mode

```
ControlMode ModelInputs::control_mode = ControlMode ::  LOAD_FOLLOWING
```

The control mode to be applied by the Controller object.

4.14.2.2 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:

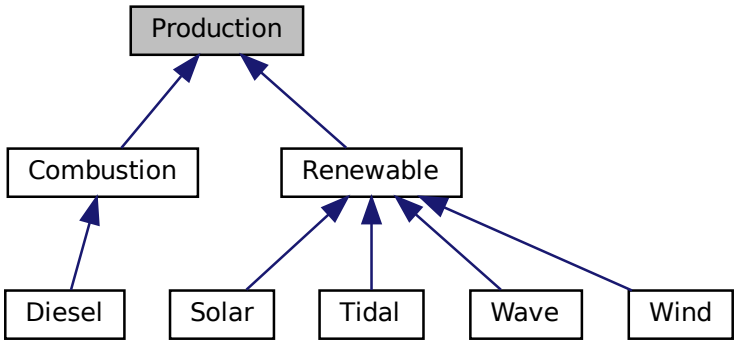
- header/Model.h

4.15 Production Class Reference

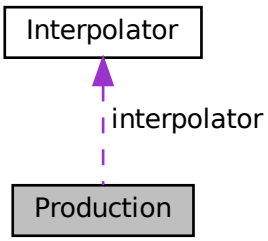
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

`#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)

  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.
- 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).
- 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::vector< bool > is_running_vec

  A boolean vector for tracking if the asset is running at a particular point in time.

- 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.

4.15.1 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.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Production() [1/2]

```
Production::Production (
            void  )
```

Constructor (dummy) for the Production class.

```
112 {
113     return;
114 }   /* Production() */
```

4.15.2.2 Production() [2/2]

```
Production::Production (
            int n_points,
            double n_years,
            ProductionInputs production_inputs )
```

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.

```
143 {
144     //  1. check inputs
145     this->__checkInputs(n_points, n_years, production_inputs);
146 
147     //  2. set attributes
148     this->print_flag = production_inputs.print_flag;
149     this->is_running = false;
150     this->is_sunk = production_inputs.is_sunk;
151 
152     this->n_points = n_points;
153     this->n_starts = 0;
154     this->n_replacements = 0;
155 
156     this->n_years = n_years;
157 
158     this->running_hours = 0;
159     this->replace_running_hrs = production_inputs.replace_running_hrs;
160 
161     this->capacity_kW = production_inputs.capacity_kW;
162 
163     this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
164     this->nominal_discount_annual = production_inputs.nominal_discount_annual;
165 
166     this->real_discount_annual = this->computeRealDiscountAnnual(
167         production_inputs.nominal_inflation_annual,
168         production_inputs.nominal_discount_annual
169     );
170 
171     this->capital_cost = 0;
172     this->operation_maintenance_cost_kWh = 0;
173     this->net_present_cost = 0;
174     this->total_dispatch_kWh = 0;
175     this->levellized_cost_of_energy_kWh = 0;
176 
177     this->is_running_vec.resize(this->n_points, 0);
178 
179     this->production_vec_kW.resize(this->n_points, 0);
180     this->dispatch_vec_kW.resize(this->n_points, 0);
181     this->storage_vec_kW.resize(this->n_points, 0);
182     this->curtailment_vec_kW.resize(this->n_points, 0);
183 
184     this->capital_cost_vec.resize(this->n_points, 0);
185     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
186 
187     //  3. construction print
188     if (this->print_flag) {
189         std::cout « "Production object constructed at " « this « std::endl;
190     }
191 
192     return;
193 }   /* Production() */
```

4.15.2.3 ~Production()

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

Destructor for the Production class.

```
411 {
412     //  1. destruction print
413     if (this->print_flag) {
414         std::cout « "Production object at " « this « " destroyed" « std::endl;
415     }
416 
417     return;
418 }   /* ~Production() */
```

4.15.3 Member Function Documentation

4.15.3.1 __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() */
```

4.15.3.2 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, Diesel, and Combustion.

```
352 {
353     //  1. record production
354     this->production_vec_kW[timestep] = production_kW;
355
356     //  2. compute and record dispatch and curtailment
357     double dispatch_kW = 0;
358     double curtailment_kW = 0;
359
360     if (production_kW > load_kW) {
361         dispatch_kW = load_kW;
362         curtailment_kW = production_kW - dispatch_kW;
363     }
364
365     else {
366         dispatch_kW = production_kW;
367     }
368
369     this->dispatch_vec_kW[timestep] = dispatch_kW;
370     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
371     this->curtailment_vec_kW[timestep] = curtailment_kW;
372
373     //  3. update load
374     load_kW -= dispatch_kW;
375
376     //  4. update and log running attributes
377     if (this->is_running) {
378         //  4.1. log running state, running hours
379         this->is_running_vec[timestep] = this->is_running;
380         this->running_hours += dt_hrs;
381
382         //  4.2. incur operation and maintenance costs
383         double produced_kWh = production_kW * dt_hrs;
384
385         double operation_maintenance_cost =
386             this->operation_maintenance_cost_kWh * produced_kWh;
```

```
387         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
388     }
389
390     //  5. trigger replacement, if applicable
391     if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
392         this->handleReplacement(timestep);
393     }
394
395     return load_kW;
396 }   /* commit() */
```

4.15.3.3 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, and Combustion.

```
281 {
282     //  1. compute net present cost
283     double t_hrs = 0;
284     double real_discount_scalar = 0;
285
286     for (int i = 0; i < this->n_points; i++) {
287         t_hrs = time_vec_hrs_ptr->at(i);
288
289         real_discount_scalar = 1.0 / pow(
290             1 + this->real_discount_annual,
291             t_hrs / 8760
292         );
293
294         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
295
296         this->net_present_cost +=
297             real_discount_scalar * this->operation_maintenance_cost_vec[i];
298     }
299
301     //      assuming 8,760 hours per year
302     double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
303
304     double capital_recovery_factor =
305         (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
306         (pow(1 + this->real_discount_annual, n_years) - 1);
307
308     double total_annualized_cost = capital_recovery_factor *
309         this->net_present_cost;
310
311     this->levellized_cost_of_energy_kWh =
312         (n_years * total_annualized_cost) /
313         this->total_dispatch_kWh;
314
315     return;
316 }   /* computeEconomics() */
```

4.15.3.4 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.

```
254 {
255     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
256     real_discount_annual /= 1 + nominal_inflation_annual;
257 
258     return real_discount_annual;
259 }   /* __computeRealDiscountAnnual() */
```

4.15.3.5 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, Diesel, and Combustion.

```
211 {
212     //  1. reset attributes
213     this->is_running = false;
214 
215     //  2. log replacement
216     this->n_replacements++;
217 
218     //  3. incur capital cost in timestep
219     this->capital_cost_vec[timestep] = this->capital_cost;
220 
221     return;
222 }   /* __handleReplacement() */
```

4.15.4 Member Data Documentation

4.15.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.15.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.15.4.3 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).

4.15.4.4 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.

4.15.4.5 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.

4.15.4.6 interpolator

```
Interpolator Production::interpolator
```

Interpolator component of Production.

4.15.4.7 is_running

```
bool Production::is_running
```

A boolean which indicates whether or not the asset is running.

4.15.4.8 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.

4.15.4.9 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).

4.15.4.10 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.

4.15.4.11 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.15.4.12 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.15.4.13 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.15.4.14 n_years

```
double Production::n_years
```

The number of years being modelled.

4.15.4.15 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.15.4.16 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

4.15.4.17 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

4.15.4.18 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.

4.15.4.19 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).

4.15.4.20 print_flag

```
bool Production::print_flag
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.15.4.21 production_vec_kW

```
std::vector<double> Production::production_vec_kW
```

A vector of production [kW] at each point in the modelling time series.

4.15.4.22 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.

4.15.4.23 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.15.4.24 running_hours

```
double Production::running_hours
```

The number of hours for which the assset has been operating.

4.15.4.25 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.

4.15.4.26 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

The total energy dispatched [kWh] over the Model run.

4.15.4.27 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

4.16 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.

4.16.1 Detailed Description

A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input.

4.16.2 Member Data Documentation

4.16.2.1 capacity_kW

```
double ProductionInputs::capacity_kW = 100
```

The rated production capacity [kW] of the asset.

4.16.2.2 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).

4.16.2.3 nominal_discount_annual

```
double ProductionInputs::nominal_discount_annual = 0.04
```

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

4.16.2.4 nominal_inflation_annual

```
double ProductionInputs::nominal_inflation_annual = 0.02
```

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

4.16.2.5 print_flag

```
bool ProductionInputs::print_flag = false
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.16.2.6 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:

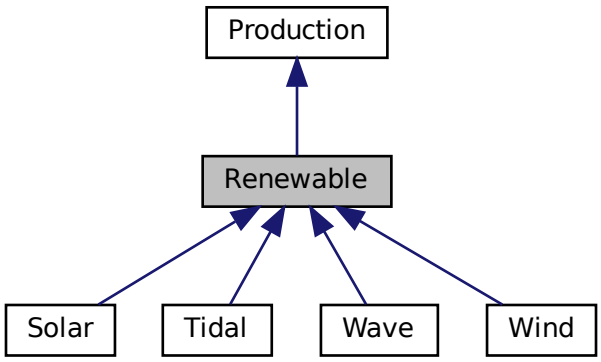
- header/Production/Production.h

4.17 Renewable Class Reference

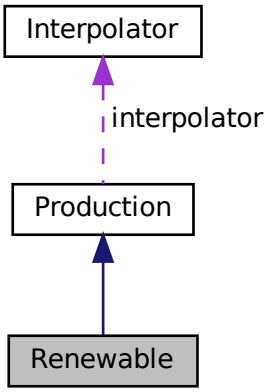
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#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)

  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)

4.17.1 Detailed Description

The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 Renewable() [1/2]

```
Renewable::Renewable (
            void  )
```

Constructor (dummy) for the Renewable class.

```
92 {
93     //...
94
95     return;
96 }   /* Renewable() */
```

4.17.2.2 Renewable() [2/2]

```
Renewable::Renewable (
            int n_points,
            double n_years,
            RenewableInputs renewable_inputs )
```

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.

```
124   :
125 Production(
126     n_points,
```

```
127     n_years,
128     renewable_inputs.production_inputs
129 )
130 {
131     //  1. check inputs
132     this->__checkInputs(renewable_inputs);
133 
134     //  2. set attributes
135     //...
136 
137     //  3. construction print
138     if (this->print_flag) {
139         std::cout « "Renewable object constructed at " « this « std::endl;
140     }
141 
142     return;
143 }   /* Renewable() */
```

4.17.2.3 ~Renewable()

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

Destructor for the Renewable class.

```
346 {
347     //  1. destruction print
348     if (this->print_flag) {
349         std::cout « "Renewable object at " « this « " destroyed" « std::endl;
350     }
351 
352     return;
353 }   /* ~Renewable() */
```

4.17.3 Member Function Documentation

4.17.3.1 __checkInputs()

```
void Renewable::__checkInputs (
            RenewableInputs renewable_inputs )  [private]
```

Helper method to check inputs to the Renewable constructor.

```
37 {
38     //...
39 
40     return;
41 }   /* __checkInputs() */
```

4.17.3.2 __handleStartStop()

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

Helper method to handle the starting/stopping of the renewable asset.

```
56 {
57     if (this->is_running) {
58         // handle stopping
59         if (production_kW <= 0) {
60             this->is_running = false;
61         }
62     }
63 
64     else {
65         // handle starting
66         if (production_kW > 0) {
67             this->is_running = true;
68             this->n_starts++;
69         }
70     }
71 
72     return;
73 }   /* __handleStartStop() */
```

4.17.3.3 __writeSummary()

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

Reimplemented in Wind, Wave, Tidal, and Solar.

```
72 {return;}
```

4.17.3.4 __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;}
```

4.17.3.5 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.

```
227 {
228     //  1. handle start/stop
229     this->__handleStartStop(timestep, dt_hrs, production_kW);
230 
231     //  2. invoke base class method
232     load_kW = Production :: commit(
233         timestep,
234         dt_hrs,
235         production_kW,
236         load_kW
237     );
238 
239 
240     //...
241 
242     return load_kW;
243 }   /* commit() */
```

4.17.3.6 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.

```
186 {
187     //  1. invoke base class method
188     Production :: computeEconomics(time_vec_hrs_ptr);
189 
190     return;
191 }   /* computeEconomics() */
```

4.17.3.7 computeProductionkW() [1/2]

```
virtual double Renewable::computeProductionkW (
             int ,
```

```
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Wind, Tidal, and Solar.

```
96 {return 0;}
```

4.17.3.8 computeProductionkW() [2/2]

```
virtual double Renewable::computeProductionkW (
            int ,
            double ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Wave.

```
97 {return 0;}
```

4.17.3.9 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.

```
161 {
162     //  1. reset attributes
163     //...
164
165     //  2. invoke base class method
166     Production :: handleReplacement(timestep);
167
168     return;
169 }   /* __handleReplacement() */
```

4.17.3.10 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.

```
287 {
288     //  1. handle sentinel
289     if (max_lines < 0) {
290         max_lines = this->n_points;
291     }
292
293     //  2. create subdirectories
294     write_path += "Production/";
295     if (not std::filesystem::is_directory(write_path)) {
296         std::filesystem::create_directory(write_path);
297     }
298
299     write_path += "Renewable/";
300     if (not std::filesystem::is_directory(write_path)) {
301         std::filesystem::create_directory(write_path);
302     }
303
304     write_path += this->type_str;
305     write_path += "_";
306     write_path += std::to_string(int(ceil(this->capacity_kW)));
307     write_path += "kW_idx";
308     write_path += std::to_string(renewable_index);
309     write_path += "/";
310     std::filesystem::create_directory(write_path);
311
312     //  3. write summary
313     this->__writeSummary(write_path);
314
315     //  4. write time series
316     if (max_lines > this->n_points) {
317         max_lines = this->n_points;
318     }
319
320     if (max_lines > 0) {
321         this->__writeTimeSeries(
322             write_path,
323             time_vec_hrs_ptr,
324             resource_map_1D_ptr,
325             resource_map_2D_ptr,
326             max_lines
327         );
328     }
329
330     return;
331 }   /* writeResults() */
```

4.17.4 Member Data Documentation

4.17.4.1 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.

4.17.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

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

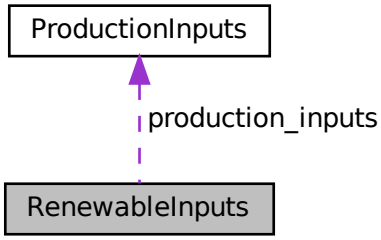
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

4.18 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.

4.18.1 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.

4.18.2 Member Data Documentation

4.18.2.1 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

4.19 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 (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 __checkTimePoint (double, double, std::string, ElectricalLoad ∗)

  Helper method to check received time point against expected time point.
- void __throwLengthError (std::string, ElectricalLoad ∗)

  Helper method to throw data length error.
- 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.

4.19.1 Detailed Description

A class which contains renewable resource data. Intended to serve as a component class of Model.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Resources()

```
Resources::Resources (
            void  )
```

Constructor for the Resources class.

```
577 {
578     return;
579 }   /* Resources() */
```

4.19.2.2 ∼Resources()

```
Resources::~Resources (
            void  )
```

Destructor for the Resources class.

```
721 {
722     this->clear();
723     return;
724 }   /* ~Resources() */
```

4.19.3 Member Function Documentation

4.19.3.1 __checkResourceKey1D()

```
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.

```
45 {
46     if (this->resource_map_1D.count(resource_key) > 0) {
47         std::string error_str = "ERROR:  Resources::addResource(";
48 
49         switch (renewable_type) {
50             case (RenewableType :: SOLAR): {
51                 error_str += "SOLAR):  ";
52 
53                 break;
54             }
55 
56             case (RenewableType :: TIDAL): {
57                 error_str += "TIDAL):  ";
58 
59                 break;
60             }
61 
62             case (RenewableType :: WIND): {
63                 error_str += "WIND):  ";
64 
65                 break;
66             }
67 
68             default: {
69                 error_str += "UNDEFINED_TYPE):  ";
70 
71                 break;
72             }
73         }
74 
75         error_str += "resource key (1D) ";
76         error_str += std::to_string(resource_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 }   /*  __checkResourceKey1D() */
```

4.19.3.2 __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.

```
109 {
110     if (this->resource_map_2D.count(resource_key) > 0) {
111         std::string error_str = "ERROR:  Resources::addResource(";
112 
113         switch (renewable_type) {
114             case (RenewableType :: WAVE): {
115                 error_str += "WAVE):  ";
116 
117                 break;
118             }
119 
120             default: {
121                 error_str += "UNDEFINED_TYPE):  ";
122 
123                 break;
124             }
125         }
126 
127         error_str += "resource key (2D) ";
128         error_str += std::to_string(resource_key);
129         error_str += " is already in use";
130 
131         #ifdef _WIN32
132             std::cout << error_str << std::endl;
133         #endif
134 
135         throw std::invalid_argument(error_str);
136     }
137 
138     return;
139 }   /*  __checkResourceKey2D() */
```

4.19.3.3 __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.

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.

```
173 {
174     if (time_received_hrs != time_expected_hrs) {
175         std::string error_str = "ERROR:  Resources::addResource():  ";
176         error_str += "the given resource time series at ";
177         error_str += path_2_resource_data;
178         error_str += " does not align with the ";
179         error_str += "previously given electrical load time series at ";
180         error_str += electrical_load_ptr->path_2_electrical_load_time_series;
181 
182         #ifdef _WIN32
183             std::cout << error_str << std::endl;
184         #endif
185 
186         throw std::runtime_error(error_str);
187     }
188 
```

```
189     return;
190 }   /* __checkTimePoint() */
```

4.19.3.4 __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.

```
257 {
258     //  1. init CSV reader, record path and type
259     io::CSVReader<2> CSV(path_2_resource_data);
260 
261     CSV.read_header(
262         io::ignore_extra_column,
263         "Time (since start of data) [hrs]",
264         "Solar GHI [kW/m2]"
265     );
266 
267     this->path_map_1D.insert(
268         std::pair<int, std::string>(resource_key, path_2_resource_data)
269     );
270 
271     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));
272 
273     //  2. init map element
274     this->resource_map_1D.insert(
275         std::pair<int, std::vector<double»(resource_key, {})
276     );
277     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
278 
279 
280     //  3. read in resource data, check against time series (point-wise and length)
281     int n_points = 0;
282     double time_hrs = 0;
283     double time_expected_hrs = 0;
284     double solar_resource_kWm2 = 0;
285 
286     while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
287         if (n_points > electrical_load_ptr->n_points) {
288             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
289         }
290 
291         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
292         this->__checkTimePoint(
293             time_hrs,
294             time_expected_hrs,
295             path_2_resource_data,
296             electrical_load_ptr
297         );
298 
299         this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
300 
301         n_points++;
302     }
303 
304     //  4. check data length
305     if (n_points != electrical_load_ptr->n_points) {
306         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
307     }
308 
309     return;
310 }   /* __readSolarResource() */
```

4.19.3.5 __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.

```
339 {
340     //  1. init CSV reader, record path and type
341     io::CSVReader<2> CSV(path_2_resource_data);
342 
343     CSV.read_header(
344         io::ignore_extra_column,
345         "Time (since start of data) [hrs]",
346         "Tidal Speed (hub depth) [m/s]"
347     );
348 
349     this->path_map_1D.insert(
350         std::pair<int, std::string>(resource_key, path_2_resource_data)
351     );
352 
353     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));
354 
355     //  2. init map element
356     this->resource_map_1D.insert(
357         std::pair<int, std::vector<double»(resource_key, {})
358     );
359     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
360 
361 
362     //  3. read in resource data, check against time series (point-wise and length)
363     int n_points = 0;
364     double time_hrs = 0;
365     double time_expected_hrs = 0;
366     double tidal_resource_ms = 0;
367 
368     while (CSV.read_row(time_hrs, tidal_resource_ms)) {
369         if (n_points > electrical_load_ptr->n_points) {
370             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
371         }
372 
373         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
374         this->__checkTimePoint(
375             time_hrs,
376             time_expected_hrs,
377             path_2_resource_data,
378             electrical_load_ptr
379         );
380 
381         this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;
382 
383         n_points++;
384     }
385 
386     //  4. check data length
387     if (n_points != electrical_load_ptr->n_points) {
388         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
389     }
390 
391     return;
392 }   /* __readTidalResource() */
```

4.19.3.6 __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.

```
421 {
422     //  1. init CSV reader, record path and type
423     io::CSVReader<3> CSV(path_2_resource_data);
424 
425     CSV.read_header(
426         io::ignore_extra_column,
427         "Time (since start of data) [hrs]",
428         "Significant Wave Height [m]",
429         "Energy Period [s]"
430     );
431 
432     this->path_map_2D.insert(
433         std::pair<int, std::string>(resource_key, path_2_resource_data)
434     );
435 
436     this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));
437 
438     //  2. init map element
439     this->resource_map_2D.insert(
440         std::pair<int, std::vector<std::vector<double»>(resource_key, {})
441     );
442     this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});
443 
444 
445     //  3. read in resource data, check against time series (point-wise and length)
446     int n_points = 0;
447     double time_hrs = 0;
448     double time_expected_hrs = 0;
449     double significant_wave_height_m = 0;
450     double energy_period_s = 0;
451 
452     while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
453         if (n_points > electrical_load_ptr->n_points) {
454             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
455         }
456 
457         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
458         this->__checkTimePoint(
459             time_hrs,
460             time_expected_hrs,
461             path_2_resource_data,
462             electrical_load_ptr
463         );
464 
465         this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
466         this->resource_map_2D[resource_key][n_points][1] = energy_period_s;
467 
468         n_points++;
469     }
470 
471     //  4. check data length
472     if (n_points != electrical_load_ptr->n_points) {
473         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
474     }
475 
476     return;
477 }   /* __readWaveResource() */
```

4.19.3.7 __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.

```
506 {
507     //  1. init CSV reader, record path and type
508     io::CSVReader<2> CSV(path_2_resource_data);
509 
510     CSV.read_header(
511         io::ignore_extra_column,
512         "Time (since start of data) [hrs]",
513         "Wind Speed (hub height) [m/s]"
514     );
515 
516     this->path_map_1D.insert(
517         std::pair<int, std::string>(resource_key, path_2_resource_data)
518     );
519 
520     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));
521 
522     //  2. init map element
523     this->resource_map_1D.insert(
524         std::pair<int, std::vector<double»(resource_key, {})
525     );
526     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
527 
528 
529     //  3. read in resource data, check against time series (point-wise and length)
530     int n_points = 0;
531     double time_hrs = 0;
532     double time_expected_hrs = 0;
533     double wind_resource_ms = 0;
534 
535     while (CSV.read_row(time_hrs, wind_resource_ms)) {
536         if (n_points > electrical_load_ptr->n_points) {
537             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
538         }
539 
540         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
541         this->__checkTimePoint(
542             time_hrs,
543             time_expected_hrs,
544             path_2_resource_data,
545             electrical_load_ptr
546         );
547 
548         this->resource_map_1D[resource_key][n_points] = wind_resource_ms;
549 
550         n_points++;
551     }
552 
553     //  4. check data length
554     if (n_points != electrical_load_ptr->n_points) {
555         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
556     }
557 
558     return;
559 }   /* __readWindResource() */
```

4.19.3.8 __throwLengthError()

```
void Resources::__throwLengthError (
```

```
            std::string path_2_resource_data,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to throw data length error.

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.

```
215 {
216     std::string error_str = "ERROR:  Resources::addResource():  ";
217     error_str += "the given resource time series at ";
218     error_str += path_2_resource_data;
219     error_str += " is not the same length as the previously given electrical";
220     error_str += " load time series at ";
221     error_str += electrical_load_ptr->path_2_electrical_load_time_series;
222 
223     #ifdef _WIN32
224         std::cout << error_str << std::endl;
225     #endif
226 
227     throw std::runtime_error(error_str);
228 
229     return;
230 }   /* __throwLengthError() */
```

4.19.3.9 addResource()

```
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.

```
616 {
617     switch (renewable_type) {
618         case (RenewableType :: SOLAR): {
619             this->__checkResourceKey1D(resource_key, renewable_type);
620 
621             this->__readSolarResource(
622                 path_2_resource_data,
623                 resource_key,
624                 electrical_load_ptr
625             );
626 
627             break;
628         }
629 
630         case (RenewableType :: TIDAL): {
631             this->__checkResourceKey1D(resource_key, renewable_type);
```

```
632
633             this->__readTidalResource(
634                 path_2_resource_data,
635                 resource_key,
636                 electrical_load_ptr
637             );
638
639             break;
640         }
641
642         case (RenewableType :: WAVE): {
643             this->__checkResourceKey2D(resource_key, renewable_type);
644
645             this->__readWaveResource(
646                 path_2_resource_data,
647                 resource_key,
648                 electrical_load_ptr
649             );
650
651             break;
652         }
653
654         case (RenewableType :: WIND): {
655             this->__checkResourceKey1D(resource_key, renewable_type);
656
657             this->__readWindResource(
658                 path_2_resource_data,
659                 resource_key,
660                 electrical_load_ptr
661             );
662
663             break;
664         }
665
666         default: {
667             std::string error_str = "ERROR:  Resources :: addResource(:  ";
668             error_str += "renewable type ";
669             error_str += std::to_string(renewable_type);
670             error_str += " not recognized";
671
672             #ifdef _WIN32
673                 std::cout « error_str « std::endl;
674             #endif
675
676             throw std::runtime_error(error_str);
677
678             break;
679         }
680     }
681
682     return;
683 }   /* addResource() */
```

4.19.3.10 clear()

```
void Resources::clear (
              void  )
```

Method to clear all attributes of the Resources object.

```
697 {
698     this->resource_map_1D.clear();
699     this->string_map_1D.clear();
700     this->path_map_1D.clear();
701
702     this->resource_map_2D.clear();
703     this->string_map_2D.clear();
704     this->path_map_2D.clear();
705
706     return;
707 }   /* clear() */
```

4.19.4 Member Data Documentation

4.19.4.1 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.

4.19.4.2 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.

4.19.4.3 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.

4.19.4.4 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.

4.19.4.5 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.

4.19.4.6 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:

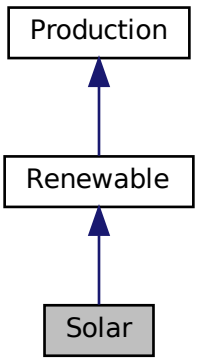
- header/Resources.h
- source/Resources.cpp

4.20 Solar Class Reference

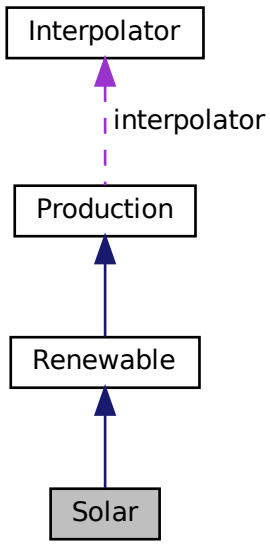
A derived class of the Renewable branch of Production which models solar production.

`#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)

  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.

4.20.1 Detailed Description

A derived class of the Renewable branch of Production which models solar production.

4.20.2 Constructor & Destructor Documentation

4.20.2.1 Solar() [1/2]

```
Solar::Solar (
            void  )
```

Constructor (dummy) for the Solar class.

