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

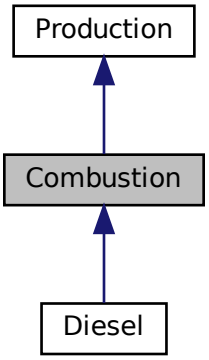
Class Documentation

4.1 Combustion Class Reference

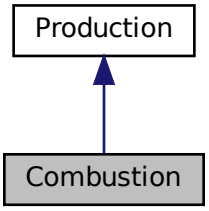
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

`#include <Combustion.h>`

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

- Combustion (void)

  Constructor (dummy) for the Combustion class.
- Combustion (int, CombustionInputs)

  Constructor (intended) for the Combustion class.
- 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.
- 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 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].
- 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 nominal 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.

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.

```
59 {
60     return;
61 }   /* Combustion() */
```

4.1.2.2 Combustion() [2/2]

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

Constructor (intended) for the Combustion class.

Parameters

n_points	The number of points in the modelling time series.
combustion_inputs	A structure of Combustion constructor inputs.

```
79                                                                                :
80 Production(n_points, combustion_inputs.production_inputs)
81 {
82     //  1. check inputs
83     this->__checkInputs(combustion_inputs);
84 
85     //  2. set attributes
86     this->fuel_cost_L = 0;
87 
88     this->linear_fuel_slope_LkWh = 0;
89     this->linear_fuel_intercept_LkWh = 0;
90 
91     this->CO2_emissions_intensity_kgL = 0;
92     this->CO_emissions_intensity_kgL = 0;
93     this->NOx_emissions_intensity_kgL = 0;
94     this->SOx_emissions_intensity_kgL = 0;
95     this->CH4_emissions_intensity_kgL = 0;
96     this->PM_emissions_intensity_kgL = 0;
97 
98     this->fuel_consumption_vec_L.resize(this->n_points, 0);
99     this->fuel_cost_vec.resize(this->n_points, 0);
100 
101     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
102     this->CO_emissions_vec_kg.resize(this->n_points, 0);
103     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
104     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
105     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
106     this->PM_emissions_vec_kg.resize(this->n_points, 0);
107 
108     //  3. construction print
109     if (this->print_flag) {
110         std::cout « "Combustion object constructed at " « this « std::endl;
111     }
112 
113     return;
114 }   /* Combustion() */
```

4.1.2.3 ~Combustion()

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

Destructor for the Combustion class.

```
252 {
253     //  1. destruction print
254     if (this->print_flag) {
255         std::cout « "Combustion object at " « this « " destroyed" « std::endl;
256     }
257 
258     return;
259 }   /* ~Combustion() */
```