```
281 {
282     //...
283 
284     return;
285 }   /* Solar() */
```

4.20.2.2 Solar() [2/2]

```
Solar::Solar (
            int n_points,
            double n_years,
            SolarInputs solar_inputs )
```

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.

```
313   :
314 Renewable(
315     n_points,
316     n_years,
317     solar_inputs.renewable_inputs
318 )
319 {
320     //  1. check inputs
321     this->__checkInputs(solar_inputs);
322 
323     //  2. set attributes
324     this->type = RenewableType :: SOLAR;
325     this->type_str = "SOLAR";
326 
327     this->resource_key = solar_inputs.resource_key;
328 
329     this->derating = solar_inputs.derating;
330 
331     if (solar_inputs.capital_cost < 0) {
332         this->capital_cost = this->__getGenericCapitalCost();
333     }
334 
335     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
336         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
337     }
338 
339     if (not this->is_sunk) {
340         this->capital_cost_vec[0] = this->capital_cost;
341     }
342 
343     //  3. construction print
344     if (this->print_flag) {
345         std::cout « "Solar object constructed at " « this « std::endl;
346     }
347 
348     return;
349 }   /* Renewable() */
```

4.20.2.3 ~Solar()

```
Solar::~Solar (
            void  )
```

Destructor for the Solar class.

```
488 {
489     //  1. destruction print
490     if (this->print_flag) {
491         std::cout « "Solar object at " « this « " destroyed" « std::endl;
492     }
493
494     return;
495 }   /* ~Solar() */
```

4.20.3 Member Function Documentation

4.20.3.1 __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() */
```

4.20.3.2 __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() */
```

4.20.3.3 __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() */
```

4.20.3.4 __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 « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
143     ofs « "Capital Cost: " « this->capital_cost « "  \n";
144     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
145         « " per kWh produced  \n";
146     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
147         « "  \n";
148     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
149         « "  \n";
150     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
151     ofs « "\n";
152 
153     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
154     ofs « "\n--------\n\n";
```

```
155
156     //  2.2. Renewable attributes
157     ofs « "## Renewable Attributes\n";
158     ofs « "\n";
159
160     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
161
162     ofs « "\n--------\n\n";
163
164     //  2.3. Solar attributes
165     ofs « "## Solar Attributes\n";
166     ofs « "\n";
167
168     ofs « "Derating Factor: " « this->derating « "  \n";
169
170     ofs « "\n--------\n\n";
171
172     //  2.4. Solar Results
173     ofs « "## Results\n";
174     ofs « "\n";
175
176     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
177     ofs « "\n";
178
179     ofs « "Total Dispatch: " « this->total_dispatch_kWh
180         « " kWh  \n";
181
182     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
183         « " per kWh dispatched  \n";
184     ofs « "\n";
185
186     ofs « "Running Hours: " « this->running_hours « "  \n";
187     ofs « "Replacements: " « this->n_replacements « "  \n";
188
189     ofs « "\n--------\n\n";
190
191     ofs.close();
192     return;
193 }   /* __writeSummary() */
```

4.20.3.5 __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.

```
231 {
232     //  1. create filestream
233     write_path += "time_series_results.csv";
234     std::ofstream ofs;
235     ofs.open(write_path, std::ofstream::out);
236
```

```
237     //  2. write time series results (comma separated value)
238     ofs « "Time (since start of data) [hrs],";
239     ofs « "Solar Resource [kW/m2],";
240     ofs « "Production [kW],";
241     ofs « "Dispatch [kW],";
242     ofs « "Storage [kW],";
243     ofs « "Curtailment [kW],";
244     ofs « "Capital Cost (actual),";
245     ofs « "Operation and Maintenance Cost (actual),";
246     ofs « "\n";
247
248     for (int i = 0; i < max_lines; i++) {
249         ofs « time_vec_hrs_ptr->at(i) « ",";
250         ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
251         ofs « this->production_vec_kW[i] « ",";
252         ofs « this->dispatch_vec_kW[i] « ",";
253         ofs « this->storage_vec_kW[i] « ",";
254         ofs « this->curtailment_vec_kW[i] « ",";
255         ofs « this->capital_cost_vec[i] « ",";
256         ofs « this->operation_maintenance_cost_vec[i] « ",";
257         ofs « "\n";
258     }
259
260     ofs.close();
261     return;
262 }   /* __writeTimeSeries() */
```

4.20.3.6 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.

```
460 {
461     //  1. invoke base class method
462     load_kW = Renewable :: commit(
463         timestep,
464         dt_hrs,
465         production_kW,
466         load_kW
467     );
468
469
470     //...
471
472     return load_kW;
473 }   /* commit() */
```

4.20.3.7 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.

```
409 {
410     // check if no resource
411     if (solar_resource_kWm2 <= 0) {
412         return 0;
413     }
414 
415     // compute production
416     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
417 
418     // cap production at capacity
419     if (production_kW > this->capacity_kW) {
420         production_kW = this->capacity_kW;
421     }
422 
423     return production_kW;
424 }   /* computeProductionkW() */
```

4.20.3.8 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.

```
367 {
368     //  1. reset attributes
369     //...
```

```
370
371     //  2. invoke base class method
372     Renewable :: handleReplacement(timestep);
373
374     return;
375 }   /* __handleReplacement() */
```

4.20.4 Member Data Documentation

4.20.4.1 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:

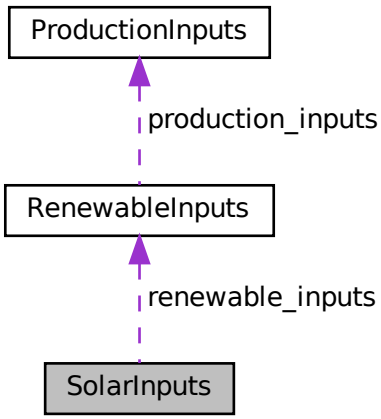
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

4.21 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.).

4.21.1 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.

4.21.2 Member Data Documentation

4.21.2.1 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].

4.21.2.2 derating

```
double SolarInputs::derating = 0.8
```

The derating of the solar PV array (i.e., shadowing, soiling, etc.).

4.21.2.3 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].

4.21.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.21.2.5 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:

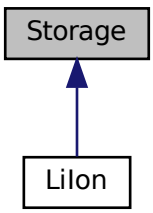
- header/Production/Renewable/Solar.h

4.22 Storage Class Reference

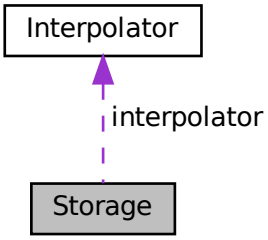
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

```
#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)

4.22.1 Detailed Description

The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

4.22.2 Constructor & Destructor Documentation

4.22.2.1 Storage() [1/2]

```
Storage::Storage (
            void  )
```

Constructor (dummy) for the Storage class.

```
151 {
152     return;
153 }   /* Storage() */
```

4.22.2.2 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() */
```

4.22.2.3 ~Storage()

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

Destructor for the Storage class.

```
408 {
409     //  1. destruction print
410     if (this->print_flag) {
411         std::cout << "Storage object at " << this << " destroyed" << std::endl;
412     }
413
414     return;
415 }   /* ~Storage() */
```

4.22.3 Member Function Documentation

4.22.3.1 __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() */
```

4.22.3.2 __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() */
```

4.22.3.3 __writeSummary()

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

Reimplemented in Lilon.

```
79 {return;}
```

4.22.3.4 __writeTimeSeries()

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

Reimplemented in Lilon.

```
80 {return;}
```

4.22.3.5 commitCharge()

```
virtual void Storage::commitCharge (
            int ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Lilon.

```
134 {return;}
```

4.22.3.6 commitDischarge()

```
virtual double Storage::commitDischarge (
            int ,
            double ,
            double ,
            double  )  [inline], [virtual]
```

Reimplemented in Lilon.

```
135 {return 0;}
```

4.22.3.7 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     double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
304 
305     double capital_recovery_factor =
306         (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
307         (pow(1 + this->real_discount_annual, n_years) - 1);
308 
309     double total_annualized_cost = capital_recovery_factor *
310         this->net_present_cost;
311 
312     this->levellized_cost_of_energy_kWh =
313         (n_years * total_annualized_cost) /
314         this->total_discharge_kWh;
315 
316     return;
317 }   /* computeEconomics() */
```

4.22.3.8 getAcceptablekW()

```
virtual double Storage::getAcceptablekW (
            double  )  [inline], [virtual]
```

Reimplemented in LiIon.

```
132 {return 0;}
```

4.22.3.9 getAvailablekW()

```
virtual double Storage::getAvailablekW (
            double  )  [inline], [virtual]
```

Reimplemented in LiIon.

```
131 {return 0;}
```

4.22.3.10 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 LiIon.

```
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() */
```

4.22.3.11 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.

```
354 {
355     //  1. handle sentinel
356     if (max_lines < 0) {
357         max_lines = this->n_points;
358     }
359 
360     //  2. create subdirectories
361     write_path += "Storage/";
362     if (not std::filesystem::is_directory(write_path)) {
363         std::filesystem::create_directory(write_path);
364     }
365 
366     write_path += this->type_str;
367     write_path += "_";
368     write_path += std::to_string(int(ceil(this->power_capacity_kW)));
369     write_path += "kW_";
370     write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
371     write_path += "kWh_idx";
372     write_path += std::to_string(storage_index);
373     write_path += "/";
374     std::filesystem::create_directory(write_path);
375 
376     //  3. write summary
377     this->__writeSummary(write_path);
378 
379     //  4. write time series
380     if (max_lines > this->n_points) {
381         max_lines = this->n_points;
382     }
383 
384     if (max_lines > 0) {
385         this->__writeTimeSeries(
386             write_path,
387             time_vec_hrs_ptr,
388             max_lines
389         );
390     }
391 
392     return;
393 }   /* writeResults() */
```

4.22.4 Member Data Documentation

4.22.4.1 capital_cost

`double Storage::capital_cost`

The capital cost of the asset (undefined currency).

4.22.4.2 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).

4.22.4.3 charge_kWh

```
double Storage::charge_kWh
```

The energy [kWh] stored in the asset.

4.22.4.4 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.

4.22.4.5 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.

4.22.4.6 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.

4.22.4.7 energy_capacity_kWh

```
double Storage::energy_capacity_kWh
```

The rated energy capacity [kWh] of the asset.

4.22.4.8 interpolator

```
Interpolator Storage::interpolator
```

Interpolator component of Storage.

4.22.4.9 is_depleted

```
bool Storage::is_depleted
```

A boolean which indicates whether or not the asset is currently considered depleted.

4.22.4.10 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).

4.22.4.11 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.

4.22.4.12 n_points

```
int Storage::n_points
```

The number of points in the modelling time series.

4.22.4.13 n_replacements

```
int Storage::n_replacements
```

The number of times the asset has been replaced.

4.22.4.14 n_years

```
double Storage::n_years
```

The number of years being modelled.

4.22.4.15 net_present_cost

```
double Storage::net_present_cost
```

The net present cost of this asset.

4.22.4.16 nominal_discount_annual

```
double Storage::nominal_discount_annual
```

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

4.22.4.17 nominal_inflation_annual

```
double Storage::nominal_inflation_annual
```

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

4.22.4.18 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.

4.22.4.19 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).

4.22.4.20 power_capacity_kW

```
double Storage::power_capacity_kW
```

The rated power capacity [kW] of the asset.

4.22.4.21 power_kW

```
double Storage::power_kW
```

The power [kW] currently being charged/discharged by the asset.

4.22.4.22 print_flag

```
bool Storage::print_flag
```

A flag which indicates whether or not object construct/destruction should be verbose.

4.22.4.23 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.

4.22.4.24 total_discharge_kWh

```
double Storage::total_discharge_kWh
```

The total energy discharged [kWh] over the Model run.

4.22.4.25 type

```
StorageType Storage::type
```

The type (StorageType) of the asset.

4.22.4.26 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

4.23 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.

4.23.1 Detailed Description

A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input.

4.23.2 Member Data Documentation

4.23.2.1 energy_capacity_kWh

```
double StorageInputs::energy_capacity_kWh = 1000
```

The rated energy capacity [kWh] of the asset.

4.23.2.2 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).

4.23.2.3 nominal_discount_annual

```
double StorageInputs::nominal_discount_annual = 0.04
```

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

4.23.2.4 nominal_inflation_annual

```
double StorageInputs::nominal_inflation_annual = 0.02
```

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

4.23.2.5 power_capacity_kW

```
double StorageInputs::power_capacity_kW = 100
```

The rated power capacity [kW] of the asset.

4.23.2.6 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:

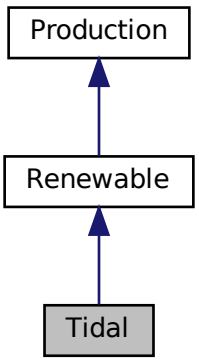
- header/Storage/Storage.h

4.24 Tidal Class Reference

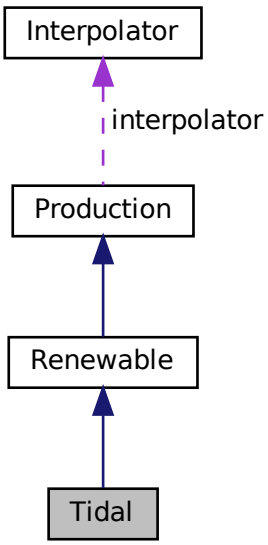
A derived class of the Renewable branch of Production which models tidal production.

`#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)

  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.

4.24.1 Detailed Description

A derived class of the Renewable branch of Production which models tidal production.

4.24.2 Constructor & Destructor Documentation

4.24.2.1 Tidal() [1/2]

```
Tidal::Tidal (
            void  )
```

Constructor (dummy) for the Tidal class.

```
427 {
428     return;
429 }   /* Tidal() */
```

4.24.2.2 Tidal() [2/2]

```
Tidal::Tidal (
            int n_points,
            double n_years,
            TidalInputs tidal_inputs )
```

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.

```
457   :
458 Renewable(
459     n_points,
460     n_years,
461     tidal_inputs.renewable_inputs
462 )
463 {
464     //  1. check inputs
465     this->__checkInputs(tidal_inputs);
466 
467     //  2. set attributes
468     this->type = RenewableType :: TIDAL;
469     this->type_str = "TIDAL";
470 
471     this->resource_key = tidal_inputs.resource_key;
472 
473     this->design_speed_ms = tidal_inputs.design_speed_ms;
474 
475     this->power_model = tidal_inputs.power_model;
476 
477     switch (this->power_model) {
478         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
479             this->power_model_string = "CUBIC";
480 
481             break;
482         }
483 
484         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
485             this->power_model_string = "EXPONENTIAL";
486 
487             break;
488         }
489 
490         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
```

```
491             this->power_model_string = "LOOKUP";
492 
493             break;
494         }
495 
496         default: {
497             std::string error_str = "ERROR:  Tidal():  ";
498             error_str += "power production model ";
499             error_str += std::to_string(this->power_model);
500             error_str += " not recognized";
501 
502             #ifdef _WIN32
503                 std::cout << error_str << std::endl;
504             #endif
505 
506             throw std::runtime_error(error_str);
507 
508             break;
509         }
510     }
511 
512     if (tidal_inputs.capital_cost < 0) {
513         this->capital_cost = this->__getGenericCapitalCost();
514     }
515 
516     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
517         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
518     }
519 
520     if (not this->is_sunk) {
521         this->capital_cost_vec[0] = this->capital_cost;
522     }
523 
524     //  3. construction print
525     if (this->print_flag) {
526         std::cout << "Tidal object constructed at " << this << std::endl;
527     }
528 
529     return;
530 }   /* Renewable() */
```

4.24.2.3 ~Tidal()

```
Tidal::~Tidal (
            void  )
```

Destructor for the Tidal class.

```
710 {
711     //  1. destruction print
712     if (this->print_flag) {
713         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
714     }
715 
716     return;
717 }   /* ~Tidal() */
```

4.24.3 Member Function Documentation

4.24.3.1 __checkInputs()

```
void Tidal::__checkInputs (
            TidalInputs tidal_inputs )  [private]
```

Helper method to check inputs to the Tidal constructor.

```
37 {
```

```
38      //  1. check design_speed_ms
39      if (tidal_inputs.design_speed_ms <= 0) {
40          std::string error_str = "ERROR:  Tidal():  ";
41          error_str += "TidalInputs::design_speed_ms must be > 0";
42
43          #ifdef _WIN32
44              std::cout « error_str « std::endl;
45          #endif
46
47          throw std::invalid_argument(error_str);
48      }
49
50      return;
51  }   /* __checkInputs() */
```

4.24.3.2 __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]

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.

```
138 {
139     double production = 0;
140
141     if (
142         tidal_resource_ms < 0.15 * this->design_speed_ms or
143         tidal_resource_ms > 1.25 * this->design_speed_ms
144     ){
145         production = 0;
146     }
147
148     else if (
149         0.15 * this->design_speed_ms <= tidal_resource_ms and
150         tidal_resource_ms <= this->design_speed_ms
151     ) {
152         production =
153             (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
154     }
155
156     else {
157         production = 1;
158     }
159
160     return production * this->capacity_kW;
161 }   /* __computeCubicProductionkW() */
```

4.24.3.3 __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.

```
195 {
196     double production = 0;
197 
198     double turbine_speed =
199         (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
200 
201     if (turbine_speed < -0.71 or turbine_speed > 0.65) {
202         production = 0;
203     }
204 
205     else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
206         production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
207     }
208 
209     else {
210         production = 1;
211     }
212 
213     return production * this->capacity_kW;
214 }   /* __computeExponentialProductionkW() */
```

4.24.3.4 __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.

```
246 {
247     // *** WORK IN PROGRESS *** //
248
249     return 0;
250 }   /* __computeLookupProductionkW() */
```

4.24.3.5 __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].

```
73 {
74     double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;
75
76     return capital_cost_per_kW * this->capacity_kW;
77 }   /* __getGenericCapitalCost() */
```

4.24.3.6 __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].

```
100 {
101     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
102
103     return operation_maintenance_cost_kWh;
104 }   /* __getGenericOpMaintCost() */
```

4.24.3.7 __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.

```
268 {
269     //  1. create filestream
270     write_path += "summary_results.md";
271     std::ofstream ofs;
272     ofs.open(write_path, std::ofstream::out);
273 
274     //  2. write summary results (markdown)
275     ofs « "# ";
276     ofs « std::to_string(int(ceil(this->capacity_kW)));
277     ofs « " kW TIDAL Summary Results\n";
278     ofs « "\n--------\n\n";
279 
280     //  2.1. Production attributes
281     ofs « "## Production Attributes\n";
282     ofs « "\n";
283 
284     ofs « "Capacity: " « this->capacity_kW « "kW  \n";
285     ofs « "\n";
286 
287     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
288     ofs « "Capital Cost: " « this->capital_cost « "  \n";
289     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
290         « " per kWh produced  \n";
291     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
292         « "  \n";
293     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
294         « "  \n";
295     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
296     ofs « "\n";
297 
298     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
299     ofs « "\n--------\n\n";
300 
301     //  2.2. Renewable attributes
302     ofs « "## Renewable Attributes\n";
303     ofs « "\n";
304 
305     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
306 
307     ofs « "\n--------\n\n";
308 
309     //  2.3. Tidal attributes
310     ofs « "## Tidal Attributes\n";
311     ofs « "\n";
312 
313     ofs « "Power Production Model: " « this->power_model_string « "  \n";
314     ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";
315 
316     ofs « "\n--------\n\n";
317 
318     //  2.4. Tidal Results
319     ofs « "## Results\n";
320     ofs « "\n";
321 
322     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
323     ofs « "\n";
324 
325     ofs « "Total Dispatch: " « this->total_dispatch_kWh
326         « " kWh  \n";
327 
328     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
329         « " per kWh dispatched  \n";
330     ofs « "\n";
331 
332     ofs « "Running Hours: " « this->running_hours « "  \n";
333     ofs « "Replacements: " « this->n_replacements « "  \n";
334 
335     ofs « "\n--------\n\n";
336 
337     ofs.close();
338 
339     return;
340 }   /* __writeSummary() */
```

4.24.3.8 __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.

```
378 {
379     //  1. create filestream
380     write_path += "time_series_results.csv";
381     std::ofstream ofs;
382     ofs.open(write_path, std::ofstream::out);
383 
384     //  2. write time series results (comma separated value)
385     ofs « "Time (since start of data) [hrs],";
386     ofs « "Tidal Resource [m/s],";
387     ofs « "Production [kW],";
388     ofs « "Dispatch [kW],";
389     ofs « "Storage [kW],";
390     ofs « "Curtailment [kW],";
391     ofs « "Capital Cost (actual),";
392     ofs « "Operation and Maintenance Cost (actual),";
393     ofs « "\n";
394 
395     for (int i = 0; i < max_lines; i++) {
396         ofs « time_vec_hrs_ptr->at(i) « ",";
397         ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
398         ofs « this->production_vec_kW[i] « ",";
399         ofs « this->dispatch_vec_kW[i] « ",";
400         ofs « this->storage_vec_kW[i] « ",";
401         ofs « this->curtailment_vec_kW[i] « ",";
402         ofs « this->capital_cost_vec[i] « ",";
403         ofs « this->operation_maintenance_cost_vec[i] « ",";
404         ofs « "\n";
405     }
406 
407     return;
408 }   /* __writeTimeSeries() */
```

4.24.3.9 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.

```
682 {
683     //  1. invoke base class method
684     load_kW = Renewable :: commit(
685         timestep,
686         dt_hrs,
687         production_kW,
688         load_kW
689     );
690 
691 
692     //...
693 
694     return load_kW;
695 }   /* commit() */
```

4.24.3.10 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.

```
588 {
589     // check if no resource
590     if (tidal_resource_ms <= 0) {
591         return 0;
592     }
593 
594     // compute production
595     double production_kW = 0;
```

```
596
597     switch (this->power_model) {
598         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
599             production_kW = this->__computeCubicProductionkW(
600                 timestep,
601                 dt_hrs,
602                 tidal_resource_ms
603             );
604
605             break;
606         }
607
608
609         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
610             production_kW = this->__computeExponentialProductionkW(
611                 timestep,
612                 dt_hrs,
613                 tidal_resource_ms
614             );
615
616             break;
617         }
618
619         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
620             production_kW = this->__computeLookupProductionkW(
621                 timestep,
622                 dt_hrs,
623                 tidal_resource_ms
624             );
625
626             break;
627         }
628
629         default: {
630             std::string error_str = "ERROR:  Tidal::computeProductionkW():  ";
631             error_str += "power model ";
632             error_str += std::to_string(this->power_model);
633             error_str += " not recognized";
634
635             #ifdef _WIN32
636                 std::cout « error_str « std::endl;
637             #endif
638
639             throw std::runtime_error(error_str);
640
641             break;
642         }
643     }
644
645     return production_kW;
646 }   /* computeProductionkW() */
```

4.24.3.11 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.

```
548 {
549     //  1. reset attributes
550     //...
551
552     //  2. invoke base class method
553     Renewable :: handleReplacement(timestep);
554
555     return;
556 }   /* __handleReplacement() */
```

4.24.4 Member Data Documentation

4.24.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.

4.24.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

4.24.4.3 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:

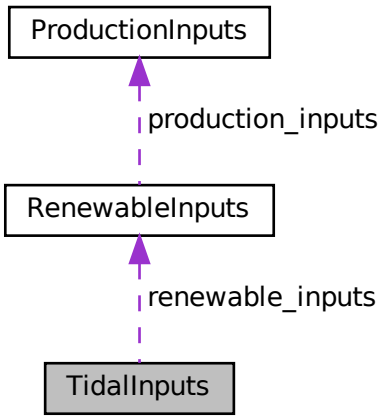
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.25 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.

4.25.1 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.

4.25.2 Member Data Documentation

4.25.2.1 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].

4.25.2.2 design_speed_ms

```
double TidalInputs::design_speed_ms = 3
```

The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.

4.25.2.3 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].

4.25.2.4 power_model

```
TidalPowerProductionModel TidalInputs::power_model = TidalPowerProductionModel ::  TIDAL_POWER_CUBIC
```

The tidal power production model to be applied.

4.25.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.25.2.6 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:

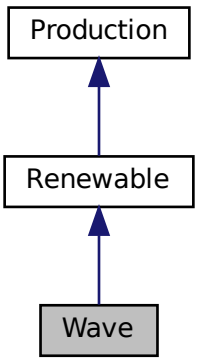
- header/Production/Renewable/Tidal.h

4.26 Wave Class Reference

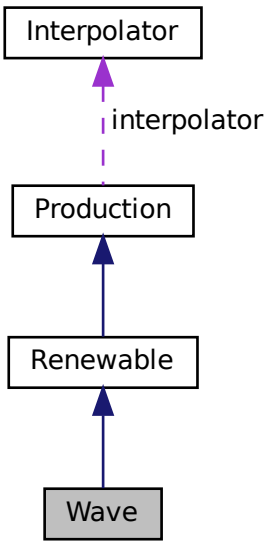
A derived class of the Renewable branch of Production which models wave production.

`#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)

  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.

4.26.1 Detailed Description

A derived class of the Renewable branch of Production which models wave production.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 Wave() [1/2]

```
Wave::Wave (
            void  )
```

Constructor (dummy) for the Wave class.

```
480 {
481     return;
482 }   /* Wave() */
```

4.26.2.2 Wave() [2/2]

```
Wave::Wave (
            int n_points,
            double n_years,
            WaveInputs wave_inputs )
```

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.

```
510   :
511 Renewable(
512     n_points,
513     n_years,
514     wave_inputs.renewable_inputs
515 )
516 {
517     //  1. check inputs
518     this->__checkInputs(wave_inputs);
519 
520     //  2. set attributes
521     this->type = RenewableType :: WAVE;
522     this->type_str = "WAVE";
523 
524     this->resource_key = wave_inputs.resource_key;
525 
526     this->design_significant_wave_height_m =
527         wave_inputs.design_significant_wave_height_m;
528     this->design_energy_period_s = wave_inputs.design_energy_period_s;
529 
530     this->power_model = wave_inputs.power_model;
531 
532     switch (this->power_model) {
533         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
534             this->power_model_string = "GAUSSIAN";
```

```
535
536             break;
537         }
538
539         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
540             this->power_model_string = "PARABOLOID";
541
542             break;
543         }
544
545         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
546             this->power_model_string = "LOOKUP";
547
548             break;
549         }
550
551         default: {
552             std::string error_str = "ERROR:  Wave():  ";
553             error_str += "power production model ";
554             error_str += std::to_string(this->power_model);
555             error_str += " not recognized";
556
557             #ifdef _WIN32
558                 std::cout << error_str << std::endl;
559             #endif
560
561             throw std::runtime_error(error_str);
562
563             break;
564         }
565     }
566
567     if (wave_inputs.capital_cost < 0) {
568         this->capital_cost = this->__getGenericCapitalCost();
569     }
570
571     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
572         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
573     }
574
575     if (not this->is_sunk) {
576         this->capital_cost_vec[0] = this->capital_cost;
577     }
578
579     //  3. construction print
580     if (this->print_flag) {
581         std::cout << "Wave object constructed at " << this << std::endl;
582     }
583
584     return;
585 }   /* Renewable() */
```

4.26.2.3 ~Wave()

```
Wave::~Wave (
            void  )
```

Destructor for the Wave class.

```
771 {
772     //  1. destruction print
773     if (this->print_flag) {
774         std::cout << "Wave object at " << this << " destroyed" << std::endl;
775     }
776
777     return;
778 }   /* ~Wave() */
```

4.26.3 Member Function Documentation

4.26.3.1 __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     return;
65 }   /* __checkInputs() */
```

4.26.3.2 __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.

```
160 {
161     double H_s_nondim =
162         (significant_wave_height_m - this->design_significant_wave_height_m) /
163         this->design_significant_wave_height_m;
164
165     double T_e_nondim =
166         (energy_period_s - this->design_energy_period_s) /
167         this->design_energy_period_s;
168
169     double production = exp(
170         -2.25119 * pow(T_e_nondim, 2) +
171         3.44570 * T_e_nondim * H_s_nondim -
172         4.01508 * pow(H_s_nondim, 2)
173     );
174
175     return production * this->capacity_kW;
176 }   /* __computeGaussianProductionkW() */
```

4.26.3.3 __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.

```
277 {
278     // *** WORK IN PROGRESS *** //
279
280     return 0;
281 }   /* __computeLookupProductionkW() */
```

4.26.3.4 __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.

```
217 {
218     // first, check for idealized wave breaking (deep water)
219     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
220         return 0;
221     }
222 
223     // otherwise, apply generic quadratic performance model
224     // (with outputs bounded to [0, 1])
225     double production =
226         0.289 * significant_wave_height_m -
227         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
228         0.0169 * energy_period_s;
229 
230     if (production < 0) {
231         production = 0;
232     }
233 
234     else if (production > 1) {
235         production = 1;
236     }
237 
238     return production * this->capacity_kW;
239 }   /* __computeParaboloidProductionkW() */
```

4.26.3.5 __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].

```
87 {
88     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
89 
90     return capital_cost_per_kW * this->capacity_kW;
91 }   /* __getGenericCapitalCost() */
```

4.26.3.6 __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].

```
115 {
116     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
117 
118     return operation_maintenance_cost_kWh;
119 }   /* __getGenericOpMaintCost() */
```

4.26.3.7 __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.

```
299 {
300     //  1. create filestream
301     write_path += "summary_results.md";
302     std::ofstream ofs;
303     ofs.open(write_path, std::ofstream::out);
304 
305     //  2. write summary results (markdown)
306     ofs « "# ";
307     ofs « std::to_string(int(ceil(this->capacity_kW)));
308     ofs « " kW WAVE Summary Results\n";
309     ofs « "\n--------\n\n";
310 
311     //  2.1. Production attributes
312     ofs « "## Production Attributes\n";
313     ofs « "\n";
314 
315     ofs « "Capacity: " « this->capacity_kW « "kW  \n";
316     ofs « "\n";
317 
318     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
319     ofs « "Capital Cost: " « this->capital_cost « "  \n";
320     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
321         « " per kWh produced  \n";
322     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
323         « "  \n";
324     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
```

```
325         « "  \n";
326     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
327     ofs « "\n";
328 
329     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
330     ofs « "\n--------\n\n";
331 
332     //  2.2. Renewable attributes
333     ofs « "## Renewable Attributes\n";
334     ofs « "\n";
335 
336     ofs « "Resource Key (2D): " « this->resource_key « "  \n";
337 
338     ofs « "\n--------\n\n";
339 
340     //  2.3. Wave attributes
341     ofs « "## Wave Attributes\n";
342     ofs « "\n";
343 
344     ofs « "Power Production Model: " « this->power_model_string « "  \n";
345     switch (this->power_model) {
346         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
347             ofs « "Design Significant Wave Height: "
348                 « this->design_significant_wave_height_m « " m  \n";
349 
350             ofs « "Design Energy Period: " « this->design_energy_period_s « " s  \n";
351 
352             break;
353         }
354 
355         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
356             //...
357 
358             break;
359         }
360 
361         default: {
362             // write nothing!
363 
364             break;
365         }
366     }
367 
368     ofs « "\n--------\n\n";
369 
370     //  2.4. Wave Results
371     ofs « "## Results\n";
372     ofs « "\n";
373 
374     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
375     ofs « "\n";
376 
377     ofs « "Total Dispatch: " « this->total_dispatch_kWh
378         « " kWh  \n";
379 
380     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
381         « " per kWh dispatched  \n";
382     ofs « "\n";
383 
384     ofs « "Running Hours: " « this->running_hours « "  \n";
385     ofs « "Replacements: " « this->n_replacements « "  \n";
386 
387     ofs « "\n--------\n\n";
388 
389     ofs.close();
390 
391     return;
392 }   /* __writeSummary() */
```

4.26.3.8 __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.

```
430 {
431     //  1. create filestream
432     write_path += "time_series_results.csv";
433     std::ofstream ofs;
434     ofs.open(write_path, std::ofstream::out);
435 
436     //  2. write time series results (comma separated value)
437     ofs « "Time (since start of data) [hrs],";
438     ofs « "Significant Wave Height [m],";
439     ofs « "Energy Period [s],";
440     ofs « "Production [kW],";
441     ofs « "Dispatch [kW],";
442     ofs « "Storage [kW],";
443     ofs « "Curtailment [kW],";
444     ofs « "Capital Cost (actual),";
445     ofs « "Operation and Maintenance Cost (actual),";
446     ofs « "\n";
447 
448     for (int i = 0; i < max_lines; i++) {
449         ofs « time_vec_hrs_ptr->at(i) « ",";
450         ofs « resource_map_2D_ptr->at(this->resource_key)[i][0] « ",";
451         ofs « resource_map_2D_ptr->at(this->resource_key)[i][1] « ",";
452         ofs « this->production_vec_kW[i] « ",";
453         ofs « this->dispatch_vec_kW[i] « ",";
454         ofs « this->storage_vec_kW[i] « ",";
455         ofs « this->curtailment_vec_kW[i] « ",";
456         ofs « this->capital_cost_vec[i] « ",";
457         ofs « this->operation_maintenance_cost_vec[i] « ",";
458         ofs « "\n";
459     }
460 
461     return;
462 }   /* __writeTimeSeries() */
```

4.26.3.9 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.

```
743 {
744     //  1. invoke base class method
745     load_kW = Renewable :: commit(
746         timestep,
747         dt_hrs,
748         production_kW,
749         load_kW
750     );
751 
752 
753     //...
754 
755     return load_kW;
756 }   /* commit() */
```

4.26.3.10 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.

```
647 {
648     // check if no resource
649     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
650         return 0;
651     }
652 
653     // compute production
654     double production_kW = 0;
655 
656     switch (this->power_model) {
657         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
658             production_kW = this->__computeParaboloidProductionkW(
659                 timestep,
660                 dt_hrs,
661                 significant_wave_height_m,
662                 energy_period_s
663             );
664 
```

```
665             break;
666         }
667 
668         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
669             production_kW = this->__computeGaussianProductionkW(
670                 timestep,
671                 dt_hrs,
672                 significant_wave_height_m,
673                 energy_period_s
674             );
675 
676             break;
677         }
678 
679         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
680             production_kW = this->__computeLookupProductionkW(
681                 timestep,
682                 dt_hrs,
683                 significant_wave_height_m,
684                 energy_period_s
685             );
686 
687             break;
688         }
689 
690         default: {
691             std::string error_str = "ERROR:  Wave::computeProductionkW():  ";
692             error_str += "power model ";
693             error_str += std::to_string(this->power_model);
694             error_str += " not recognized";
695 
696             #ifdef _WIN32
697                 std::cout « error_str « std::endl;
698             #endif
699 
700             throw std::runtime_error(error_str);
701 
702             break;
703         }
704     }
705 
706     return production_kW;
707 }   /* computeProductionkW() */
```

4.26.3.11 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.

```
603 {
604     //  1. reset attributes
605     //...
606 
607     //  2. invoke base class method
608     Renewable :: handleReplacement(timestep);
609 
610     return;
611 }   /* __handleReplacement() */
```

4.26.4 Member Data Documentation

4.26.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

The energy period [s] at which the wave energy converter achieves its rated capacity.

4.26.4.2 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.

4.26.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

4.26.4.4 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:

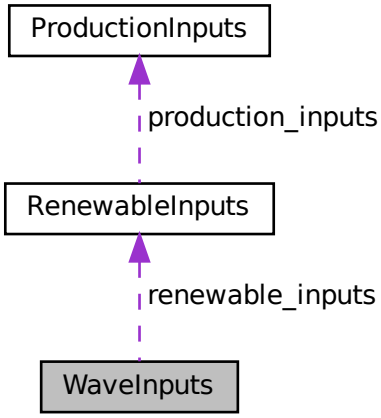
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

4.27 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.

4.27.1 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.

4.27.2 Member Data Documentation

4.27.2.1 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].

4.27.2.2 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.

4.27.2.3 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.

4.27.2.4 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].

4.27.2.5 power_model

```
WavePowerProductionModel WaveInputs::power_model = WavePowerProductionModel ::  WAVE_POWER_PARABOLOID
```

The wave power production model to be applied.

4.27.2.6 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.27.2.7 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:

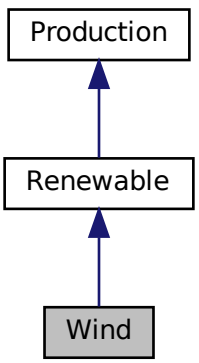
- header/Production/Renewable/Wave.h

4.28 Wind Class Reference

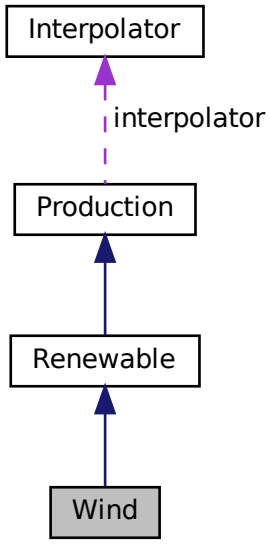
A derived class of the Renewable branch of Production which models wind production.

```
#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)

  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 __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.

4.28.1 Detailed Description

A derived class of the Renewable branch of Production which models wind production.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 Wind() [1/2]

```
Wind::Wind (
            void  )
```

Constructor (dummy) for the Wind class.

```
390 {
391     return;
392 }   /* Wind() */
```

4.28.2.2 Wind() [2/2]

```
Wind::Wind (
            int n_points,
            double n_years,
            WindInputs wind_inputs )
```

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.

```
420   :
421 Renewable(
422     n_points,
423     n_years,
424     wind_inputs.renewable_inputs
425 )
426 {
427     //  1. check inputs
428     this->__checkInputs(wind_inputs);
429 
430     //  2. set attributes
431     this->type = RenewableType :: WIND;
432     this->type_str = "WIND";
433 
434     this->resource_key = wind_inputs.resource_key;
435 
436     this->design_speed_ms = wind_inputs.design_speed_ms;
437 
438     this->power_model = wind_inputs.power_model;
439 
440     switch (this->power_model) {
441         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
442             this->power_model_string = "EXPONENTIAL";
443 
444             break;
445         }
446 
447         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
448             this->power_model_string = "LOOKUP";
449 
450             break;
451         }
452 
453         default: {
454             std::string error_str = "ERROR:  Wind():  ";
455             error_str += "power production model ";
456             error_str += std::to_string(this->power_model);
457             error_str += " not recognized";
458 
459             #ifdef _WIN32
460                 std::cout << error_str << std::endl;
461             #endif
462 
463             throw std::runtime_error(error_str);
464 
465             break;
466         }
467     }
468 
469     if (wind_inputs.capital_cost < 0) {
470         this->capital_cost = this->__getGenericCapitalCost();
471     }
472 
473     if (wind_inputs.operation_maintenance_cost_kWh < 0) {
474         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
475     }
476 
477     if (not this->is_sunk) {
478         this->capital_cost_vec[0] = this->capital_cost;
479     }
480 
481     //  3. construction print
482     if (this->print_flag) {
483         std::cout << "Wind object constructed at " << this << std::endl;
484     }
485 
486     return;
487 }   /* Renewable() */
```

4.28.2.3 ~Wind()

```
Wind::~Wind (
            void  )
```

Destructor for the Wind class.

```
656 {
657     //  1. destruction print
658     if (this->print_flag) {
659         std::cout « "Wind object at " « this « " destroyed" « std::endl;
660     }
661 
662     return;
663 }   /* ~Wind() */
```

4.28.3 Member Function Documentation

4.28.3.1 __checkInputs()

```
void Wind::__checkInputs (
            WindInputs wind_inputs )  [private]
```

Helper method to check inputs to the Wind constructor.

Parameters

wind_inputs	A structure of Wind constructor inputs.

```
39 {
40     //  1. check design_speed_ms
41     if (wind_inputs.design_speed_ms <= 0) {
42         std::string error_str = "ERROR:  Wind():  ";
43         error_str += "WindInputs::design_speed_ms 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     return;
53 }   /* __checkInputs() */
```

4.28.3.2 __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.

```
140 {
141     double production = 0;
142 
143     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
144         this->design_speed_ms;
145 
146     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
147         production = 0;
148     }
149 
150     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
151         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
152     }
153 
154     else {
155         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
156     }
157 
158     return production * this->capacity_kW;
159 }   /* __computeExponentialProductionkW() */
```

4.28.3.3 __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.

```
191 {
192     // *** WORK IN PROGRESS *** //
193 
194     return 0;
195 }   /* __computeLookupProductionkW() */
```

4.28.3.4 __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].

```
75 {
76     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
77 
78     return capital_cost_per_kW * this->capacity_kW;
79 }   /* __getGenericCapitalCost() */
```

4.28.3.5 __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].

```
102 {
103     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
104 
105     return operation_maintenance_cost_kWh;
106 }   /* __getGenericOpMaintCost() */
```

4.28.3.6 __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.

```
213 {
214     //  1. create filestream
215     write_path += "summary_results.md";
216     std::ofstream ofs;
217     ofs.open(write_path, std::ofstream::out);
218 
219     //  2. write summary results (markdown)
220     ofs « "# ";
221     ofs « std::to_string(int(ceil(this->capacity_kW)));
222     ofs « " kW WIND Summary Results\n";
223     ofs « "\n--------\n\n";
224 
225 
226     //  2.1. Production attributes
```

```
227     ofs « "## Production Attributes\n";
228     ofs « "\n";
229 
230     ofs « "Capacity: " « this->capacity_kW « "kW  \n";
231     ofs « "\n";
232 
233     ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
234     ofs « "Capital Cost: " « this->capital_cost « "  \n";
235     ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
236         « " per kWh produced  \n";
237     ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
238         « "  \n";
239     ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
240         « "  \n";
241     ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
242     ofs « "\n";
243 
244     ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
245     ofs « "\n--------\n\n";
246 
247     //  2.2. Renewable attributes
248     ofs « "## Renewable Attributes\n";
249     ofs « "\n";
250 
251     ofs « "Resource Key (1D): " « this->resource_key « "  \n";
252 
253     ofs « "\n--------\n\n";
254 
255     //  2.3. Wind attributes
256     ofs « "## Wind Attributes\n";
257     ofs « "\n";
258 
259     ofs « "Power Production Model: " « this->power_model_string « "  \n";
260     switch (this->power_model) {
261         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
262             ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";
263 
264             break;
265         }
266 
267         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
268             //...
269 
270             break;
271         }
272 
273         default: {
274             // write nothing!
275 
276             break;
277         }
278     }
279 
280     ofs « "\n--------\n\n";
281 
282     //  2.4. Wind Results
283     ofs « "## Results\n";
284     ofs « "\n";
285 
286     ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
287     ofs « "\n";
288 
289     ofs « "Total Dispatch: " « this->total_dispatch_kWh
290         « " kWh  \n";
291 
292     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
293         « " per kWh dispatched  \n";
294     ofs « "\n";
295 
296     ofs « "Running Hours: " « this->running_hours « "  \n";
297     ofs « "Replacements: " « this->n_replacements « "  \n";
298 
299     ofs « "\n--------\n\n";
300 
301     ofs.close();
302 
303     return;
304 }   /* __writeSummary() */
```

4.28.3.7 __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.

```
342 {
343     //  1. create filestream
344     write_path += "time_series_results.csv";
345     std::ofstream ofs;
346     ofs.open(write_path, std::ofstream::out);
347 
348     //  2. write time series results (comma separated value)
349     ofs « "Time (since start of data) [hrs],";
350     ofs « "Wind Resource [m/s],";
351     ofs « "Production [kW],";
352     ofs « "Dispatch [kW],";
353     ofs « "Storage [kW],";
354     ofs « "Curtailment [kW],";
355     ofs « "Capital Cost (actual),";
356     ofs « "Operation and Maintenance Cost (actual),";
357     ofs « "\n";
358 
359     for (int i = 0; i < max_lines; i++) {
360         ofs « time_vec_hrs_ptr->at(i) « ",";
361         ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
362         ofs « this->production_vec_kW[i] « ",";
363         ofs « this->dispatch_vec_kW[i] « ",";
364         ofs « this->storage_vec_kW[i] « ",";
365         ofs « this->curtailment_vec_kW[i] « ",";
366         ofs « this->capital_cost_vec[i] « ",";
367         ofs « this->operation_maintenance_cost_vec[i] « ",";
368         ofs « "\n";
369     }
370 
371     return;
372 }   /* __writeTimeSeries() */
```

4.28.3.8 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.

```
628 {
629     //  1. invoke base class method
630     load_kW = Renewable :: commit(
631         timestep,
632         dt_hrs,
633         production_kW,
634         load_kW
635     );
636 
637 
638     //...
639 
640     return load_kW;
641 }   /* commit() */
```

4.28.3.9 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.

```
545 {
546     // check if no resource
547     if (wind_resource_ms <= 0) {
548         return 0;
549     }
550 
551     // compute production
552     double production_kW = 0;
```

```
553
554     switch (this->power_model) {
555         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
556             production_kW = this->__computeExponentialProductionkW(
557                 timestep,
558                 dt_hrs,
559                 wind_resource_ms
560             );
561
562             break;
563         }
564
565         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
566             production_kW = this->__computeLookupProductionkW(
567                 timestep,
568                 dt_hrs,
569                 wind_resource_ms
570             );
571
572             break;
573         }
574
575         default: {
576             std::string error_str = "ERROR:  Wind::computeProductionkW():  ";
577             error_str += "power model ";
578             error_str += std::to_string(this->power_model);
579             error_str += " not recognized";
580
581             #ifdef _WIN32
582                 std::cout « error_str « std::endl;
583             #endif
584
585             throw std::runtime_error(error_str);
586
587             break;
588         }
589     }
590
591     return production_kW;
592 }   /* computeProductionkW() */
```

4.28.3.10 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.

```
505 {
506     //  1. reset attributes
507     //...
508
509     //  2. invoke base class method
510     Renewable :: handleReplacement(timestep);
511
512     return;
513 }   /* __handleReplacement() */
```

4.28.4 Member Data Documentation

4.28.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

The wind speed [m/s] at which the wind turbine achieves its rated capacity.

4.28.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

4.28.4.3 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:

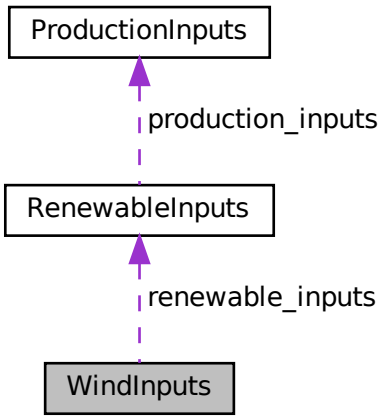
- header/Production/Renewable/Wind.h
- source/Production/Renewable/Wind.cpp

4.29 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 = 8

  The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- WindPowerProductionModel power_model = WindPowerProductionModel :: WIND_POWER_EXPONENTIAL

  The wind power production model to be applied.

4.29.1 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.

4.29.2 Member Data Documentation

4.29.2.1 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].

4.29.2.2 design_speed_ms

```
double WindInputs::design_speed_ms = 8
```

The wind speed [m/s] at which the wind turbine achieves its rated capacity.

4.29.2.3 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].

4.29.2.4 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel ::  WIND_POWER_EXPONENTIAL
```

The wind power production model to be applied.

4.29.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.29.2.6 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

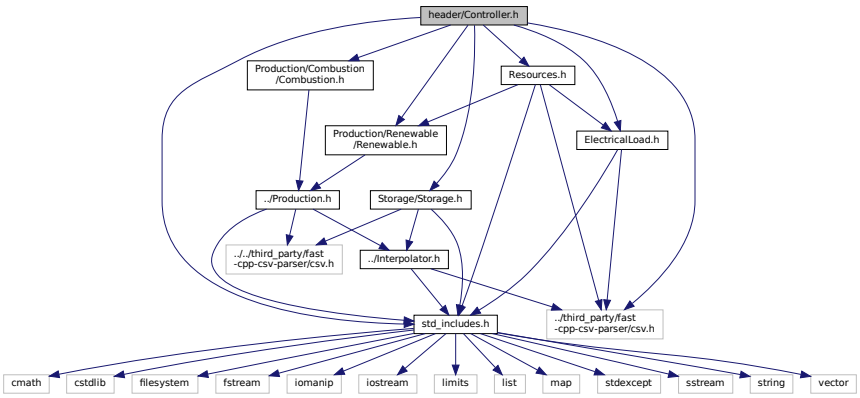
File Documentation

5.1 header/Controller.h File Reference

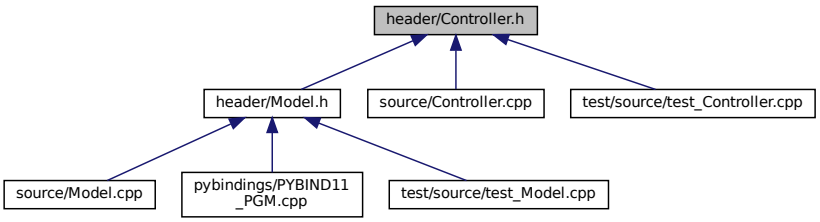
Header file for the Controller class.

```
#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/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.

```
43                  {
44     LOAD_FOLLOWING,
```

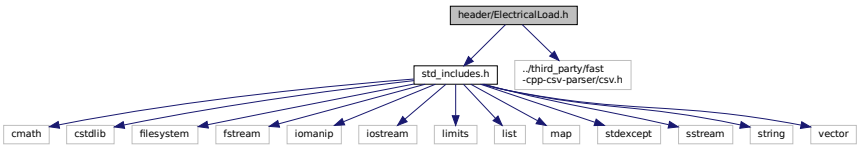
```
45     CYCLE_CHARGING,
46     N_CONTROL_MODES
47 };
```

5.2 header/ElectricalLoad.h File Reference

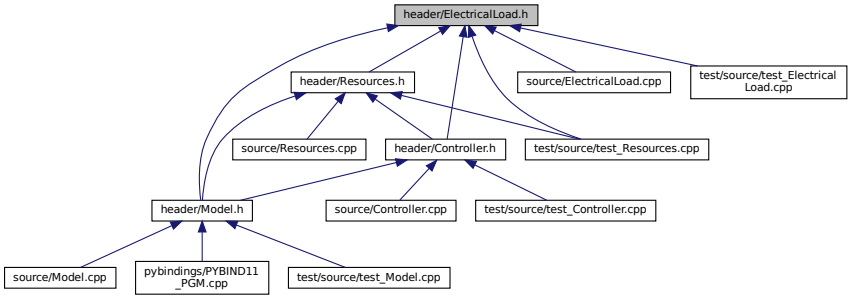
Header file for the ElectricalLoad class.

```
#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.

5.2.1 Detailed Description

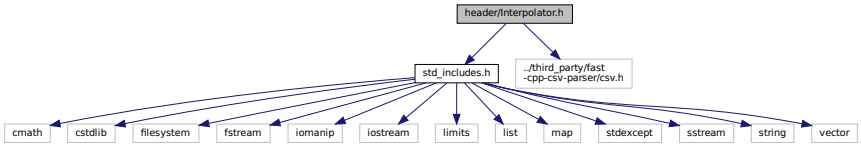
Header file for the ElectricalLoad class.

5.3 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.

5.3.1 Detailed Description

Header file for the Interpolator class.

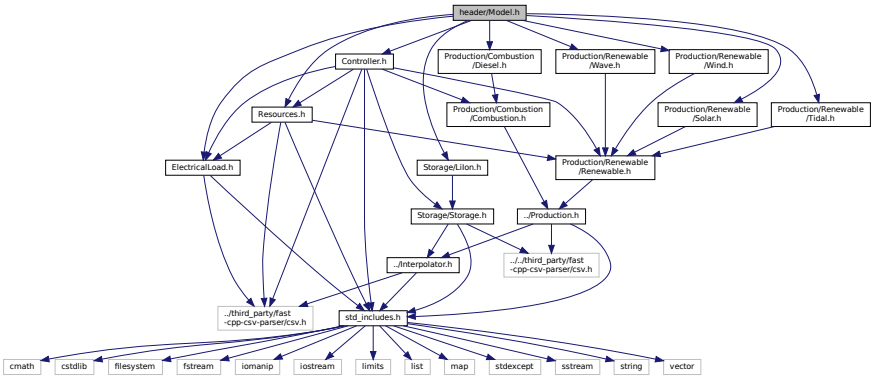
5.4 header/Model.h File Reference

Header file for the Model class.

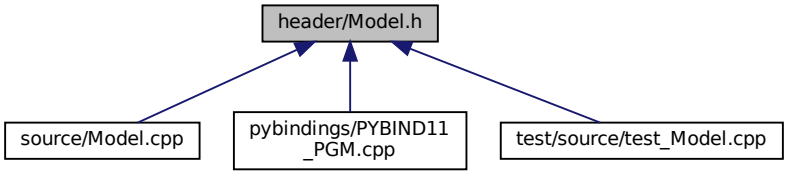
```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
```

```
#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.

5.4.1 Detailed Description

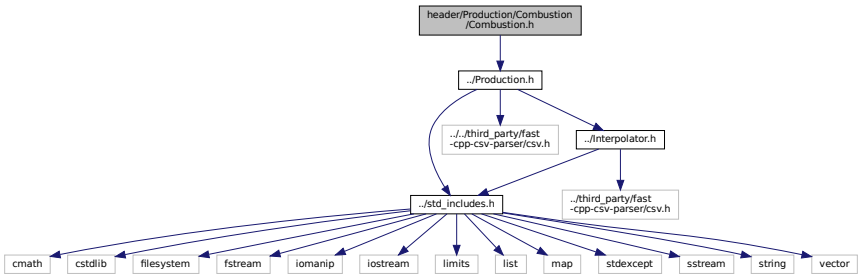
Header file for the Model class.

5.5 header/Production/Combustion/Combustion.h File Reference

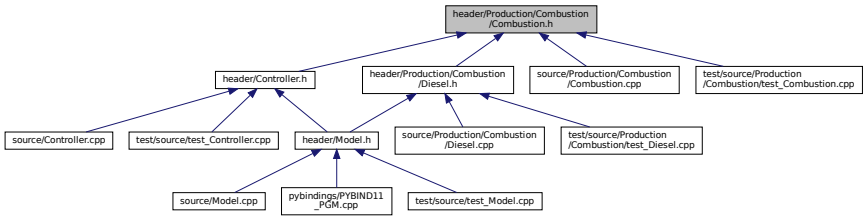
Header file for the Combustion class.

```
#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.

5.5.1 Detailed Description

Header file for the Combustion class.

5.5.2 Enumeration Type Documentation

5.5.2.1 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.

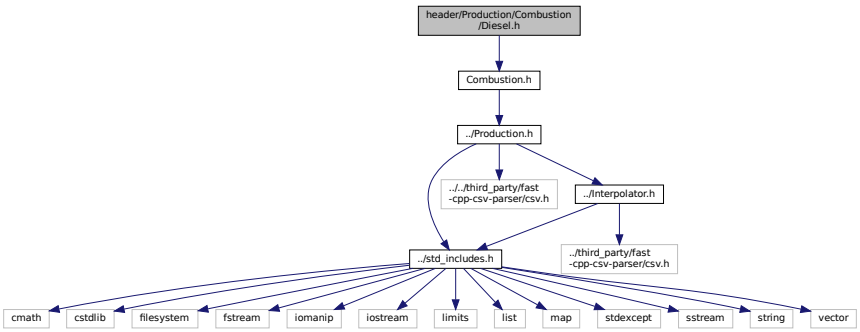
```
33                  {
34     DIESEL,
35     N_COMBUSTION_TYPES
36 };
```

5.6 header/Production/Combustion/Diesel.h File Reference

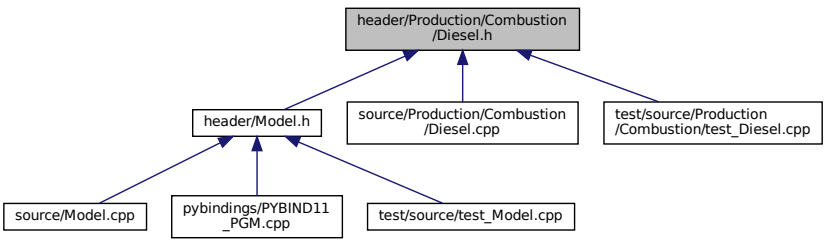
Header file for the Diesel class.

```
#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.

5.6.1 Detailed Description

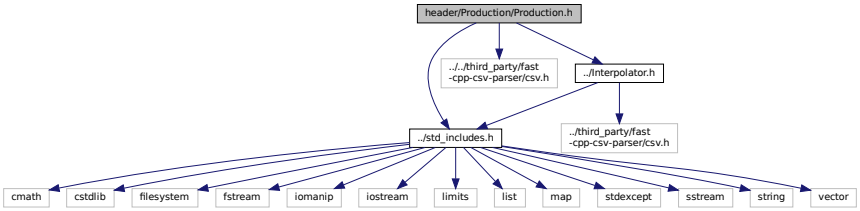
Header file for the Diesel class.

5.7 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.

5.7.1 Detailed Description

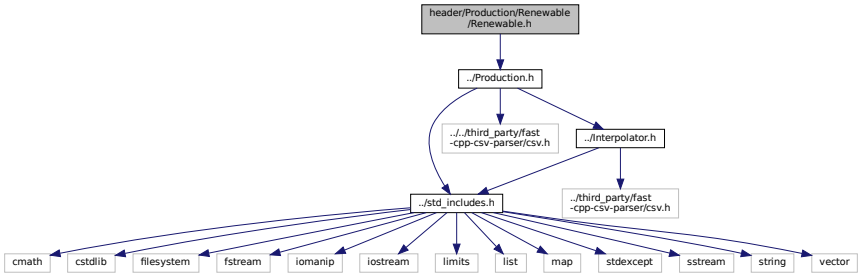
Header file for the Production class.

5.8 header/Production/Renewable/Renewable.h File Reference

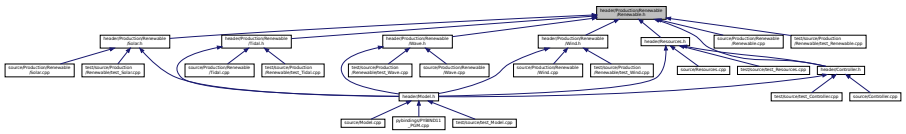
Header file for the Renewable class.

```
#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.

5.8.1 Detailed Description

Header file for the Renewable class.

5.8.2 Enumeration Type Documentation

5.8.2.1 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.

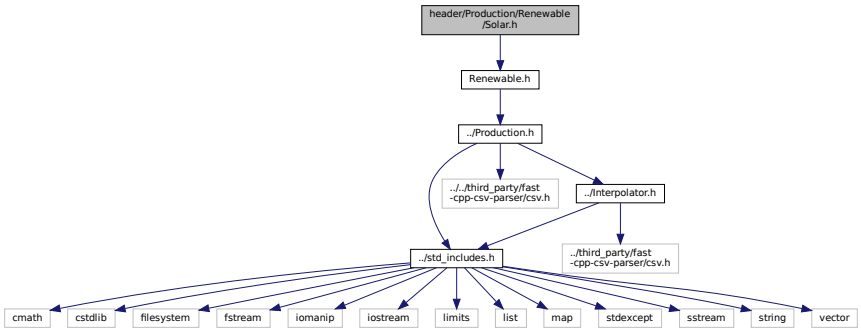
```
33                  {
34     SOLAR,
35     TIDAL,
36     WAVE,
37     WIND,
38     N_RENEWABLE_TYPES
39 };
```

5.9 header/Production/Renewable/Solar.h File Reference

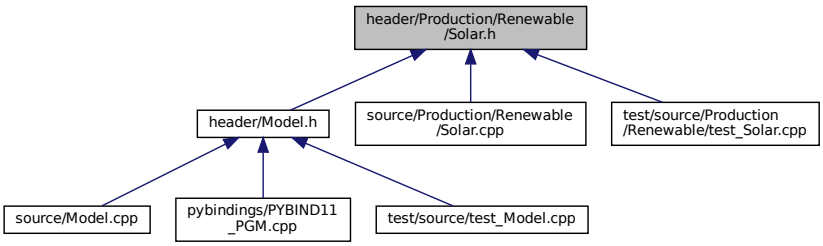
Header file for the Solar class.

```
#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.

5.9.1 Detailed Description

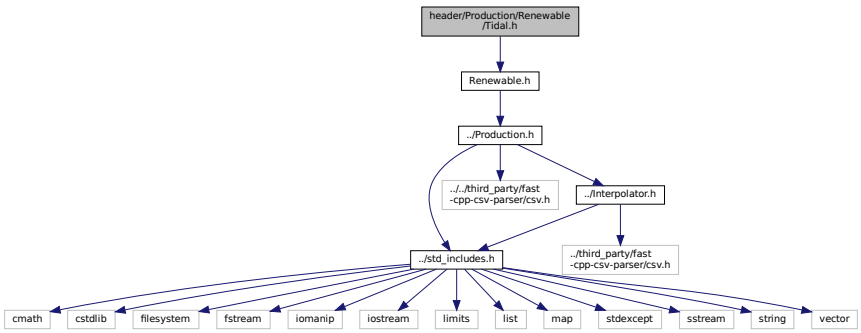
Header file for the Solar class.

5.10 header/Production/Renewable/Tidal.h File Reference

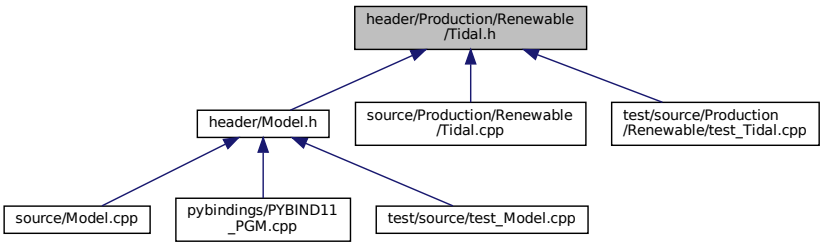
Header file for the Tidal class.

```
#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 }

5.10.1 Detailed Description

Header file for the Tidal class.

5.10.2 Enumeration Type Documentation

5.10.2.1 TidalPowerProductionModel

`enum TidalPowerProductionModel`

Enumerator

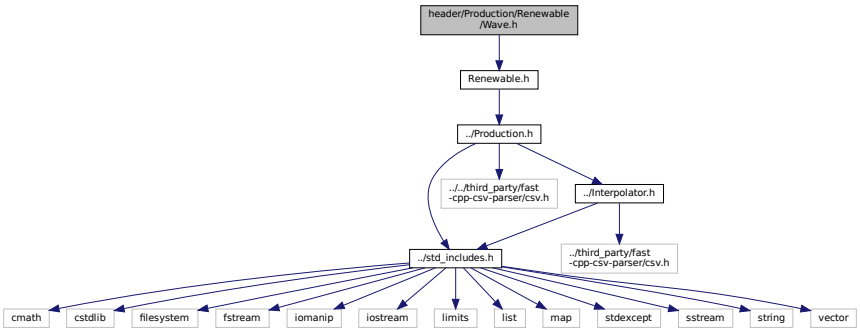
TIDAL_POWER_CUBIC	A cubic power production model.
TIDAL_POWER_EXPONENTIAL	An exponential power production model.
TIDAL_POWER_LOOKUP	Lookup from a given set of power curve data.
N_TIDAL_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in TidalPowerProductionModel.

```
34                                  {
35     TIDAL_POWER_CUBIC,
36     TIDAL_POWER_EXPONENTIAL,
37     TIDAL_POWER_LOOKUP,
38     N_TIDAL_POWER_PRODUCTION_MODELS
39 };
```

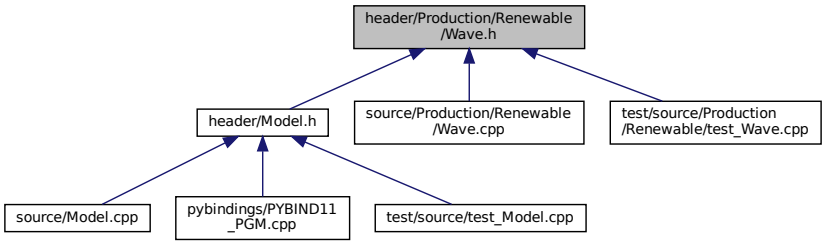
5.11 header/Production/Renewable/Wave.h File Reference

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 }

5.11.1 Detailed Description

Header file for the Wave class.

5.11.2 Enumeration Type Documentation

5.11.2.1 WavePowerProductionModel

```
enum WavePowerProductionModel
```

Enumerator

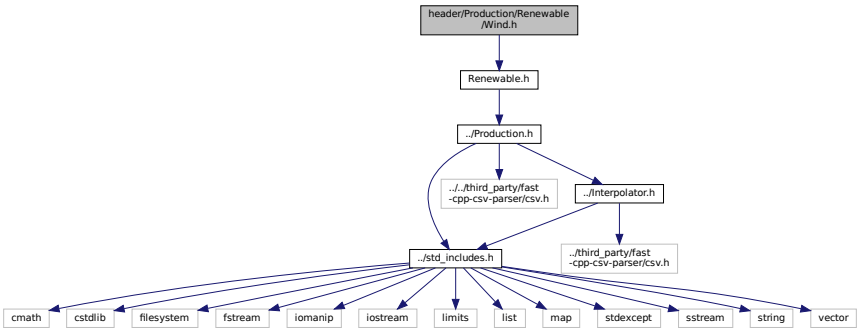
WAVE_POWER_GAUSSIAN	A Gaussian power production model.
WAVE_POWER_PARABOLOID	A paraboloid power production model.
WAVE_POWER_LOOKUP	Lookup from a given performance matrix.
N_WAVE_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in WavePowerProductionModel.

```
34                              {
35     WAVE_POWER_GAUSSIAN,
36     WAVE_POWER_PARABOLOID,
37     WAVE_POWER_LOOKUP,
38     N_WAVE_POWER_PRODUCTION_MODELS
39 };
```

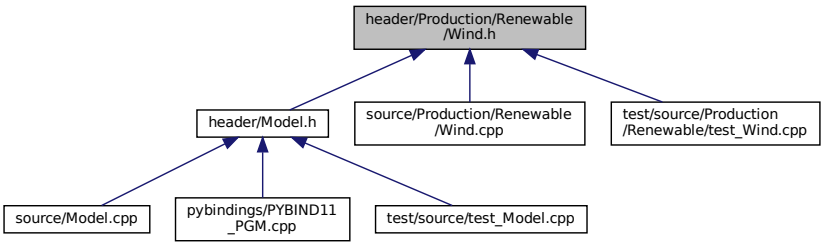
5.12 header/Production/Renewable/Wind.h File Reference

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_EXPONENTIAL , WIND_POWER_LOOKUP , N_WIND_POWER_PRODUCTION_MODELS }

5.12.1 Detailed Description

Header file for the Wind class.

5.12.2 Enumeration Type Documentation

5.12.2.1 WindPowerProductionModel

```
enum WindPowerProductionModel
```

Enumerator

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_EXPONENTIAL,
```

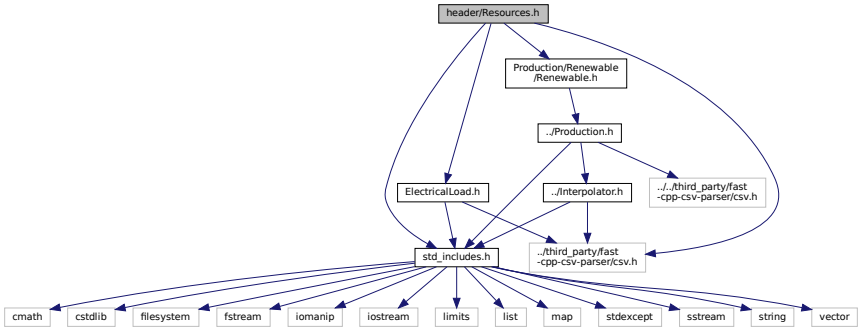
```
36     WIND_POWER_LOOKUP,
37     N_WIND_POWER_PRODUCTION_MODELS
38 };
```

5.13 header/Resources.h File Reference

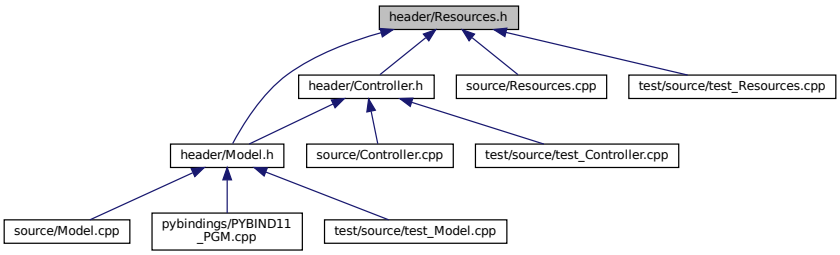
Header file for the Resources class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/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.

5.13.1 Detailed Description

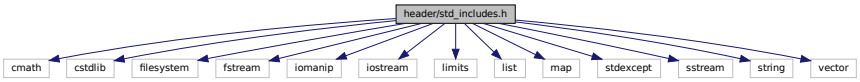
Header file for the Resources class.

5.14 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:

5.14.1 Detailed Description

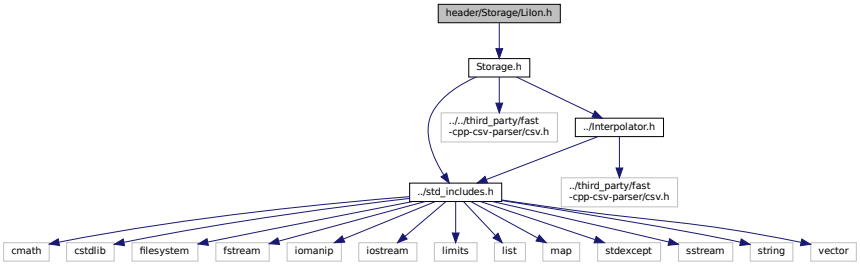
Header file which simply batches together some standard includes.

5.15 header/Storage/Lilon.h File Reference

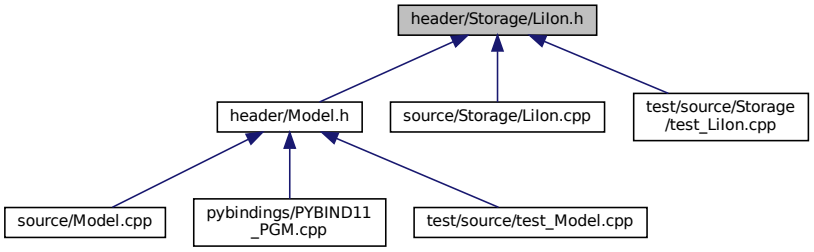
Header file for the Lilon class.

```
#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.

5.15.1 Detailed Description

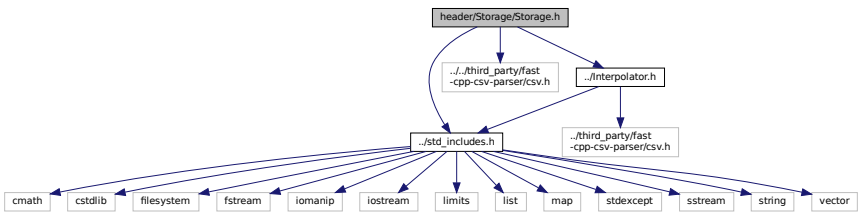
Header file for the LiIon class.

5.16 header/Storage/Storage.h File Reference

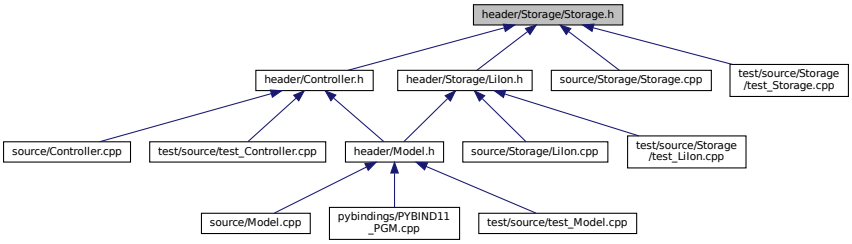
Header file for the Storage class.

```
#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.

5.16.1 Detailed Description

Header file for the Storage class.

5.16.2 Enumeration Type Documentation

5.16.2.1 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,
```

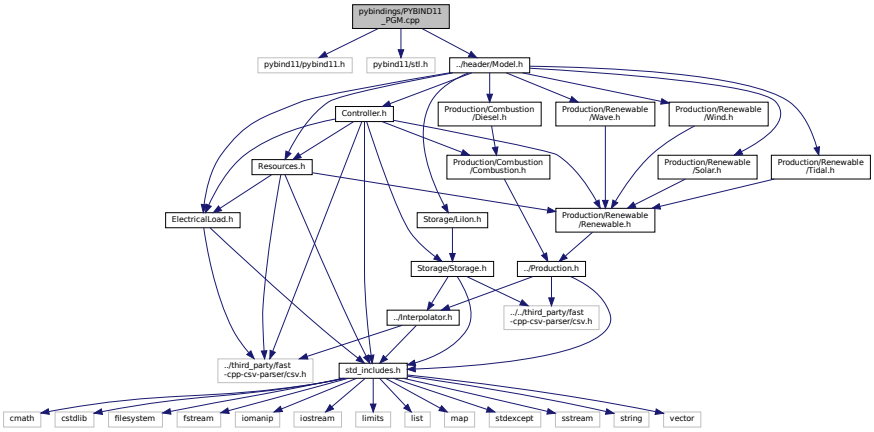
```
38     N_STORAGE_TYPES
39 };
```

5.17 pybindings/PYBIND11_PGM.cpp File Reference

Python 3 bindings file for PGMcpp.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include "../header/Model.h"
```

Include dependency graph for PYBIND11_PGM.cpp:



Functions

- PYBIND11_MODULE (PGMcpp, m)

5.17.1 Detailed Description

Python 3 bindings file for PGMcpp.

This is a file which defines the Python 3 bindings to be generated for PGMcpp. To generate bindings, use the provided setup.py.

ref: https://pybind11.readthedocs.io/en/stable/

5.17.2 Function Documentation

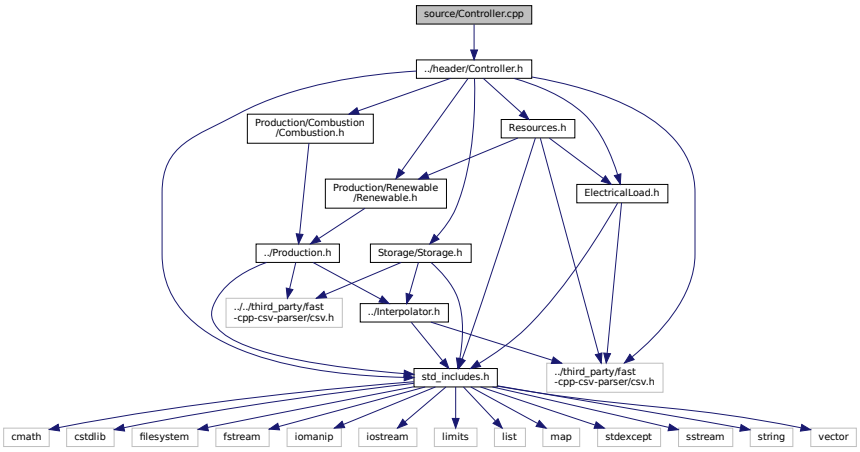
5.17.2.1 PYBIND11_MODULE()

```
PYBIND11_MODULE (
             PGMcpp ,
             m  )
30                           {
31
32 // ================ Controller ================ //
33 /*
34 pybind11::class_<Controller>(m, "Controller")
35     .def(pybind11::init());
36 */
37 // ================ END Controller ================ //
38
39
40
41 // ================ ElectricalLoad ================ //
42 /*
43 pybind11::class_<ElectricalLoad>(m, "ElectricalLoad")
44     .def_readwrite("n_points", &ElectricalLoad::n_points)
45     .def_readwrite("max_load_kW", &ElectricalLoad::max_load_kW)
46     .def_readwrite("mean_load_kW", &ElectricalLoad::mean_load_kW)
47     .def_readwrite("min_load_kW", &ElectricalLoad::min_load_kW)
48     .def_readwrite("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs)
49     .def_readwrite("load_vec_kW", &ElectricalLoad::load_vec_kW)
50     .def_readwrite("time_vec_hrs", &ElectricalLoad::time_vec_hrs)
51
52     .def(pybind11::init<std::string>());
53 */
54 // ================ END ElectricalLoad ================ //
55
56
57
58 // ================ Model ================ //
59 /*
60 pybind11::class_<Model>(m, "Model")
61     .def(
62         pybind11::init<
63             ElectricalLoad*,
64             RenewableResources*
65         >()
66     );
67 */
68 // ================ END Model ================ //
69
70
71
72 // ================ RenewableResources ================ //
73 /*
74 pybind11::class_<RenewableResources>(m, "RenewableResources")
75     .def(pybind11::init());
76     /*
77     .def(pybind11::init<>());
78     */
79 */
80 // ================ END RenewableResources ================ //
81
82 }   /* PYBIND11_MODULE() */
```

5.18 source/Controller.cpp File Reference

Implementation file for the Controller class.

```
#include "../header/Controller.h"
```
Include dependency graph for Controller.cpp:



5.18.1 Detailed Description

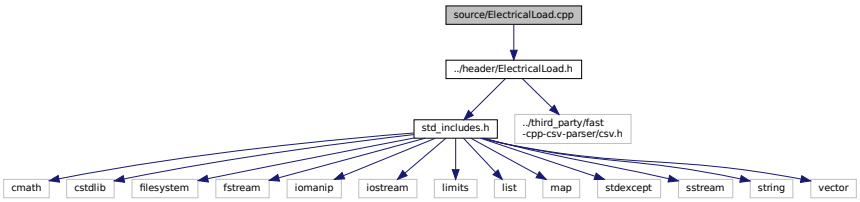
Implementation file for the Controller class.

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

5.19 source/ElectricalLoad.cpp File Reference

Implementation file for the ElectricalLoad class.

```
#include "../header/ElectricalLoad.h"
```
Include dependency graph for ElectricalLoad.cpp:



5.19.1 Detailed Description

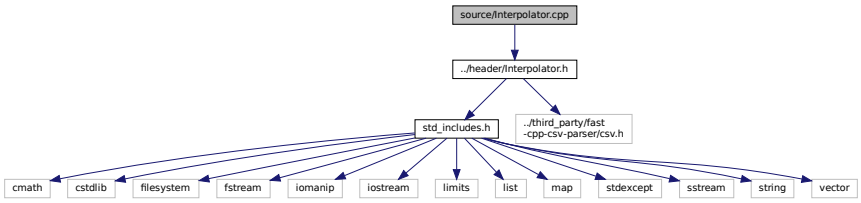
Implementation file for the ElectricalLoad class.

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

5.20 source/Interpolator.cpp File Reference

Implementation file for the Interpolator class.

`#include "../header/Interpolator.h"`
Include dependency graph for Interpolator.cpp:



5.20.1 Detailed Description

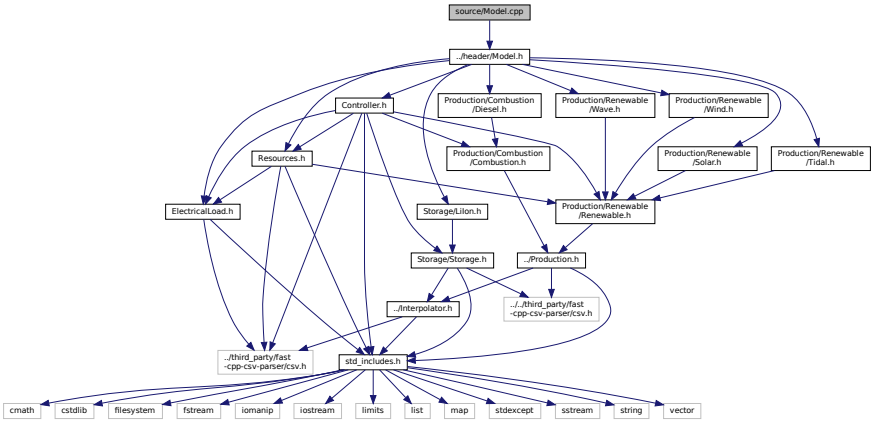
Implementation file for the Interpolator class.

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

5.21 source/Model.cpp File Reference

Implementation file for the Model class.

`#include "../header/Model.h"`
Include dependency graph for Model.cpp:



5.21.1 Detailed Description

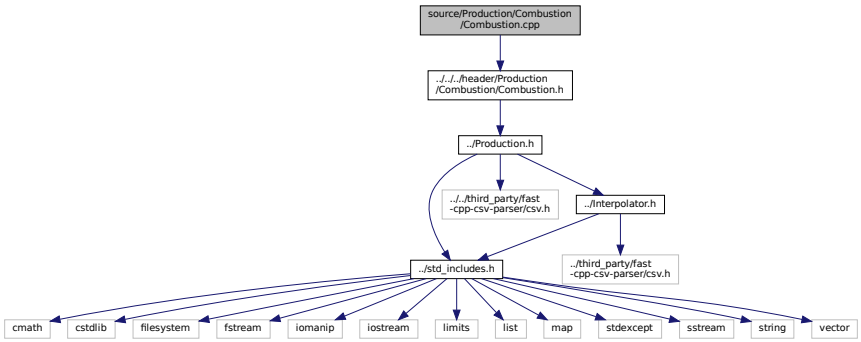
Implementation file for the Model class.

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.

5.22 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:



5.22.1 Detailed Description

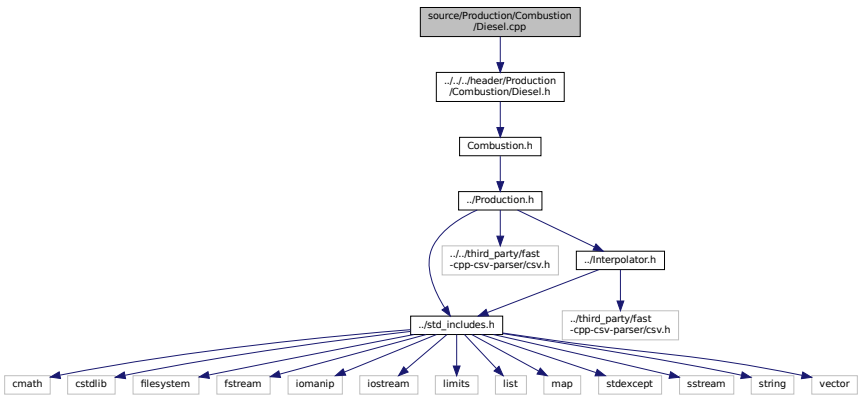
Implementation file for the Combustion class.

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.

5.23 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:



5.23.1 Detailed Description

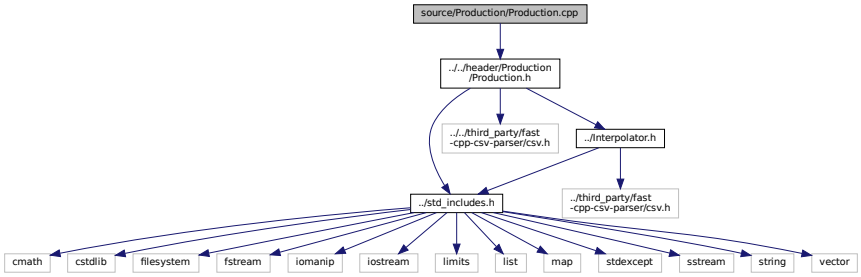
Implementation file for the Diesel class.

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

5.24 source/Production/Production.cpp File Reference

Implementation file for the Production class.

```
#include "../../header/Production/Production.h"
```
Include dependency graph for Production.cpp:



5.24.1 Detailed Description

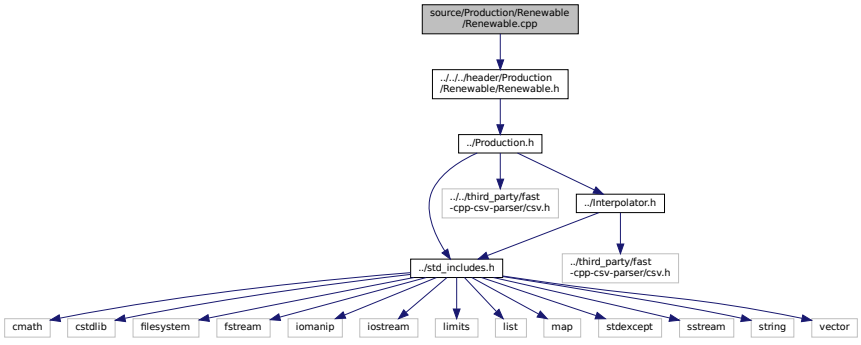
Implementation file for the Production class.

The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.25 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:



5.25.1 Detailed Description

Implementation file for the Renewable class.

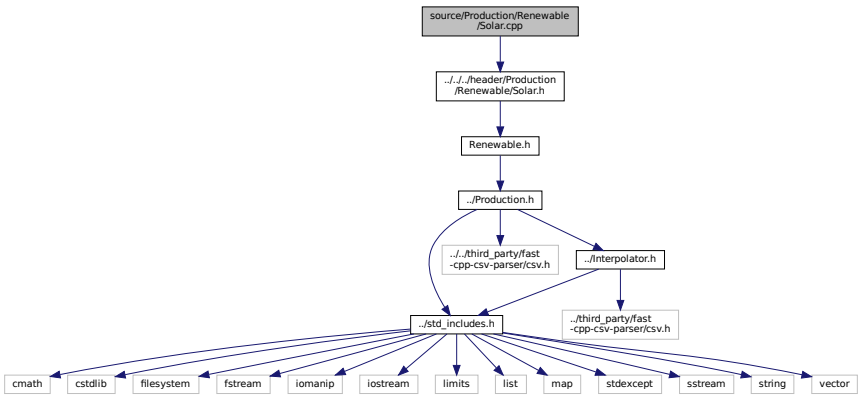
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

5.26 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:



5.26.1 Detailed Description

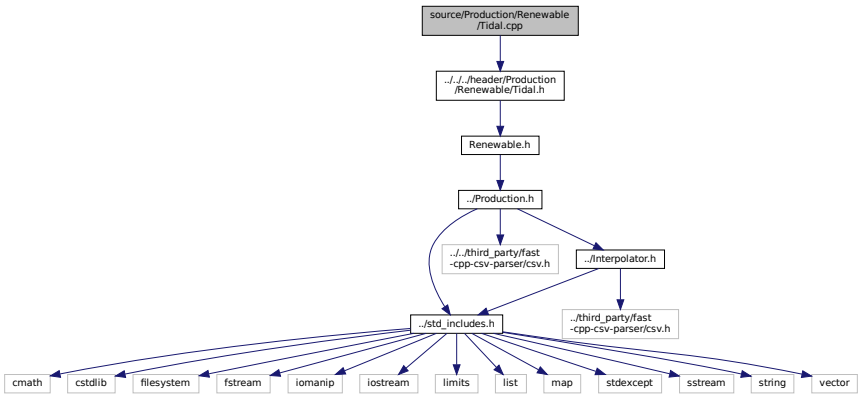
Implementation file for the Solar class.

A derived class of the Renewable branch of Production which models solar production.

5.27 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:



5.27.1 Detailed Description

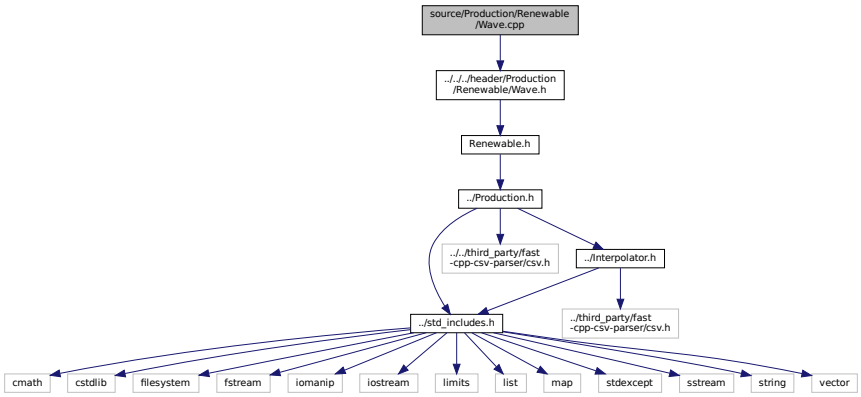
Implementation file for the Tidal class.

A derived class of the Renewable branch of Production which models tidal production.

5.28 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:



5.28.1 Detailed Description

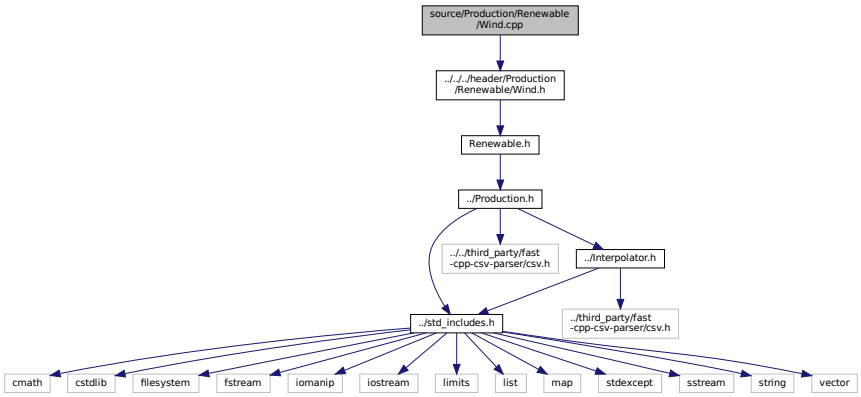
Implementation file for the Wave class.

A derived class of the Renewable branch of Production which models wave production.

5.29 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:



5.29.1 Detailed Description

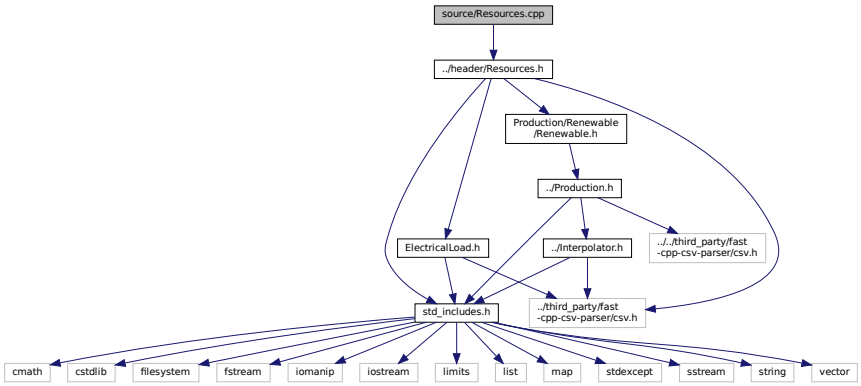
Implementation file for the Wind class.

A derived class of the Renewable branch of Production which models wind production.

5.30 source/Resources.cpp File Reference

Implementation file for the Resources class.

`#include "../header/Resources.h"`
Include dependency graph for Resources.cpp:



5.30.1 Detailed Description

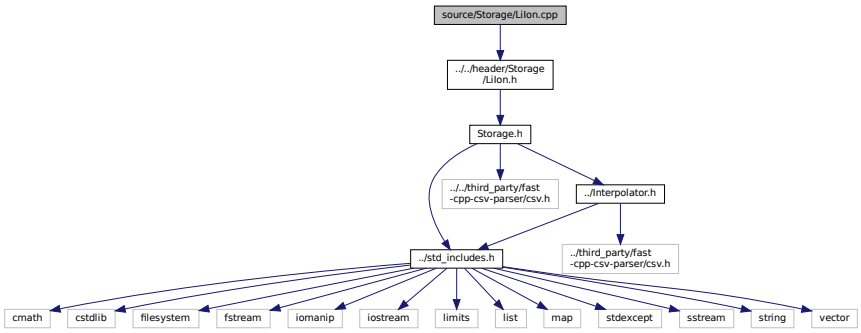
Implementation file for the Resources class.

A class which contains renewable resource data. Intended to serve as a component class of Model.

5.31 source/Storage/LiIon.cpp File Reference

Implementation file for the LiIon class.

```
#include "../../header/Storage/LiIon.h"
```
Include dependency graph for LiIon.cpp:



5.31.1 Detailed Description

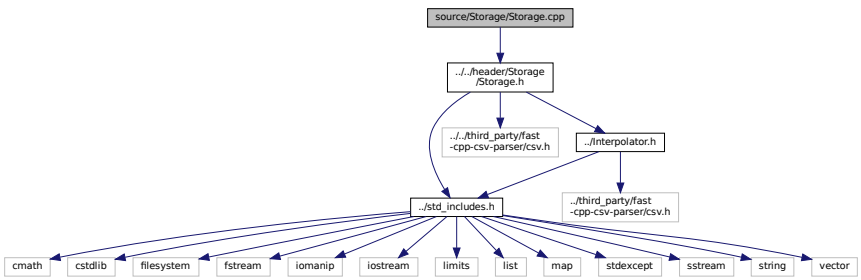
Implementation file for the LiIon class.

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

5.32 source/Storage/Storage.cpp File Reference

Implementation file for the Storage class.

```
#include "../../header/Storage/Storage.h"
```
Include dependency graph for Storage.cpp:



5.32.1 Detailed Description

Implementation file for the Storage class.

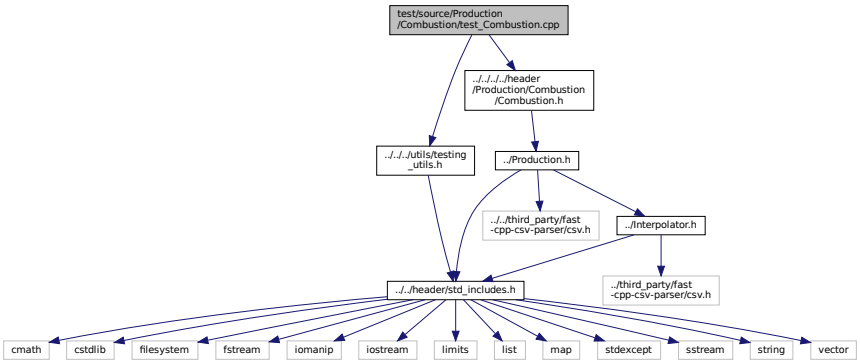
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

5.33 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

- int main (int argc, char **argv)

5.33.1 Detailed Description

Testing suite for Combustion class.

A suite of tests for the Combustion class.

5.33.2 Function Documentation

5.33.2.1 main()

```
int main (
             int argc,
             char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Combustion");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ========================================================== //

CombustionInputs combustion_inputs;

Combustion test_combustion(8760, 1, combustion_inputs);

// ======== END CONSTRUCTION ====================================================== //


// ======== ATTRIBUTES ============================================================ //

testTruth(
    not combustion_inputs.production_inputs.print_flag,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.fuel_consumption_vec_L.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.fuel_cost_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.CO2_emissions_vec_kg.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.CO_emissions_vec_kg.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.NOx_emissions_vec_kg.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_combustion.SOx_emissions_vec_kg.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
     test_combustion.CH4_emissions_vec_kg.size(),
     8760,
     __FILE__,
     __LINE__
);

testFloatEquals(
```

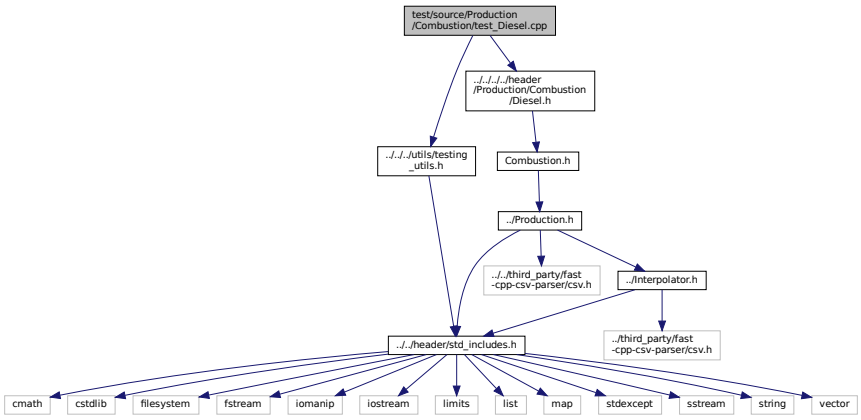
```
107     test_combustion.PM_emissions_vec_kg.size(),
108     8760,
109     __FILE__,
110     __LINE__
111 );
112
113 // ======== END ATTRIBUTES ============================================================ //
114
115 }   /* try */
116
117
118 catch (...) {
119     //...
120
121     printGold(" .............. ");
122     printRed("FAIL");
123     std::cout « std::endl;
124     throw;
125 }
126
127
128 printGold(" .............. ");
129 printGreen("PASS");
130 std::cout « std::endl;
131 return 0;
132
133 }   /* main() */
```

5.34 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

- int main (int argc, char ∗∗argv)

5.34.1 Detailed Description

Testing suite for Diesel class.

A suite of tests for the Diesel class.

5.34.2 Function Documentation

5.34.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31 
32     printGold("\tTesting Production <-- Combustion <-- Diesel");
33 
34     srand(time(NULL));
35 
36 
37     Combustion* test_diesel_ptr;
38 
39 try {
40 
41 // ======== CONSTRUCTION ============================================================ //
42 
43 bool error_flag = true;
44 
45 try {
46     DieselInputs bad_diesel_inputs;
47     bad_diesel_inputs.fuel_cost_L = -1;
48 
49     Diesel bad_diesel(8760, 1, bad_diesel_inputs);
50 
51     error_flag = false;
52 } catch (...) {
53     // Task failed successfully! =P
54 }
55 if (not error_flag) {
56     expectedErrorNotDetected(__FILE__, __LINE__);
57 }
58 
59 DieselInputs diesel_inputs;
60 
61 test_diesel_ptr =  new Diesel(8760, 1, diesel_inputs);
62 
63 
64 // ======== END CONSTRUCTION ======================================================== //
65 
66 
67 
68 // ======== ATTRIBUTES ============================================================== //
69 
70 testTruth(
71     not diesel_inputs.combustion_inputs.production_inputs.print_flag,
72     __FILE__,
73     __LINE__
74 );
75 
76 testFloatEquals(
77     test_diesel_ptr->type,
78     CombustionType :: DIESEL,
79     __FILE__,
80     __LINE__
81 );
82 
83 testTruth(
84     test_diesel_ptr->type_str == "DIESEL",
85     __FILE__,
86     __LINE__
87 );
88 
89 testFloatEquals(
90     test_diesel_ptr->linear_fuel_slope_LkWh,
91     0.265675,
92     __FILE__,
93     __LINE__
94 );
95 
96 testFloatEquals(
97     test_diesel_ptr->linear_fuel_intercept_LkWh,
```

```
98      0.026676,
99      __FILE__,
100      __LINE__
101 );
102
103 testFloatEquals(
104     test_diesel_ptr->capital_cost,
105     94125.375446,
106     __FILE__,
107     __LINE__
108 );
109
110 testFloatEquals(
111     test_diesel_ptr->operation_maintenance_cost_kWh,
112     0.069905,
113     __FILE__,
114     __LINE__
115 );
116
117 testFloatEquals(
118     ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
119     0.2,
120     __FILE__,
121     __LINE__
122 );
123
124 testFloatEquals(
125     ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
126     4,
127     __FILE__,
128     __LINE__
129 );
130
131 testFloatEquals(
132     test_diesel_ptr->replace_running_hrs,
133     30000,
134     __FILE__,
135     __LINE__
136 );
137
138 // ======== END ATTRIBUTES ============================================================ //
139
140
141
142 // ======== METHODS ================================================================== //
143
144 //  test capacity constraint
145 testFloatEquals(
146     test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
147     test_diesel_ptr->capacity_kW,
148     __FILE__,
149     __LINE__
150 );
151
152 //  test minimum load ratio constraint
153 testFloatEquals(
154     test_diesel_ptr->requestProductionkW(
155         0,
156         1,
157         0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
158             test_diesel_ptr->capacity_kW
159     ),
160     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
161     __FILE__,
162     __LINE__
163 );
164
165 //  test commit()
166 std::vector<double> dt_vec_hrs (48, 1);
167
168 std::vector<double> load_vec_kW = {
169     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
170     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
171     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
172     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
173 };
174
175 std::vector<bool> expected_is_running_vec = {
176     1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
177     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
178     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
179     1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
180 };
181
182 double load_kW = 0;
183 double production_kW = 0;
184 double roll = 0;
```

```
185
186 for (int i = 0; i < 48; i++) {
187     roll = (double)rand() / RAND_MAX;
188
189     if (roll >= 0.95) {
190         roll = 1.25;
191     }
192
193     load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
194     load_kW = load_vec_kW[i];
195
196     production_kW = test_diesel_ptr->requestProductionkW(
197         i,
198         dt_vec_hrs[i],
199         load_kW
200     );
201
202     load_kW = test_diesel_ptr->commit(
203         i,
204         dt_vec_hrs[i],
205         production_kW,
206         load_kW
207     );
208
209     // load_kW <= load_vec_kW (i.e., after vs before)
210     testLessThanOrEqualTo(
211         load_kW,
212         load_vec_kW[i],
213         __FILE__,
214         __LINE__
215     );
216
217     // production = dispatch + storage + curtailment
218     testFloatEquals(
219         test_diesel_ptr->production_vec_kW[i] -
220         test_diesel_ptr->dispatch_vec_kW[i] -
221         test_diesel_ptr->storage_vec_kW[i] -
222         test_diesel_ptr->curtailment_vec_kW[i],
223         0,
224         __FILE__,
225         __LINE__
226     );
227
228     // capacity constraint
229     if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
230         testFloatEquals(
231             test_diesel_ptr->production_vec_kW[i],
232             test_diesel_ptr->capacity_kW,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // minimum load ratio constraint
239     else if (
240         test_diesel_ptr->is_running and
241         test_diesel_ptr->production_vec_kW[i] > 0 and
242         load_vec_kW[i] <
243         ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
244     ) {
245         testFloatEquals(
246             test_diesel_ptr->production_vec_kW[i],
247             ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
248                 test_diesel_ptr->capacity_kW,
249             __FILE__,
250             __LINE__
251         );
252     }
253
254     // minimum runtime constraint
255     testFloatEquals(
256         test_diesel_ptr->is_running_vec[i],
257         expected_is_running_vec[i],
258         __FILE__,
259         __LINE__
260     );
261
262     // O&M, fuel consumption, and emissions > 0 whenever diesel is running
263     if (test_diesel_ptr->is_running) {
264         testGreaterThan(
265             test_diesel_ptr->operation_maintenance_cost_vec[i],
266             0,
267             __FILE__,
268             __LINE__
269         );
270
271         testGreaterThan(
```

```
272             test_diesel_ptr->fuel_consumption_vec_L[i],
273             0,
274             __FILE__,
275             __LINE__
276         );
277 
278         testGreaterThan(
279             test_diesel_ptr->fuel_cost_vec[i],
280             0,
281             __FILE__,
282             __LINE__
283         );
284 
285         testGreaterThan(
286             test_diesel_ptr->CO2_emissions_vec_kg[i],
287             0,
288             __FILE__,
289             __LINE__
290         );
291 
292         testGreaterThan(
293             test_diesel_ptr->CO_emissions_vec_kg[i],
294             0,
295             __FILE__,
296             __LINE__
297         );
298 
299         testGreaterThan(
300             test_diesel_ptr->NOx_emissions_vec_kg[i],
301             0,
302             __FILE__,
303             __LINE__
304         );
305 
306         testGreaterThan(
307             test_diesel_ptr->SOx_emissions_vec_kg[i],
308             0,
309             __FILE__,
310             __LINE__
311         );
312 
313         testGreaterThan(
314             test_diesel_ptr->CH4_emissions_vec_kg[i],
315             0,
316             __FILE__,
317             __LINE__
318         );
319 
320         testGreaterThan(
321             test_diesel_ptr->PM_emissions_vec_kg[i],
322             0,
323             __FILE__,
324             __LINE__
325         );
326     }
327 
328     // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
329     else {
330         testFloatEquals(
331             test_diesel_ptr->operation_maintenance_cost_vec[i],
332             0,
333             __FILE__,
334             __LINE__
335         );
336 
337         testFloatEquals(
338             test_diesel_ptr->fuel_consumption_vec_L[i],
339             0,
340             __FILE__,
341             __LINE__
342         );
343 
344         testFloatEquals(
345             test_diesel_ptr->fuel_cost_vec[i],
346             0,
347             __FILE__,
348             __LINE__
349         );
350 
351         testFloatEquals(
352             test_diesel_ptr->CO2_emissions_vec_kg[i],
353             0,
354             __FILE__,
355             __LINE__
356         );
357 
358         testFloatEquals(
```

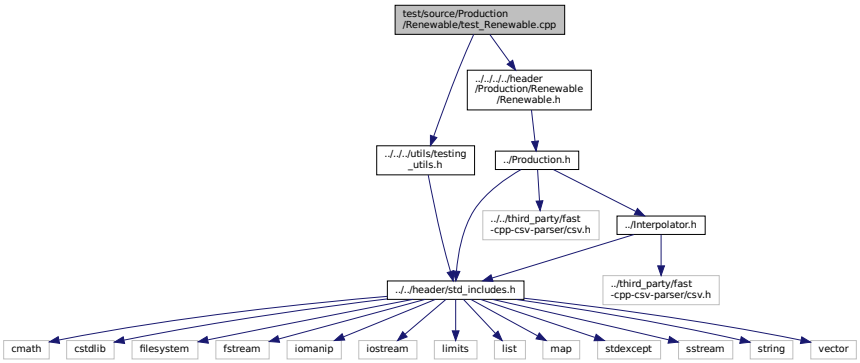
```
359             test_diesel_ptr->CO_emissions_vec_kg[i],
360             0,
361             __FILE__,
362             __LINE__
363         );
364 
365         testFloatEquals(
366             test_diesel_ptr->NOx_emissions_vec_kg[i],
367             0,
368             __FILE__,
369             __LINE__
370         );
371 
372         testFloatEquals(
373             test_diesel_ptr->SOx_emissions_vec_kg[i],
374             0,
375             __FILE__,
376             __LINE__
377         );
378 
379         testFloatEquals(
380             test_diesel_ptr->CH4_emissions_vec_kg[i],
381             0,
382             __FILE__,
383             __LINE__
384         );
385 
386         testFloatEquals(
387             test_diesel_ptr->PM_emissions_vec_kg[i],
388             0,
389             __FILE__,
390             __LINE__
391         );
392     }
393 }
394 
395 // ======== END METHODS ============================================================ //
396 
397 }   /* try */
398 
399 
400 catch (...) {
401     delete test_diesel_ptr;
402 
403     printGold(" ... ");
404     printRed("FAIL");
405     std::cout « std::endl;
406     throw;
407 }
408 
409 
410 delete test_diesel_ptr;
411 
412 printGold(" ... ");
413 printGreen("PASS");
414 std::cout « std::endl;
415 return 0;
416 
417 }   /* main() */
```

5.35 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

- int main (int argc, char **argv)

5.35.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

5.35.2 Function Documentation

5.35.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ========================================================== //
40
41 RenewableInputs renewable_inputs;
42
43 Renewable test_renewable(8760, 1, renewable_inputs);
44
45 // ======== END CONSTRUCTION ====================================================== //
46
47
48
```

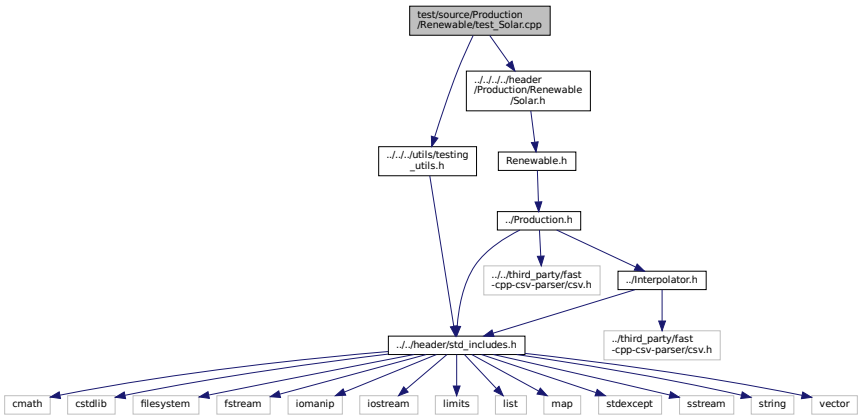
```
49 // ======== ATTRIBUTES ============================================================ //
50
51 testTruth(
52     not renewable_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56
57 // ======== END ATTRIBUTES ======================================================== //
58
59 }   /* try */
60
61
62 catch (...) {
63     //...
64
65     printGold(" ............... ");
66     printRed("FAIL");
67     std::cout « std::endl;
68     throw;
69 }
70
71
72 printGold(" ............... ");
73 printGreen("PASS");
74 std::cout « std::endl;
75 return 0;
76 }   /* main() */
```

5.36 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

- int main (int argc, char **argv)

5.36.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar class.

5.36.2 Function Documentation

5.36.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31 
32     printGold("\tTesting Production <-- Renewable <-- Solar");
33 
34     srand(time(NULL));
35 
36     Renewable* test_solar_ptr;
37 
38 try {
39 
40 // ======== CONSTRUCTION ============================================================ //
41 
42 bool error_flag = true;
43 
44 try {
45     SolarInputs bad_solar_inputs;
46     bad_solar_inputs.derating = -1;
47 
48     Solar bad_solar(8760, 1, bad_solar_inputs);
49 
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57 
58 SolarInputs solar_inputs;
59 
60 test_solar_ptr = new Solar(8760, 1, solar_inputs);
61 
62 // ======== END CONSTRUCTION ======================================================== //
63 
64 
65 
66 // ======== ATTRIBUTES ============================================================== //
67 
68 testTruth(
69     not solar_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73 
74 testFloatEquals(
75     test_solar_ptr->type,
76     RenewableType :: SOLAR,
77     __FILE__,
78     __LINE__
79 );
80 
81 testTruth(
82     test_solar_ptr->type_str == "SOLAR",
83     __FILE__,
84     __LINE__
85 );
86 
87 testFloatEquals(
88     test_solar_ptr->capital_cost,
89     350118.723363,
90     __FILE__,
91     __LINE__
92 );
93 
94 testFloatEquals(
95     test_solar_ptr->operation_maintenance_cost_kWh,
96     0.01,
97     __FILE__,
```

```
98      __LINE__
99  );
100
101 // ======== END ATTRIBUTES ============================================================ //
102
103
104
105 // ======== METHODS ================================================================== //
106
107 // test production constraints
108 testFloatEquals(
109     test_solar_ptr->computeProductionkW(0, 1, 2),
110     100,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_solar_ptr->computeProductionkW(0, 1, -1),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double solar_resource_kWm2 = 0;
136
137 for (int i = 0; i < 48; i++) {
138     roll = (double)rand() / RAND_MAX;
139
140     solar_resource_kWm2 = roll;
141
142     roll = (double)rand() / RAND_MAX;
143
144     if (roll <= 0.1) {
145         solar_resource_kWm2 = 0;
146     }
147
148     else if (roll >= 0.95) {
149         solar_resource_kWm2 = 1.25;
150     }
151
152     roll = (double)rand() / RAND_MAX;
153
154     if (roll >= 0.95) {
155         roll = 1.25;
156     }
157
158     load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
159     load_kW = load_vec_kW[i];
160
161     production_kW = test_solar_ptr->computeProductionkW(
162         i,
163         dt_vec_hrs[i],
164         solar_resource_kWm2
165     );
166
167     load_kW = test_solar_ptr->commit(
168         i,
169         dt_vec_hrs[i],
170         production_kW,
171         load_kW
172     );
173
174     // is running (or not) as expected
175     if (solar_resource_kWm2 > 0) {
176         testTruth(
177             test_solar_ptr->is_running,
178             __FILE__,
179             __LINE__
180         );
181     }
182
183     else {
184         testTruth(
```

```
185             not test_solar_ptr->is_running,
186             __FILE__,
187             __LINE__
188         );
189     }
190
191     // load_kW <= load_vec_kW (i.e., after vs before)
192     testLessThanOrEqualTo(
193         load_kW,
194         load_vec_kW[i],
195         __FILE__,
196         __LINE__
197     );
198
199     // production = dispatch + storage + curtailment
200     testFloatEquals(
201         test_solar_ptr->production_vec_kW[i] -
202         test_solar_ptr->dispatch_vec_kW[i] -
203         test_solar_ptr->storage_vec_kW[i] -
204         test_solar_ptr->curtailment_vec_kW[i],
205         0,
206         __FILE__,
207         __LINE__
208     );
209
210     // capacity constraint
211     if (solar_resource_kWm2 > 1) {
212         testFloatEquals(
213             test_solar_ptr->production_vec_kW[i],
214             test_solar_ptr->capacity_kW,
215             __FILE__,
216             __LINE__
217         );
218     }
219
220     // resource, O&M > 0 whenever solar is running (i.e., producing)
221     if (test_solar_ptr->is_running) {
222         testGreaterThan(
223             solar_resource_kWm2,
224             0,
225             __FILE__,
226             __LINE__
227         );
228
229         testGreaterThan(
230             test_solar_ptr->operation_maintenance_cost_vec[i],
231             0,
232             __FILE__,
233             __LINE__
234         );
235     }
236
237     // resource, O&M = 0 whenever solar is not running (i.e., not producing)
238     else {
239         testFloatEquals(
240             solar_resource_kWm2,
241             0,
242             __FILE__,
243             __LINE__
244         );
245
246         testFloatEquals(
247             test_solar_ptr->operation_maintenance_cost_vec[i],
248             0,
249             __FILE__,
250             __LINE__
251         );
252     }
253 }
254
255
256 // ======== END METHODS ============================================================ //
257
258 }   /* try */
259
260
261 catch (...) {
262     delete test_solar_ptr;
263
264     printGold(" ..... ");
265     printRed("FAIL");
266     std::cout « std::endl;
267     throw;
268 }
269
270
271 delete test_solar_ptr;
```

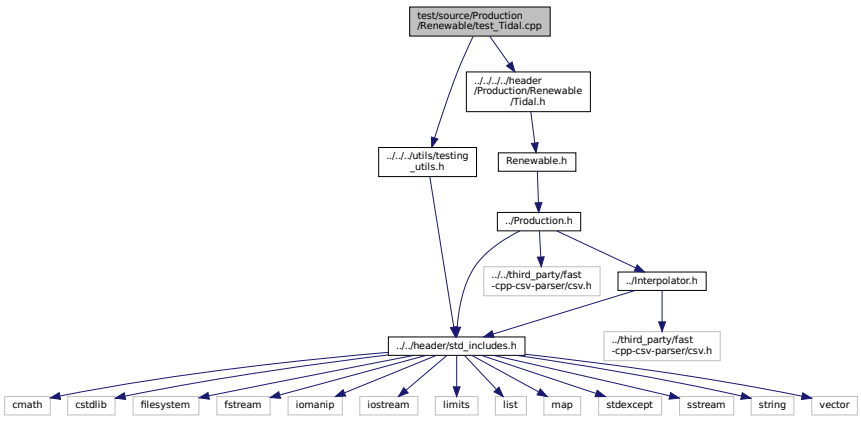
```
272
273 printGold(" ..... ");
274 printGreen("PASS");
275 std::cout « std::endl;
276 return 0;
277 }   /* main() */
```

5.37 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

- int main (int argc, char ∗∗argv)

5.37.1 Detailed Description

Testing suite for Tidal class.

A suite of tests for the Tidal class.

5.37.2 Function Documentation

5.37.2.1 main()

```
int main (
             int argc,
             char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Tidal");

    srand(time(NULL));

    Renewable* test_tidal_ptr;

try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    TidalInputs bad_tidal_inputs;
    bad_tidal_inputs.design_speed_ms = -1;

    Tidal bad_tidal(8760, 1, bad_tidal_inputs);

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

TidalInputs tidal_inputs;

test_tidal_ptr = new Tidal(8760, 1, tidal_inputs);

// ======== END CONSTRUCTION ======================================================== //


// ======== ATTRIBUTES ============================================================== //

testTruth(
    not tidal_inputs.renewable_inputs.production_inputs.print_flag,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_tidal_ptr->type,
    RenewableType :: TIDAL,
    __FILE__,
    __LINE__
);

testTruth(
    test_tidal_ptr->type_str == "TIDAL",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_tidal_ptr->capital_cost,
    500237.446725,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_tidal_ptr->operation_maintenance_cost_kWh,
    0.069905,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ========================================================== //


// ======== METHODS ================================================================= //

```

```
107 // test production constraints
108 testFloatEquals(
109     test_tidal_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_tidal_ptr->computeProductionkW(
117         0,
118         1,
119         ((Tidal*)test_tidal_ptr)->design_speed_ms
120     ),
121     test_tidal_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_tidal_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()
134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double tidal_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         tidal_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164
165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_tidal_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         tidal_resource_ms
176     );
177
178     load_kW = test_tidal_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_tidal_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193
```

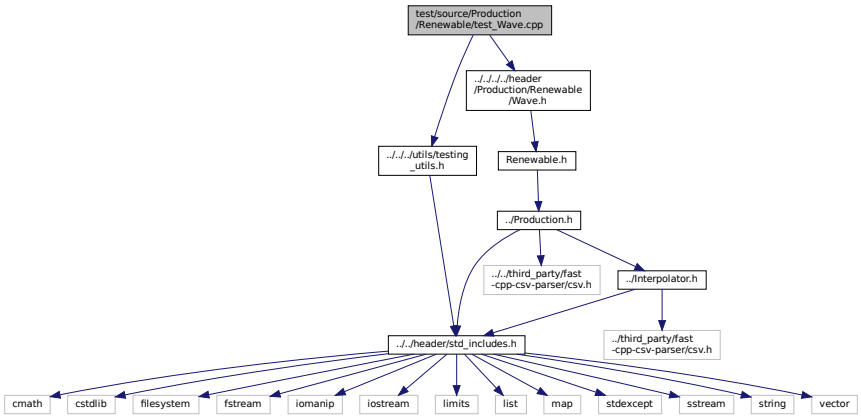
```
194     else {
195         testTruth(
196             not test_tidal_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201 
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209 
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_tidal_ptr->production_vec_kW[i] -
213         test_tidal_ptr->dispatch_vec_kW[i] -
214         test_tidal_ptr->storage_vec_kW[i] -
215         test_tidal_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220 
221     // resource, O&M > 0 whenever tidal is running (i.e., producing)
222     if (test_tidal_ptr->is_running) {
223         testGreaterThan(
224             tidal_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229 
230         testGreaterThan(
231             test_tidal_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237 
238     // O&M = 0 whenever tidal is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_tidal_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248 
249 
250 // ======== END METHODS ============================================================ //
251 
252 }   /* try */
253 
254 
255 catch (...) {
256     delete test_tidal_ptr;
257 
258     printGold(" ..... ");
259     printRed("FAIL");
260     std::cout « std::endl;
261     throw;
262 }
263 
264 
265 delete test_tidal_ptr;
266 
267 printGold(" ..... ");
268 printGreen("PASS");
269 std::cout « std::endl;
270 return 0;
271 }   /* main() */
```

5.38 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

- int main (int argc, char **argv)

5.38.1 Detailed Description

Testing suite for Wave class.

A suite of tests for the Wave class.

5.38.2 Function Documentation

5.38.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wave");
33
34     srand(time(NULL));
35
36     Renewable* test_wave_ptr;
37
38 try {
39
40 // ======== CONSTRUCTION ========================================================== //
41
42 bool error_flag = true;
43
```

```
44 try {
45     WaveInputs bad_wave_inputs;
46     bad_wave_inputs.design_significant_wave_height_m = -1;
47 
48     Wave bad_wave(8760, 1, bad_wave_inputs);
49 
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57 
58 WaveInputs wave_inputs;
59 
60 test_wave_ptr = new Wave(8760, 1, wave_inputs);
61 
62 // ======== END CONSTRUCTION ============================================================ //
63 
64 
65 
66 // ======== ATTRIBUTES ================================================================== //
67 
68 testTruth(
69     not wave_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73 
74 testFloatEquals(
75     test_wave_ptr->type,
76     RenewableType :: WAVE,
77     __FILE__,
78     __LINE__
79 );
80 
81 testTruth(
82     test_wave_ptr->type_str == "WAVE",
83     __FILE__,
84     __LINE__
85 );
86 
87 testFloatEquals(
88     test_wave_ptr->capital_cost,
89     850831.063539,
90     __FILE__,
91     __LINE__
92 );
93 
94 testFloatEquals(
95     test_wave_ptr->operation_maintenance_cost_kWh,
96     0.069905,
97     __FILE__,
98     __LINE__
99 );
100 
101 // ======== END ATTRIBUTES ============================================================== //
102 
103 
104 
105 // ======== METHODS ===================================================================== //
106 
107 // test production constraints
108 testFloatEquals(
109     test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
110     0,
111     __FILE__,
112     __LINE__
113 );
114 
115 testFloatEquals(
116     test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
117     0,
118     __FILE__,
119     __LINE__
120 );
121 
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124 
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
```

```
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double significant_wave_height_m = 0;
136 double energy_period_s = 0;
137
138 for (int i = 0; i < 48; i++) {
139     roll = (double)rand() / RAND_MAX;
140
141     if (roll <= 0.05) {
142         roll = 0;
143     }
144
145     significant_wave_height_m = roll *
146         ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
147
148     roll = (double)rand() / RAND_MAX;
149
150     if (roll <= 0.05) {
151         roll = 0;
152     }
153
154     energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
155
156     roll = (double)rand() / RAND_MAX;
157
158     if (roll >= 0.95) {
159         roll = 1.25;
160     }
161
162     load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
163     load_kW = load_vec_kW[i];
164
165     production_kW = test_wave_ptr->computeProductionkW(
166         i,
167         dt_vec_hrs[i],
168         significant_wave_height_m,
169         energy_period_s
170     );
171
172     load_kW = test_wave_ptr->commit(
173         i,
174         dt_vec_hrs[i],
175         production_kW,
176         load_kW
177     );
178
179     // is running (or not) as expected
180     if (production_kW > 0) {
181         testTruth(
182             test_wave_ptr->is_running,
183             __FILE__,
184             __LINE__
185         );
186     }
187
188     else {
189         testTruth(
190             not test_wave_ptr->is_running,
191             __FILE__,
192             __LINE__
193         );
194     }
195
196     // load_kW <= load_vec_kW (i.e., after vs before)
197     testLessThanOrEqualTo(
198         load_kW,
199         load_vec_kW[i],
200         __FILE__,
201         __LINE__
202     );
203
204     // production = dispatch + storage + curtailment
205     testFloatEquals(
206         test_wave_ptr->production_vec_kW[i] -
207         test_wave_ptr->dispatch_vec_kW[i] -
208         test_wave_ptr->storage_vec_kW[i] -
209         test_wave_ptr->curtailment_vec_kW[i],
210         0,
211         __FILE__,
212         __LINE__
213     );
214
215     // resource, O&M > 0 whenever wave is running (i.e., producing)
216     if (test_wave_ptr->is_running) {
217         testGreaterThan(
```

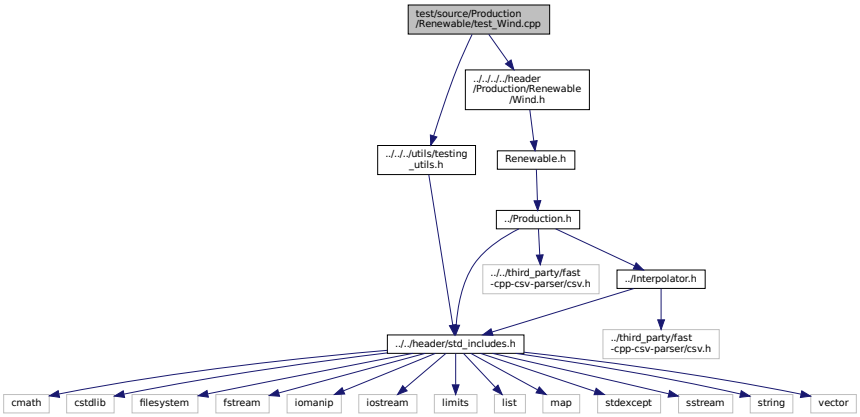
```
218             significant_wave_height_m,
219             0,
220             __FILE__,
221             __LINE__
222         );
223 
224         testGreaterThan(
225             energy_period_s,
226             0,
227             __FILE__,
228             __LINE__
229         );
230 
231         testGreaterThan(
232             test_wave_ptr->operation_maintenance_cost_vec[i],
233             0,
234             __FILE__,
235             __LINE__
236         );
237     }
238 
239     // O&M = 0 whenever wave is not running (i.e., not producing)
240     else {
241         testFloatEquals(
242             test_wave_ptr->operation_maintenance_cost_vec[i],
243             0,
244             __FILE__,
245             __LINE__
246         );
247     }
248 }
249 // ======== END METHODS ============================================================ //
250 
251 }   /* try */
252 
253 
254 catch (...) {
255     delete test_wave_ptr;
256 
257     printGold(" ...... ");
258     printRed("FAIL");
259     std::cout « std::endl;
260     throw;
261 }
262 
263 
264 delete test_wave_ptr;
265 
266 printGold(" ...... ");
267 printGreen("PASS");
268 std::cout « std::endl;
269 return 0;
270 }   /* main() */
```

5.39 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

- int main (int argc, char **argv)

5.39.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind class.

5.39.2 Function Documentation

5.39.2.1 main()

```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wind");
33
34     srand(time(NULL));
35
36     Renewable* test_wind_ptr;
37
38 try {
39
40 // ======== CONSTRUCTION ============================================================ //
41
42 bool error_flag = true;
43
44 try {
45     WindInputs bad_wind_inputs;
46     bad_wind_inputs.design_speed_ms = -1;
```

```
47
48      Wind bad_wind(8760, 1, bad_wind_inputs);
49
50      error_flag = false;
51  } catch (...) {
52      // Task failed successfully! =P
53  }
54  if (not error_flag) {
55      expectedErrorNotDetected(__FILE__, __LINE__);
56  }
57
58  WindInputs wind_inputs;
59
60  test_wind_ptr = new Wind(8760, 1, wind_inputs);
61
62  // ======== END CONSTRUCTION ============================================================ //
63
64
65
66  // ======== ATTRIBUTES ================================================================== //
67
68  testTruth(
69      not wind_inputs.renewable_inputs.production_inputs.print_flag,
70      __FILE__,
71      __LINE__
72  );
73
74  testFloatEquals(
75      test_wind_ptr->type,
76      RenewableType :: WIND,
77      __FILE__,
78      __LINE__
79  );
80
81  testTruth(
82      test_wind_ptr->type_str == "WIND",
83      __FILE__,
84      __LINE__
85  );
86
87  testFloatEquals(
88      test_wind_ptr->capital_cost,
89      450356.170088,
90      __FILE__,
91      __LINE__
92  );
93
94  testFloatEquals(
95      test_wind_ptr->operation_maintenance_cost_kWh,
96      0.034953,
97      __FILE__,
98      __LINE__
99  );
100
101 // ======== END ATTRIBUTES ============================================================== //
102
103
104
105 // ======== METHODS ===================================================================== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wind_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wind_ptr->computeProductionkW(
117         0,
118         1,
119         ((Wind*)test_wind_ptr)->design_speed_ms
120     ),
121     test_wind_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_wind_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()
```

```
134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double wind_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         wind_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164
165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_wind_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         wind_resource_ms
176     );
177
178     load_kW = test_wind_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_wind_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193
194     else {
195         testTruth(
196             not test_wind_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_wind_ptr->production_vec_kW[i] -
213         test_wind_ptr->dispatch_vec_kW[i] -
214         test_wind_ptr->storage_vec_kW[i] -
215         test_wind_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220
```

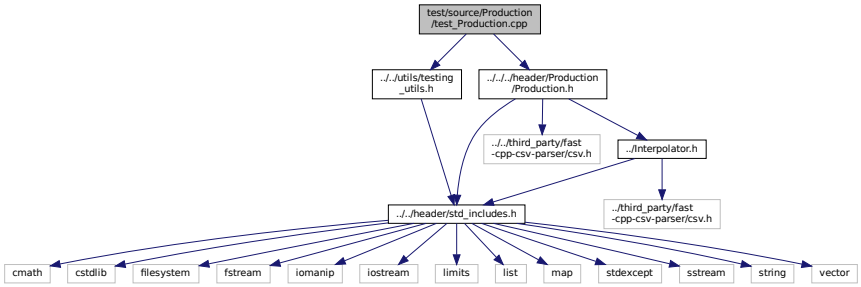
```
221     // resource, O&M > 0 whenever wind is running (i.e., producing)
222     if (test_wind_ptr->is_running) {
223         testGreaterThan(
224             wind_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229 
230         testGreaterThan(
231             test_wind_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237 
238     // O&M = 0 whenever wind is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_wind_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248 
249 
250 // ======== END METHODS ============================================================ //
251 
252 }   /* try */
253 
254 
255 catch (...) {
256     delete test_wind_ptr;
257 
258     printGold(" ...... ");
259     printRed("FAIL");
260     std::cout « std::endl;
261     throw;
262 }
263 
264 
265 delete test_wind_ptr;
266 
267 printGold(" ...... ");
268 printGreen("PASS");
269 std::cout « std::endl;
270 return 0;
271 }   /* main() */
```

5.40 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

- int main (int argc, char **argv)

5.40.1 Detailed Description

Testing suite for Production class.

A suite of tests for the Production class.

5.40.2 Function Documentation

5.40.2.1 main()

```
int main (
              int argc,
              char ** argv )
```

```
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31 
32     printGold("\tTesting Production");
33 
34     srand(time(NULL));
35 
36 
37 try {
38 
39 // ======== CONSTRUCTION ========================================================== //
40 
41 bool error_flag = true;
42 
43 try {
44     ProductionInputs production_inputs;
45 
46     Production bad_production(0, 1, production_inputs);
47 
48     error_flag = false;
49 } catch (...) {
50     // Task failed successfully! =P
51 }
52 if (not error_flag) {
53     expectedErrorNotDetected(__FILE__, __LINE__);
54 }
55 
56 ProductionInputs production_inputs;
57 
58 Production test_production(8760, 1, production_inputs);
59 
60 // ======== END CONSTRUCTION ====================================================== //
61 
62 
63 
64 // ======== ATTRIBUTES ============================================================ //
65 
66 testTruth(
67     not production_inputs.print_flag,
68     __FILE__,
69     __LINE__
70 );
71 
72 testFloatEquals(
73     production_inputs.nominal_inflation_annual,
74     0.02,
75     __FILE__,
76     __LINE__
77 );
```

```
78
79 testFloatEquals(
80     production_inputs.nominal_discount_annual,
81     0.04,
82     __FILE__,
83     __LINE__
84 );
85
86 testFloatEquals(
87     test_production.n_points,
88     8760,
89     __FILE__,
90     __LINE__
91 );
92
93 testFloatEquals(
94     test_production.capacity_kW,
95     100,
96     __FILE__,
97     __LINE__
98 );
99
100 testFloatEquals(
101     test_production.real_discount_annual,
102     0.0196078431372549,
103     __FILE__,
104     __LINE__
105 );
106
107 testFloatEquals(
108     test_production.production_vec_kW.size(),
109     8760,
110     __FILE__,
111     __LINE__
112 );
113
114 testFloatEquals(
115     test_production.dispatch_vec_kW.size(),
116     8760,
117     __FILE__,
118     __LINE__
119 );
120
121 testFloatEquals(
122     test_production.storage_vec_kW.size(),
123     8760,
124     __FILE__,
125     __LINE__
126 );
127
128 testFloatEquals(
129     test_production.curtailment_vec_kW.size(),
130     8760,
131     __FILE__,
132     __LINE__
133 );
134
135 testFloatEquals(
136     test_production.capital_cost_vec.size(),
137     8760,
138     __FILE__,
139     __LINE__
140 );
141
142 testFloatEquals(
143     test_production.operation_maintenance_cost_vec.size(),
144     8760,
145     __FILE__,
146     __LINE__
147 );
148
149 // ======== END ATTRIBUTES ============================================================ //
150
151 }   /* try */
152
153
154 catch (...) {
155     //...
156
157     printGold(" ............................. ");
158     printRed("FAIL");
159     std::cout « std::endl;
160     throw;
161 }
162
163
164 printGold(" ............................. ");
```

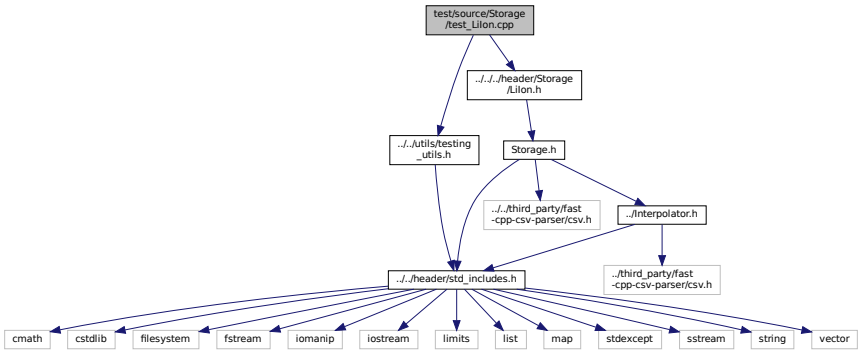
```
165 printGreen("PASS");
166 std::cout « std::endl;
167 return 0;
168
169 }   /* main() */
```

5.41 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

- int main (int argc, char **argv)

5.41.1 Detailed Description

Testing suite for LiIon class.

A suite of tests for the LiIon class.

5.41.2 Function Documentation

5.41.2.1 main()

```
int main (
             int argc,
             char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Storage <-- LiIon");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    LiIonInputs bad_liion_inputs;
    bad_liion_inputs.min_SOC = -1;

    LiIon bad_liion(8760, 1, bad_liion_inputs);

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

LiIonInputs liion_inputs;

LiIon test_liion(8760, 1, liion_inputs);

// ======== END CONSTRUCTION ======================================================== //


// ======== ATTRIBUTES ============================================================== //

testTruth(
    test_liion.type_str == "LIION",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.init_SOC,
    0.5,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.min_SOC,
    0.15,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.hysteresis_SOC,
    0.5,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.max_SOC,
    0.9,
    __FILE__,
    __LINE__
);

testFloatEquals(
     test_liion.charging_efficiency,
     0.9,
     __FILE__,
     __LINE__
);
```

```
107
108 testFloatEquals(
109     test_liion.discharging_efficiency,
110     0.9,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_liion.replace_SOH,
117     0.8,
118     __FILE__,
119     __LINE__
120 );
121
122 testFloatEquals(
123     test_liion.power_kW,
124     0,
125     __FILE__,
126     __LINE__
127 );
128
129 testFloatEquals(
130     test_liion.SOH_vec.size(),
131     8760,
132     __FILE__,
133     __LINE__
134 );
135
136 // ======== END ATTRIBUTES ============================================================ //
137
138
139
140 // ======== METHODS ================================================================== //
141
142 testFloatEquals(
143     test_liion.getAvailablekW(1),
144     100,    // hits power capacity constraint
145     __FILE__,
146     __LINE__
147 );
148
149 testFloatEquals(
150     test_liion.getAcceptablekW(1),
151     100,    // hits power capacity constraint
152     __FILE__,
153     __LINE__
154 );
155
156 test_liion.power_kW = 100;
157
158 testFloatEquals(
159     test_liion.getAvailablekW(1),
160     100,    // hits power capacity constraint
161     __FILE__,
162     __LINE__
163 );
164
165 testFloatEquals(
166     test_liion.getAcceptablekW(1),
167     100,    // hits power capacity constraint
168     __FILE__,
169     __LINE__
170 );
171
172 test_liion.power_kW = 1e6;
173
174 testFloatEquals(
175     test_liion.getAvailablekW(1),
176     0,    // is already hitting power capacity constraint
177     __FILE__,
178     __LINE__
179 );
180
181 testFloatEquals(
182     test_liion.getAcceptablekW(1),
183     0,    // is already hitting power capacity constraint
184     __FILE__,
185     __LINE__
186 );
187
188 test_liion.commitCharge(0, 1, 100);
189
190 testFloatEquals(
191     test_liion.power_kW,
192     0,
193     __FILE__,
```

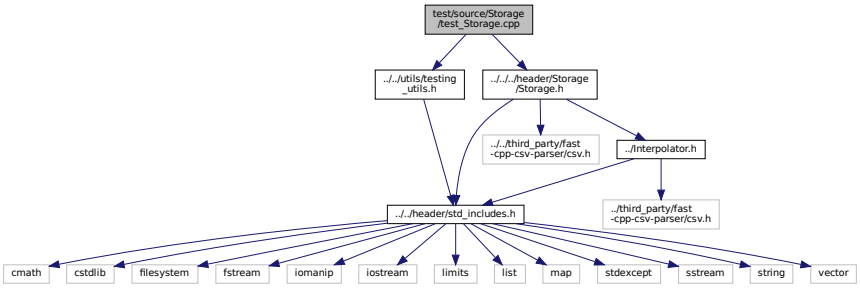
```
194     __LINE__
195 );
196
197 // ======== END METHODS ============================================================ //
198
199 } /* try */
200
201
202 catch (...) {
203     //...
204
205     printGold(" ..................... ");
206     printRed("FAIL");
207     std::cout « std::endl;
208     throw;
209 }
210
211
212 printGold(" ..................... ");
213 printGreen("PASS");
214 std::cout « std::endl;
215 return 0;
216 }   /* main() */
```

5.42 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

- int main (int argc, char ∗∗argv)

5.42.1 Detailed Description

Testing suite for Storage class.

A suite of tests for the Storage class.

5.42.2 Function Documentation

5.42.2.1 main()

```
int main (
             int argc,
             char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Storage");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ========================================================== //
40
41 bool error_flag = true;
42
43 try {
44     StorageInputs bad_storage_inputs;
45     bad_storage_inputs.energy_capacity_kWh = 0;
46
47     Storage bad_storage(8760, 1, bad_storage_inputs);
48
49     error_flag = false;
50 } catch (...) {
51     // Task failed successfully! =P
52 }
53 if (not error_flag) {
54     expectedErrorNotDetected(__FILE__, __LINE__);
55 }
56
57 StorageInputs storage_inputs;
58
59 Storage test_storage(8760, 1, storage_inputs);
60
61 // ======== END CONSTRUCTION ====================================================== //
62
63
64
65 // ======== ATTRIBUTES ============================================================ //
66
67 testFloatEquals(
68     test_storage.power_capacity_kW,
69     100,
70     __FILE__,
71     __LINE__
72 );
73
74 testFloatEquals(
75     test_storage.energy_capacity_kWh,
76     1000,
77     __FILE__,
78     __LINE__
79 );
80
81 testFloatEquals(
82     test_storage.charge_vec_kWh.size(),
83     8760,
84     __FILE__,
85     __LINE__
86 );
87
88 testFloatEquals(
89     test_storage.charging_power_vec_kW.size(),
90     8760,
91     __FILE__,
92     __LINE__
93 );
94
95 testFloatEquals(
96     test_storage.discharging_power_vec_kW.size(),
97     8760,
98     __FILE__,
99     __LINE__
100 );
101
102 testFloatEquals(
103     test_storage.capital_cost_vec.size(),
104     8760,
105     __FILE__,
106     __LINE__
```

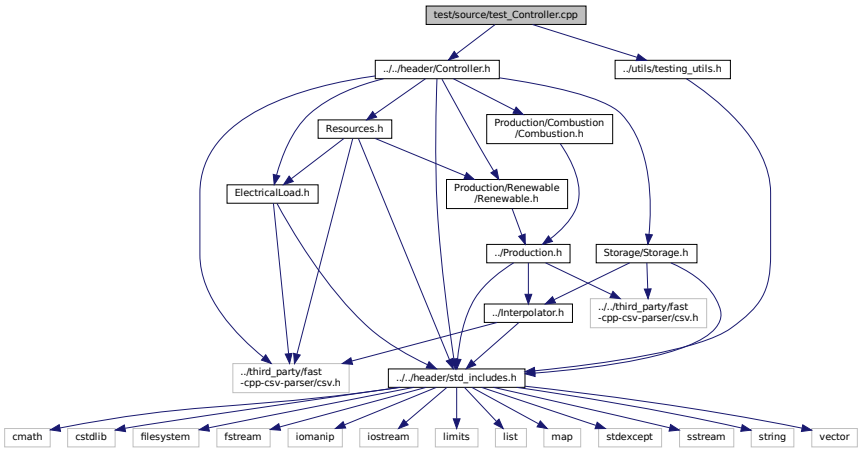
```
107 );
108
109 testFloatEquals(
110     test_storage.operation_maintenance_cost_vec.size(),
111     8760,
112     __FILE__,
113     __LINE__
114 );
115
116 // ======== END ATTRIBUTES ============================================================ //
117
118
119
120 // ======== METHODS ================================================================== //
121
122 //...
123
124 // ======== END METHODS ============================================================== //
125
126 }   /* try */
127
128
129 catch (...) {
130     //...
131
132     printGold(" ...................... ");
133     printRed("FAIL");
134     std::cout « std::endl;
135     throw;
136 }
137
138
139 printGold(" ............................... ");
140 printGreen("PASS");
141 std::cout « std::endl;
142 return 0;
143 }   /* main() */
```

5.43 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

- int main (int argc, char **argv)

5.43.1 Detailed Description

Testing suite for Controller class.

A suite of tests for the Controller class.

5.43.2 Function Documentation

5.43.2.1 main()

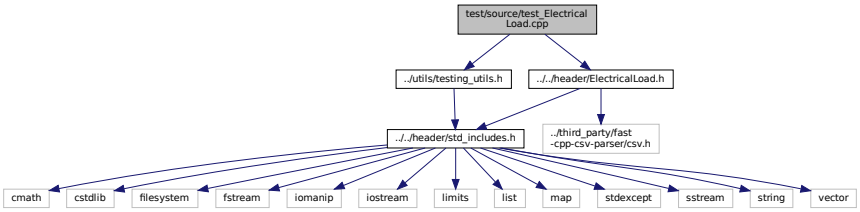
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Controller");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ========================================================== //
40
41 Controller test_controller;
42
43 // ======== END CONSTRUCTION ========================================================//
44
45
46
47 // ======== ATTRIBUTES ============================================================ //
48
49 //...
50
51 // ======== END ATTRIBUTES ======================================================== //
52
53
54
55 // ======== METHODS =============================================================== //
56
57 //...
58
59 // ======== END METHODS ============================================================ //
60
61 }   /* try */
62
63
64 catch (...) {
65     //...
66
67     printGold(" ............................. ");
68     printRed("FAIL");
69     std::cout « std::endl;
70     throw;
71 }
72
73
74 printGold(" ............................. ");
75 printGreen("PASS");
76 std::cout « std::endl;
77 return 0;
78 }   /* main() */
```

5.44 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

- int main (int argc, char **argv)

5.44.1 Detailed Description

Testing suite for ElectricalLoad class.

A suite of tests for the ElectricalLoad class.

5.44.2 Function Documentation

5.44.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting ElectricalLoad");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ========================================================== //
40
41 std::string path_2_electrical_load_time_series =
42     "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
43
44 ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
45
```

```
46 // ======== END CONSTRUCTION ========================================================= //
47
48
49
50 // ======== ATTRIBUTES ================================================================ //
51
52 testTruth(
53     test_electrical_load.path_2_electrical_load_time_series ==
54     path_2_electrical_load_time_series,
55     __FILE__,
56     __LINE__
57 );
58
59 testFloatEquals(
60     test_electrical_load.n_points,
61     8760,
62     __FILE__,
63     __LINE__
64 );
65
66 testFloatEquals(
67     test_electrical_load.n_years,
68     0.999886,
69     __FILE__,
70     __LINE__
71 );
72
73 testFloatEquals(
74     test_electrical_load.min_load_kW,
75     82.1211213927802,
76     __FILE__,
77     __LINE__
78 );
79
80 testFloatEquals(
81     test_electrical_load.mean_load_kW,
82     258.373472633202,
83     __FILE__,
84     __LINE__
85 );
86
87
88 testFloatEquals(
89     test_electrical_load.max_load_kW,
90     500,
91     __FILE__,
92     __LINE__
93 );
94
95
96 std::vector<double> expected_dt_vec_hrs (48, 1);
97
98 std::vector<double> expected_time_vec_hrs = {
99      0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
100     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
101     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
102     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
103 };
104
105 std::vector<double> expected_load_vec_kW = {
106     360.253836463674,
107     355.171277826775,
108     353.776453532298,
109     353.75405737934,
110     346.592867404975,
111     340.132411175118,
112     337.354867340578,
113     340.644115618736,
114     363.639028500678,
115     378.787797779238,
116     372.215798201712,
117     395.093925731298,
118     402.325427142659,
119     386.907725462306,
120     380.709170928091,
121     372.062070914977,
122     372.328646856954,
123     391.841444284136,
124     394.029351759596,
125     383.369407765254,
126     381.093099675206,
127     382.604158946193,
128     390.744843709034,
129     383.13949492437,
130     368.150393976985,
131     364.629744480226,
132     363.572736804082,
```

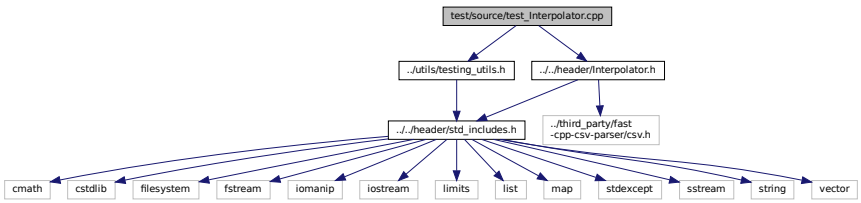
```
133     359.854924202248,
134     355.207590170267,
135     349.094656012401,
136     354.365935871597,
137     343.380608328546,
138     404.673065729266,
139     486.296896820126,
140     480.225974100847,
141     457.318764401085,
142     418.177339948609,
143     414.399018364126,
144     409.678420185754,
145     404.768766016563,
146     401.699589920585,
147     402.44339040654,
148     398.138372541906,
149     396.010498627646,
150     390.165117432277,
151     375.850429417013,
152     365.567100746484,
153     365.429624610923
154 };
155
156 for (int i = 0; i < 48; i++) {
157     testFloatEquals(
158         test_electrical_load.dt_vec_hrs[i],
159         expected_dt_vec_hrs[i],
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals(
165         test_electrical_load.time_vec_hrs[i],
166         expected_time_vec_hrs[i],
167         __FILE__,
168         __LINE__
169     );
170
171     testFloatEquals(
172         test_electrical_load.load_vec_kW[i],
173         expected_load_vec_kW[i],
174         __FILE__,
175         __LINE__
176     );
177 }
178
179 // ======== END ATTRIBUTES ============================================================ //
180
181 }   /* try */
182
183
184 catch (...) {
185     //...
186
187     printGold(" ......................... ");
188     printRed("FAIL");
189     std::cout « std::endl;
190     throw;
191 }
192
193
194 printGold(" ......................... ");
195 printGreen("PASS");
196 std::cout « std::endl;
197 return 0;
198 }   /* main() */
```

5.45 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

- int main (int argc, char **argv)

5.45.1 Detailed Description

Testing suite for Interpolator class.

A suite of tests for the Interpolator class.

5.45.2 Function Documentation

5.45.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31 
32     printGold("\n\tTesting Interpolator");
33 
34     srand(time(NULL));
35 
36 
37 try {
38 
39 // ======== CONSTRUCTION ========================================================== //
40 
41 Interpolator test_interpolator;
42 
43 // ======== END CONSTRUCTION ====================================================//
44 
45 
46 
47 // ======== ATTRIBUTES ============================================================ //
48 
49 //...
50 
51 // ======== END ATTRIBUTES ======================================================== //
52 
53 
54 
55 // ======== METHODS =============================================================== //
56 
```

```
57 //  1. 1D interpolation
58
59 int data_key = 1;
60 std::string path_2_data = "data/test/interpolation/diesel_fuel_curve.csv";
61
62 test_interpolator.addData1D(data_key, path_2_data);
63
64 testTruth(
65     test_interpolator.path_map_1D[data_key] == path_2_data,
66     __FILE__,
67     __LINE__
68 );
69
70 testFloatEquals(
71     test_interpolator.interp_map_1D[data_key].n_points,
72     16,
73     __FILE__,
74     __LINE__
75 );
76
77 testFloatEquals(
78     test_interpolator.interp_map_1D[data_key].x_vec.size(),
79     16,
80     __FILE__,
81     __LINE__
82 );
83
84 std::vector<double> expected_x_vec = {
85     0,
86     0.3,
87     0.35,
88     0.4,
89     0.45,
90     0.5,
91     0.55,
92     0.6,
93     0.65,
94     0.7,
95     0.75,
96     0.8,
97     0.85,
98     0.9,
99     0.95,
100     1
101 };
102
103 std::vector<double> expected_y_vec = {
104     4.68079520372916,
105     11.1278522361839,
106     12.4787834830748,
107     13.7808847600209,
108     15.0417468303382,
109     16.277263,
110     17.4612831516442,
111     18.6279054806525,
112     19.7698039220515,
113     20.8893499214868,
114     21.955378,
115     23.0690535155297,
116     24.1323614374927,
117     25.1797231192866,
118     26.2122451458747,
119     27.254952
120 };
121
122 for (int i = 0; i < test_interpolator.interp_map_1D[data_key].n_points; i++) {
123     testFloatEquals(
124         test_interpolator.interp_map_1D[data_key].x_vec[i],
125         expected_x_vec[i],
126         __FILE__,
127         __LINE__
128     );
129
130     testFloatEquals(
131         test_interpolator.interp_map_1D[data_key].y_vec[i],
132         expected_y_vec[i],
133         __FILE__,
134         __LINE__
135     );
136 }
137
138 testFloatEquals(
139     test_interpolator.interp_map_1D[data_key].min_x,
140     expected_x_vec[0],
141     __FILE__,
142     __LINE__
143 );
```

```
144
145 testFloatEquals(
146     test_interpolator.interp_map_1D[data_key].max_x,
147     expected_x_vec[expected_x_vec.size() - 1],
148     __FILE__,
149     __LINE__
150 );
151
152 std::vector<double> interp_x_vec = {
153     0,
154     0.170812859791767,
155     0.322739274162545,
156     0.369750203682042,
157     0.443532869135929,
158     0.471567864244626,
159     0.536513734479662,
160     0.586125806988674,
161     0.601101175455075,
162     0.658356862575221,
163     0.70576929893201,
164     0.784069734739331,
165     0.805765927542453,
166     0.884747873186048,
167     0.930870496062112,
168     0.979415217694769,
169     1
170 };
171
172 std::vector<double> expected_interp_y_vec = {
173     4.68079520372916,
174     8.35159603357656,
175     11.7422361561399,
176     12.9931187917615,
177     14.8786636301325,
178     15.5746957307243,
179     17.1419229487141,
180     18.3041866133728,
181     18.6530540913696,
182     19.9569217633299,
183     21.012354614584,
184     22.7142305879957,
185     23.1916726441968,
186     24.8602332554707,
187     25.8172124624032,
188     26.8256741279932,
189     27.254952
190 };
191
192 for (size_t i = 0; i < interp_x_vec.size(); i++) {
193     testFloatEquals(
194         test_interpolator.interp1D(data_key, interp_x_vec[i]),
195         expected_interp_y_vec[i],
196         __FILE__,
197         __LINE__
198     );
199 }
200
201
202 //  2. 2D interpolation
203
204 data_key = 2;
205 path_2_data =
206     "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
207
208 test_interpolator.addData2D(data_key, path_2_data);
209
210 testTruth(
211     test_interpolator.path_map_2D[data_key] == path_2_data,
212     __FILE__,
213     __LINE__
214 );
215
216 testFloatEquals(
217     test_interpolator.interp_map_2D[data_key].n_rows,
218     16,
219     __FILE__,
220     __LINE__
221 );
222
223 testFloatEquals(
224     test_interpolator.interp_map_2D[data_key].n_cols,
225     16,
226     __FILE__,
227     __LINE__
228 );
229
230 testFloatEquals(
```

```
231     test_interpolator.interp_map_2D[data_key].x_vec.size(),
232     16,
233     __FILE__,
234     __LINE__
235 );
236
237 testFloatEquals(
238     test_interpolator.interp_map_2D[data_key].y_vec.size(),
239     16,
240     __FILE__,
241     __LINE__
242 );
243
244 testFloatEquals(
245     test_interpolator.interp_map_2D[data_key].z_matrix.size(),
246     16,
247     __FILE__,
248     __LINE__
249 );
250
251 testFloatEquals(
252     test_interpolator.interp_map_2D[data_key].z_matrix[0].size(),
253     16,
254     __FILE__,
255     __LINE__
256 );
257
258 expected_x_vec = {
259     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
260 };
261
262 expected_y_vec = {
263     5,
264     6,
265     7,
266     8,
267     9,
268     10,
269     11,
270     12,
271     13,
272     14,
273     15,
274     16,
275     17,
276     18,
277     19,
278     20
279 };
280
281 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_cols; i++) {
282     testFloatEquals(
283         test_interpolator.interp_map_2D[data_key].x_vec[i],
284         expected_x_vec[i],
285         __FILE__,
286         __LINE__
287     );
288 }
289
290 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_rows; i++) {
291     testFloatEquals(
292         test_interpolator.interp_map_2D[data_key].y_vec[i],
293         expected_y_vec[i],
294         __FILE__,
295         __LINE__
296     );
297 }
298
299 testFloatEquals(
300     test_interpolator.interp_map_2D[data_key].min_x,
301     expected_x_vec[0],
302     __FILE__,
303     __LINE__
304 );
305
306 testFloatEquals(
307     test_interpolator.interp_map_2D[data_key].max_x,
308     expected_x_vec[expected_x_vec.size() - 1],
309     __FILE__,
310     __LINE__
311 );
312
313 testFloatEquals(
314     test_interpolator.interp_map_2D[data_key].min_y,
315     expected_y_vec[0],
316     __FILE__,
317     __LINE__
```

```
318 );
319
320 testFloatEquals(
321     test_interpolator.interp_map_2D[data_key].max_y,
322     expected_y_vec[expected_y_vec.size() - 1],
323     __FILE__,
324     __LINE__
325 );
326
327 std::vector<std::vector<double» expected_z_matrix = {
328     {0, 0.129128125, 0.268078125, 0.404253125, 0.537653125, 0.668278125, 0.796128125, 0.921203125, 1, 1,
      1, 0, 0, 0, 0, 0},
329     {0, 0.11160375, 0.24944375, 0.38395375, 0.51513375, 0.64298375, 0.76750375, 0.88869375, 1, 1, 1, 1,
      1, 1, 1, 1},
330     {0, 0.094079375, 0.230809375, 0.363654375, 0.492614375, 0.617689375, 0.738879375, 0.856184375,
      0.969604375, 1, 1, 1, 1, 1, 1, 1},
331     {0, 0.076555, 0.212175, 0.343355, 0.470095, 0.592395, 0.710255, 0.823675, 0.932655, 1, 1, 1, 1, 1,
      1, 1},
332     {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},
333     {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},
334     {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},
335     {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},
336     {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},
337     {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},
338     {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},
339     {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},
340     {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},
341     {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},
342     {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},
343     {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}
344 };
345
346 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_rows; i++) {
347     for (int j = 0; j < test_interpolator.interp_map_2D[data_key].n_cols; j++) {
348         testFloatEquals(
349             test_interpolator.interp_map_2D[data_key].z_matrix[i][j],
350             expected_z_matrix[i][j],
351             __FILE__,
352             __LINE__
353         );
354     }
355 }
356
357 interp_x_vec = {
358     0.389211848822208,
359     0.836477431896843,
360     1.52738334015579,
361     1.92640601114508,
362     2.27297317532019,
363     2.87416589636605,
364     3.72275770908175,
365     3.95063175885536,
366     4.68097139867404,
367     4.97775020449812,
368     5.55184219980547,
369     6.06566629451658,
370     6.27927876785062,
371     6.96218133671013,
372     7.51754442460228
373 };
374
375 std::vector<double> interp_y_vec = {
376     5.45741899698926,
377     6.00101329139007,
378     7.50567689404182,
379     8.77681262912881,
380     9.45143678206774,
381     10.7767876462885,
382     11.4795760857165,
383     12.9430684577599,
384     13.303544885703,
385     14.5069863517863,
386     15.1487890438045,
387     16.086524049077,
388     17.176609978648,
```

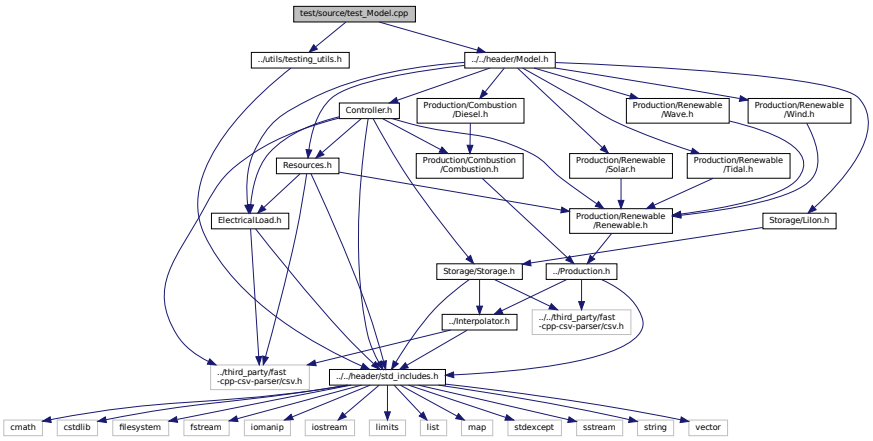
```
389     18.4155153740256,
390     19.1704554940162
391 };
392
393 std::vector<std::vector<double» expected_interp_z_matrix = {
394
    {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
395
    {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
396
    {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
397
    {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
398
    {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
399
    {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
400
    {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
401
    {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
402
    {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
403
    {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
404
    {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
405
    {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
406
    {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
407
    {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.510
408
    {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.484
409 };
410
411 for (size_t i = 0; i < interp_y_vec.size(); i++) {
412     for (size_t j = 0; j < interp_x_vec.size(); j++) {
413         testFloatEquals(
414             test_interpolator.interp2D(data_key, interp_x_vec[j], interp_y_vec[i]),
415             expected_interp_z_matrix[i][j],
416             __FILE__,
417             __LINE__
418         );
419     }
420 }
421
422 // ======== END METHODS ============================================================ //
423
424 }   /* try */
425
426
427 catch (...) {
428     //...
429
430     printGold(" ........................... ");
431     printRed("FAIL");
432     std::cout « std::endl;
433     throw;
434 }
435
436
437 printGold(" ........................... ");
438 printGreen("PASS");
439 std::cout « std::endl;
440 return 0;
441 }   /* main() */
```

5.46 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

- int main (int argc, char **argv)

5.46.1 Detailed Description

Testing suite for Model class.

A suite of tests for the Model class.

5.46.2 Function Documentation

5.46.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Model");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ============================================================ //
40
41 bool error_flag = true;
42
43 try {
44     ModelInputs bad_model_inputs;   // path_2_electrical_load_time_series left empty
45
```

```
46     Model bad_model(bad_model_inputs);
47
48     error_flag = false;
49 } catch (...) {
50     // Task failed successfully! =P
51 }
52 if (not error_flag) {
53     expectedErrorNotDetected(__FILE__, __LINE__);
54 }
55
56
57 try {
58     ModelInputs bad_model_inputs;
59     bad_model_inputs.path_2_electrical_load_time_series =
60         "data/test/electrical_load/bad_path_240984069830.csv";
61
62     Model bad_model(bad_model_inputs);
63
64     error_flag = false;
65 } catch (...) {
66     // Task failed successfully! =P
67 }
68 if (not error_flag) {
69     expectedErrorNotDetected(__FILE__, __LINE__);
70 }
71
72
73 std::string path_2_electrical_load_time_series =
74     "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
75
76 ModelInputs test_model_inputs;
77 test_model_inputs.path_2_electrical_load_time_series =
78     path_2_electrical_load_time_series;
79
80 Model test_model(test_model_inputs);
81
82 // ======== END CONSTRUCTION ============================================================ //
83
84
85 // ======== ATTRIBUTES ================================================================== //
86
87 testTruth(
88     test_model.electrical_load.path_2_electrical_load_time_series ==
89     path_2_electrical_load_time_series,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_model.electrical_load.n_points,
96     8760,
97     __FILE__,
98     __LINE__
99 );
100
101 testFloatEquals(
102     test_model.electrical_load.n_years,
103     0.999886,
104     __FILE__,
105     __LINE__
106 );
107
108 testFloatEquals(
109     test_model.electrical_load.min_load_kW,
110     82.1211213927802,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_model.electrical_load.mean_load_kW,
117     258.373472633202,
118     __FILE__,
119     __LINE__
120 );
121
122
123 testFloatEquals(
124     test_model.electrical_load.max_load_kW,
125     500,
126     __FILE__,
127     __LINE__
128 );
129
130
131 std::vector<double> expected_dt_vec_hrs (48, 1);
132
```

```
133 std::vector<double> expected_time_vec_hrs = {
134      0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
135     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
136     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
137     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
138 };
139
140 std::vector<double> expected_load_vec_kW = {
141     360.253836463674,
142     355.171277826775,
143     353.776453532298,
144     353.75405737934,
145     346.592867404975,
146     340.132411175118,
147     337.354867340578,
148     340.644115618736,
149     363.639028500678,
150     378.787797779238,
151     372.215798201712,
152     395.093925731298,
153     402.325427142659,
154     386.907725462306,
155     380.709170928091,
156     372.062070914977,
157     372.328646856954,
158     391.841444284136,
159     394.029351759596,
160     383.369407765254,
161     381.093099675206,
162     382.604158946193,
163     390.744843709034,
164     383.13949492437,
165     368.150393976985,
166     364.629744480226,
167     363.572736804082,
168     359.854924202248,
169     355.207590170267,
170     349.094656012401,
171     354.365935871597,
172     343.380608328546,
173     404.673065729266,
174     486.296896820126,
175     480.225974100847,
176     457.318764401085,
177     418.177339948609,
178     414.399018364126,
179     409.678420185754,
180     404.768766016563,
181     401.699589920585,
182     402.44339040654,
183     398.138372541906,
184     396.010498627646,
185     390.165117432277,
186     375.850429417013,
187     365.567100746484,
188     365.429624610923
189 };
190
191 for (int i = 0; i < 48; i++) {
192     testFloatEquals(
193         test_model.electrical_load.dt_vec_hrs[i],
194         expected_dt_vec_hrs[i],
195         __FILE__,
196         __LINE__
197     );
198
199     testFloatEquals(
200         test_model.electrical_load.time_vec_hrs[i],
201         expected_time_vec_hrs[i],
202         __FILE__,
203         __LINE__
204     );
205
206     testFloatEquals(
207         test_model.electrical_load.load_vec_kW[i],
208         expected_load_vec_kW[i],
209         __FILE__,
210         __LINE__
211     );
212 }
213
214 // ======== END ATTRIBUTES ============================================================ //
215
216
217
218 // ======== METHODS ================================================================== //
219
```

```
220 //  add Solar resource
221 int solar_resource_key = 0;
222 std::string path_2_solar_resource_data =
223     "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
224
225 test_model.addResource(
226     RenewableType :: SOLAR,
227     path_2_solar_resource_data,
228     solar_resource_key
229 );
230
231 std::vector<double> expected_solar_resource_vec_kWm2 = {
232     0,
233     0,
234     0,
235     0,
236     0,
237     0,
238     8.51702662684015E-05,
239     0.000348341567045,
240     0.00213793728593,
241     0.004099863613322,
242     0.000997135230553,
243     0.009534527624657,
244     0.022927996790616,
245     0.0136071715294,
246     0.002535134127751,
247     0.005206897515821,
248     0.005627658648597,
249     0.000701186722215,
250     0.00017119827089,
251     0,
252     0,
253     0,
254     0,
255     0,
256     0,
257     0,
258     0,
259     0,
260     0,
261     0,
262     0,
263     0.000141055102242,
264     0.00084525014743,
265     0.024893647822702,
266     0.091245556190749,
267     0.158722176731637,
268     0.152859680515876,
269     0.149922903895116,
270     0.13049996570866,
271     0.03081254222795,
272     0.001218928911125,
273     0.000206092647423,
274     0,
275     0,
276     0,
277     0,
278     0,
279     0
280 };
281
282 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
283     testFloatEquals(
284         test_model.resources.resource_map_1D[solar_resource_key][i],
285         expected_solar_resource_vec_kWm2[i],
286         __FILE__,
287         __LINE__
288     );
289 }
290
291
292 //  add Tidal resource
293 int tidal_resource_key = 1;
294 std::string path_2_tidal_resource_data =
295     "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
296
297 test_model.addResource(
298     RenewableType :: TIDAL,
299     path_2_tidal_resource_data,
300     tidal_resource_key
301 );
302
303
304 //  add Wave resource
305 int wave_resource_key = 2;
306 std::string path_2_wave_resource_data =
```

```
307     "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
308
309 test_model.addResource(
310     RenewableType :: WAVE,
311     path_2_wave_resource_data,
312     wave_resource_key
313 );
314
315
316 //  add Wind resource
317 int wind_resource_key = 3;
318 std::string path_2_wind_resource_data =
319     "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
320
321 test_model.addResource(
322     RenewableType :: WIND,
323     path_2_wind_resource_data,
324     wind_resource_key
325 );
326
327
328 //  add Diesel assets
329 DieselInputs diesel_inputs;
330 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
331 diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
332
333 test_model.addDiesel(diesel_inputs);
334
335 testFloatEquals(
336     test_model.combustion_ptr_vec.size(),
337     1,
338     __FILE__,
339     __LINE__
340 );
341
342 testFloatEquals(
343     test_model.combustion_ptr_vec[0]->type,
344     CombustionType :: DIESEL,
345     __FILE__,
346     __LINE__
347 );
348
349 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
350
351 test_model.addDiesel(diesel_inputs);
352
353 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
354
355 test_model.addDiesel(diesel_inputs);
356
357 testFloatEquals(
358     test_model.combustion_ptr_vec.size(),
359     3,
360     __FILE__,
361     __LINE__
362 );
363
364 std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
365
366 for (int i = 0; i < 3; i++) {
367     testFloatEquals(
368         test_model.combustion_ptr_vec[i]->capacity_kW,
369         expected_diesel_capacity_vec_kW[i],
370         __FILE__,
371         __LINE__
372     );
373 }
374
375 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
376
377 for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
378     test_model.addDiesel(diesel_inputs);
379 }
380
381
382 //  add Solar asset
383 SolarInputs solar_inputs;
384 solar_inputs.resource_key = solar_resource_key;
385
386 test_model.addSolar(solar_inputs);
387
388 testFloatEquals(
389     test_model.renewable_ptr_vec.size(),
390     1,
391     __FILE__,
392     __LINE__
393 );
```

```
394
395 testFloatEquals(
396     test_model.renewable_ptr_vec[0]->type,
397     RenewableType :: SOLAR,
398     __FILE__,
399     __LINE__
400 );
401
402
403 //  add Tidal asset
404 TidalInputs tidal_inputs;
405 tidal_inputs.resource_key = tidal_resource_key;
406
407 test_model.addTidal(tidal_inputs);
408
409 testFloatEquals(
410     test_model.renewable_ptr_vec.size(),
411     2,
412     __FILE__,
413     __LINE__
414 );
415
416 testFloatEquals(
417     test_model.renewable_ptr_vec[1]->type,
418     RenewableType :: TIDAL,
419     __FILE__,
420     __LINE__
421 );
422
423
424 //  add Wave asset
425 WaveInputs wave_inputs;
426 wave_inputs.resource_key = wave_resource_key;
427
428 test_model.addWave(wave_inputs);
429
430 testFloatEquals(
431     test_model.renewable_ptr_vec.size(),
432     3,
433     __FILE__,
434     __LINE__
435 );
436
437 testFloatEquals(
438     test_model.renewable_ptr_vec[2]->type,
439     RenewableType :: WAVE,
440     __FILE__,
441     __LINE__
442 );
443
444
445 //  add Wind asset
446 WindInputs wind_inputs;
447 wind_inputs.resource_key = wind_resource_key;
448
449 test_model.addWind(wind_inputs);
450
451 testFloatEquals(
452     test_model.renewable_ptr_vec.size(),
453     4,
454     __FILE__,
455     __LINE__
456 );
457
458 testFloatEquals(
459     test_model.renewable_ptr_vec[3]->type,
460     RenewableType :: WIND,
461     __FILE__,
462     __LINE__
463 );
464
465
466 //  add LiIon asset
467 LiIonInputs liion_inputs;
468
469 test_model.addLiIon(liion_inputs);
470
471 testFloatEquals(
472     test_model.storage_ptr_vec.size(),
473     1,
474     __FILE__,
475     __LINE__
476 );
477
478 testFloatEquals(
479     test_model.storage_ptr_vec[0]->type,
480     StorageType :: LIION,
```

```
481     __FILE__,
482     __LINE__
483 );
484
485
486 //  run
487 test_model.run();
488
489
490 //  write results
491 test_model.writeResults("test/test_results/");
492
493
494 //  test post-run attributes
495 double net_load_kW;
496
497 Combustion* combustion_ptr;
498 Renewable* renewable_ptr;
499 Storage* storage_ptr;
500
501 for (int i = 0; i < test_model.electrical_load.n_points; i++) {
502     net_load_kW = test_model.controller.net_load_vec_kW[i];
503
504     testLessThanOrEqualTo(
505         test_model.controller.net_load_vec_kW[i],
506         test_model.electrical_load.max_load_kW,
507         __FILE__,
508         __LINE__
509     );
510
511     for (size_t j = 0; j < test_model.combustion_ptr_vec.size(); j++) {
512         combustion_ptr = test_model.combustion_ptr_vec[j];
513
514         testFloatEquals(
515             combustion_ptr->production_vec_kW[i] -
516             combustion_ptr->dispatch_vec_kW[i] -
517             combustion_ptr->curtailment_vec_kW[i] -
518             combustion_ptr->storage_vec_kW[i],
519             0,
520             __FILE__,
521             __LINE__
522         );
523
524         net_load_kW -= combustion_ptr->production_vec_kW[i];
525     }
526
527     for (size_t j = 0; j < test_model.renewable_ptr_vec.size(); j++) {
528         renewable_ptr = test_model.renewable_ptr_vec[j];
529
530         testFloatEquals(
531             renewable_ptr->production_vec_kW[i] -
532             renewable_ptr->dispatch_vec_kW[i] -
533             renewable_ptr->curtailment_vec_kW[i] -
534             renewable_ptr->storage_vec_kW[i],
535             0,
536             __FILE__,
537             __LINE__
538         );
539
540         net_load_kW -= renewable_ptr->production_vec_kW[i];
541     }
542
543     for (size_t j = 0; j < test_model.storage_ptr_vec.size(); j++) {
544         storage_ptr = test_model.storage_ptr_vec[j];
545
546         testTruth(
547             not (
548                 storage_ptr->charging_power_vec_kW[i] > 0 and
549                 storage_ptr->discharging_power_vec_kW[i] > 0
550             ),
551             __FILE__,
552             __LINE__
553         );
554
555         net_load_kW -= storage_ptr->discharging_power_vec_kW[i];
556     }
557
558     testLessThanOrEqualTo(
559         net_load_kW,
560         0,
561         __FILE__,
562         __LINE__
563     );
564 }
565
566 testGreaterThan(
567     test_model.net_present_cost,
```

```
568     0,
569     __FILE__,
570     __LINE__
571 );
572
573 testFloatEquals(
574     test_model.total_dispatch_discharge_kWh,
575     2263351.62026685,
576     __FILE__,
577     __LINE__
578 );
579
580 testGreaterThan(
581     test_model.levellized_cost_of_energy_kWh,
582     0,
583     __FILE__,
584     __LINE__
585 );
586
587 testGreaterThan(
588     test_model.total_fuel_consumed_L,
589     0,
590     __FILE__,
591     __LINE__
592 );
593
594 testGreaterThan(
595     test_model.total_emissions.CO2_kg,
596     0,
597     __FILE__,
598     __LINE__
599 );
600
601 testGreaterThan(
602     test_model.total_emissions.CO_kg,
603     0,
604     __FILE__,
605     __LINE__
606 );
607
608 testGreaterThan(
609     test_model.total_emissions.NOx_kg,
610     0,
611     __FILE__,
612     __LINE__
613 );
614
615 testGreaterThan(
616     test_model.total_emissions.SOx_kg,
617     0,
618     __FILE__,
619     __LINE__
620 );
621
622 testGreaterThan(
623     test_model.total_emissions.CH4_kg,
624     0,
625     __FILE__,
626     __LINE__
627 );
628
629 testGreaterThan(
630     test_model.total_emissions.PM_kg,
631     0,
632     __FILE__,
633     __LINE__
634 );
635
636 // ======== END METHODS ================================================================ //
637
638 }   /* try */
639
640
641 catch (...) {
642     //...
643
644     printGold(" ................................. ");
645     printRed("FAIL");
646     std::cout « std::endl;
647     throw;
648 }
649
650
651 printGold(" ................................. ");
652 printGreen("PASS");
653 std::cout « std::endl;
654 return 0;
```

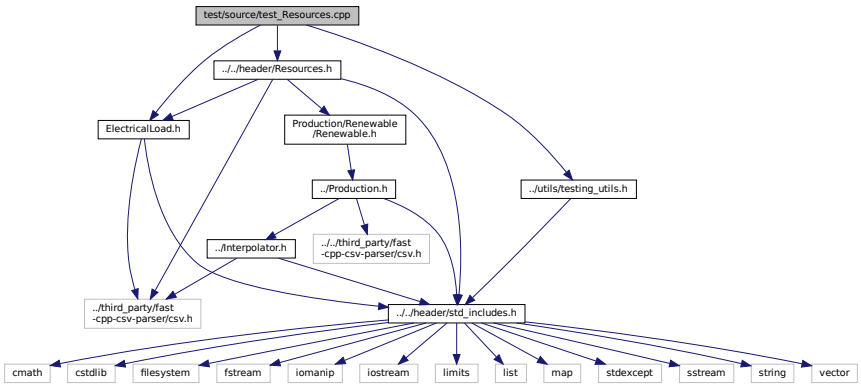
```
655 }   /* main() */
```

5.47 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

- int main (int argc, char **argv)

5.47.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

5.47.2 Function Documentation

5.47.2.1 main()

```
int main (
             int argc,
             char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Resources");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ========================================================== //

std::string path_2_electrical_load_time_series =
    "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);

Resources test_resources;

// ======== END CONSTRUCTION ====================================================== //



// ======== ATTRIBUTES ============================================================ //

testFloatEquals(
    test_resources.resource_map_1D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.path_map_1D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.resource_map_2D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.path_map_2D.size(),
    0,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ======================================================== //


// ======== METHODS =============================================================== //

int solar_resource_key = 0;
std::string path_2_solar_resource_data =
    "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

test_resources.addResource(
    RenewableType::SOLAR,
    path_2_solar_resource_data,
    solar_resource_key,
    &test_electrical_load
);

bool error_flag = true;
try {
    test_resources.addResource(
        RenewableType::SOLAR,
        path_2_solar_resource_data,
        solar_resource_key,
        &test_electrical_load
    );

```

```
108     error_flag = false;
109 } catch (...) {
110     // Task failed successfully! =P
111 }
112 if (not error_flag) {
113     expectedErrorNotDetected(__FILE__, __LINE__);
114 }
115
116
117 try {
118     std::string path_2_solar_resource_data_BAD_TIMES =
119         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";
120
121     test_resources.addResource(
122         RenewableType::SOLAR,
123         path_2_solar_resource_data_BAD_TIMES,
124         -1,
125         &test_electrical_load
126     );
127
128     error_flag = false;
129 } catch (...) {
130     // Task failed successfully! =P
131 }
132 if (not error_flag) {
133     expectedErrorNotDetected(__FILE__, __LINE__);
134 }
135
136
137 try {
138     std::string path_2_solar_resource_data_BAD_LENGTH =
139         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";
140
141     test_resources.addResource(
142         RenewableType::SOLAR,
143         path_2_solar_resource_data_BAD_LENGTH,
144         -2,
145         &test_electrical_load
146     );
147
148     error_flag = false;
149 } catch (...) {
150     // Task failed successfully! =P
151 }
152 if (not error_flag) {
153     expectedErrorNotDetected(__FILE__, __LINE__);
154 }
155
156 std::vector<double> expected_solar_resource_vec_kWm2 = {
157     0,
158     0,
159     0,
160     0,
161     0,
162     0,
163     8.51702662684015E-05,
164     0.000348341567045,
165     0.00213793728593,
166     0.004099863613322,
167     0.000997135230553,
168     0.009534527624657,
169     0.022927996790616,
170     0.0136071715294,
171     0.002535134127751,
172     0.005206897515821,
173     0.005627658648597,
174     0.000701186722215,
175     0.00017119827089,
176     0,
177     0,
178     0,
179     0,
180     0,
181     0,
182     0,
183     0,
184     0,
185     0,
186     0,
187     0,
188     0.000141055102242,
189     0.00084525014743,
190     0.024893647822702,
191     0.091245556190749,
192     0.158722176731637,
193     0.152859680515876,
194     0.149922903895116,
```

```
195     0.13049996570866,
196     0.03081254222795,
197     0.001218928911125,
198     0.000206092647423,
199     0,
200     0,
201     0,
202     0,
203     0,
204     0
205 };
206
207 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
208     testFloatEquals(
209         test_resources.resource_map_1D[solar_resource_key][i],
210         expected_solar_resource_vec_kWm2[i],
211         __FILE__,
212         __LINE__
213     );
214 }
215
216
217 int tidal_resource_key = 1;
218 std::string path_2_tidal_resource_data =
219     "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
220
221 test_resources.addResource(
222     RenewableType::TIDAL,
223     path_2_tidal_resource_data,
224     tidal_resource_key,
225     &test_electrical_load
226 );
227
228 std::vector<double> expected_tidal_resource_vec_ms = {
229     0.347439913040533,
230     0.770545522195602,
231     0.731352084836198,
232     0.293389814389542,
233     0.209959110813115,
234     0.610609623896497,
235     1.78067162013604,
236     2.53522775118089,
237     2.75966627832024,
238     2.52101111143895,
239     2.05389330201031,
240     1.3461515862445,
241     0.28909254878384,
242     0.897754086048563,
243     1.71406453837407,
244     1.85047408742869,
245     1.71507908595979,
246     1.33540349705416,
247     0.434586143463003,
248     0.500623815700637,
249     1.37172172646733,
250     1.68294125491228,
251     1.56101300975417,
252     1.04925834219412,
253     0.211395463930223,
254     1.03720048903385,
255     1.85059536356448,
256     1.85203242794517,
257     1.4091471616277,
258     0.767776539039899,
259     0.251464906990961,
260     1.47018469375652,
261     2.36260493698197,
262     2.46653750048625,
263     2.12851908739291,
264     1.62783753197988,
265     0.734594890957439,
266     0.441886297300355,
267     1.6574418350918,
268     2.0684558286637,
269     1.87717416992136,
270     1.58871262337931,
271     1.03451227609235,
272     0.193371305159817,
273     0.976400122458815,
274     1.6583227369707,
275     1.76690616570953,
276     1.54801328553115
277 };
278
279 for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
280     testFloatEquals(
281         test_resources.resource_map_1D[tidal_resource_key][i],
```

```
282         expected_tidal_resource_vec_ms[i],
283         __FILE__,
284         __LINE__
285     );
286 }
287
288
289 int wave_resource_key = 2;
290 std::string path_2_wave_resource_data =
291     "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
292
293 test_resources.addResource(
294     RenewableType::WAVE,
295     path_2_wave_resource_data,
296     wave_resource_key,
297     &test_electrical_load
298 );
299
300 std::vector<double> expected_significant_wave_height_vec_m = {
301     4.26175222125028,
302     4.25020976167872,
303     4.25656524330349,
304     4.27193854786718,
305     4.28744955711233,
306     4.29421815278154,
307     4.2839937266082,
308     4.25716982457976,
309     4.22419391611483,
310     4.19588925217606,
311     4.17338788587412,
312     4.14672746914214,
313     4.10560041173665,
314     4.05074966447193,
315     3.9953696962433,
316     3.95316976150866,
317     3.92771018142378,
318     3.91129562488595,
319     3.89558312094911,
320     3.87861093931749,
321     3.86538307240754,
322     3.86108961027929,
323     3.86459448853189,
324     3.86796474016882,
325     3.86357412779993,
326     3.85554872014731,
327     3.86044266668675,
328     3.89445961915999,
329     3.95554798115731,
330     4.02265508610476,
331     4.07419587011404,
332     4.10314247143958,
333     4.11738045085928,
334     4.12554995596708,
335     4.12923992001675,
336     4.1229292327442,
337     4.10123955307441,
338     4.06748827895363,
339     4.0336230651344,
340     4.01134236393876,
341     4.00136570034559,
342     3.99368787690411,
343     3.97820924247644,
344     3.95369335178055,
345     3.92742545608532,
346     3.90683362771686,
347     3.89331520944006,
348     3.88256045801583
349 };
350
351 std::vector<double> expected_energy_period_vec_s = {
352     10.4456008226821,
353     10.4614151137651,
354     10.4462827795433,
355     10.4127692097884,
356     10.3734397942723,
357     10.3408599227669,
358     10.32637292093,
359     10.3245412676322,
360     10.310409818185,
361     10.2589529840966,
362     10.1728100603103,
363     10.0862908658929,
364     10.03480243813,
365     10.023673635806,
366     10.0243418565116,
367     10.0063487117653,
368     9.96050302286607,
```

```
369     9.9011999635568,
370     9.84451822125472,
371     9.79726875879626,
372     9.75614594835158,
373     9.7173447961368,
374     9.68342904390577,
375     9.66380508567062,
376     9.6674009575699,
377     9.68927134575103,
378     9.70979984863046,
379     9.70967357906908,
380     9.68983025704562,
381     9.6722855524805,
382     9.67973599910003,
383     9.71977125328293,
384     9.78450442291421,
385     9.86532355233449,
386     9.96158937600019,
387     10.0807018356507,
388     10.2291022504937,
389     10.39458528356,
390     10.5464393581004,
391     10.6553277500484,
392     10.7245553190084,
393     10.7893127285064,
394     10.8846512240849,
395     11.0148158739075,
396     11.1544325654719,
397     11.2772785848343,
398     11.3744362756187,
399     11.4533643503183
400 };
401
402 for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
403     testFloatEquals(
404         test_resources.resource_map_2D[wave_resource_key][i][0],
405         expected_significant_wave_height_vec_m[i],
406         __FILE__,
407         __LINE__
408     );
409
410     testFloatEquals(
411         test_resources.resource_map_2D[wave_resource_key][i][1],
412         expected_energy_period_vec_s[i],
413         __FILE__,
414         __LINE__
415     );
416 }
417
418
419 int wind_resource_key = 3;
420 std::string path_2_wind_resource_data =
421     "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
422
423 test_resources.addResource(
424     RenewableType::WIND,
425     path_2_wind_resource_data,
426     wind_resource_key,
427     &test_electrical_load
428 );
429
430 std::vector<double> expected_wind_resource_vec_ms = {
431     6.88566688469997,
432     5.02177105466549,
433     3.74211715899568,
434     5.67169579985362,
435     4.90670669971858,
436     4.29586955031368,
437     7.41155377205065,
438     10.2243290476943,
439     13.1258696725555,
440     13.7016198628274,
441     16.2481482330233,
442     16.5096744355418,
443     13.4354482206162,
444     14.0129230731609,
445     14.5554549260515,
446     13.4454539065912,
447     13.3447169512094,
448     11.7372615098554,
449     12.7200070078013,
450     10.6421127908149,
451     6.09869498990661,
452     5.66355596602321,
453     4.97316966910831,
454     3.48937138360567,
455     2.15917470979169,
```

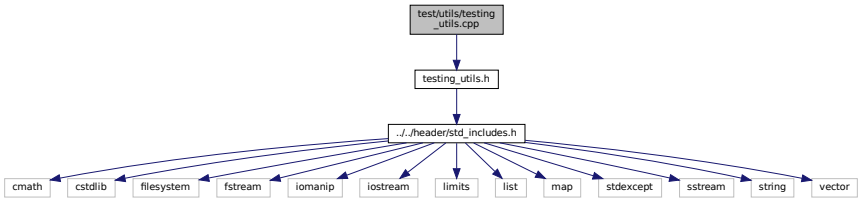
```
456     1.29061103587027,
457     3.43475751425219,
458     4.11706326260927,
459     4.28905275747408,
460     5.75850263196241,
461     8.98293663055264,
462     11.7069822941315,
463     12.4031987075858,
464     15.4096570910089,
465     16.6210843829552,
466     13.3421219142573,
467     15.2112831900548,
468     18.350864533037,
469     15.8751799822971,
470     15.3921198799796,
471     15.9729192868434,
472     12.4728950178772,
473     10.177050481096,
474     10.7342247355551,
475     8.98846695631389,
476     4.14671169124739,
477     3.17256452697149,
478     3.40036336968628
479 };
480
481 for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
482     testFloatEquals(
483         test_resources.resource_map_1D[wind_resource_key][i],
484         expected_wind_resource_vec_ms[i],
485         __FILE__,
486         __LINE__
487     );
488 }
489
490 // ======== END METHODS ============================================================ //
491
492 }   /* try */
493
494
495 catch (...) {
496     printGold(" ............................. ");
497     printRed("FAIL");
498     std::cout « std::endl;
499     throw;
500 }
501
502
503 printGold(" ............................. ");
504 printGreen("PASS");
505 std::cout « std::endl;
506 return 0;
507 }   /* main() */
```

5.48 test/utils/testing_utils.cpp File Reference

Header 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.

5.48.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.48.2 Function Documentation

5.48.2.1 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() */
```

5.48.2.2 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() */
```

5.48.2.3 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() */
```

5.48.2.4 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() */
```

5.48.2.5 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() */
```

5.48.2.6 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() */
```

5.48.2.7 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() */
```

5.48.2.8 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() */
```

5.48.2.9 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() */
```

5.48.2.10 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__").

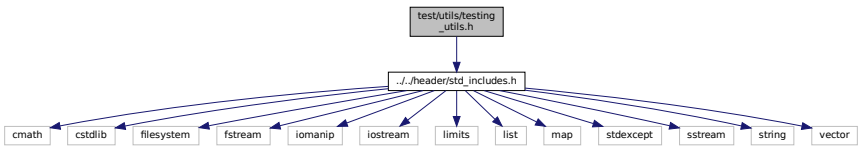
```
392 {
393     if (statement) {
394         return;
395     }
396 
397     std::string error_str = "ERROR: testTruth():\t in ";
398     error_str += file;
399     error_str += "\tline ";
400     error_str += std::to_string(line);
401     error_str += ":\t\n";
402     error_str += "Given statement 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() */
```

5.49 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.

5.49.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.49.2 Macro Definition Documentation

5.49.2.1 FLOAT_TOLERANCE

```
#define FLOAT_TOLERANCE 1e-6
```

A tolerance for application to floating point equality tests.

5.49.3 Function Documentation

5.49.3.1 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() */
```

5.49.3.2 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() */
```

5.49.3.3 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() */
```

5.49.3.4 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() */
```

5.49.3.5 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() */
```

5.49.3.6 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() */
```

5.49.3.7 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() */
```

5.49.3.8 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() */
```

5.49.3.9 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() */
```

5.49.3.10 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

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. 189

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. 55

HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 131, 176

HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 15, 131, 132, 174, 176

HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 47, 55

HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 47, 55

HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 47, 55

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. 166

HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 131, 176

HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 132, 174

HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 131, 176

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. 191, 206, 207

NRCan. Auto$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 55

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. 205

A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023. Included: docs/refs/battery_degradation.pdf. 89, 90, 92, 103

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. 190, 203, 219

Index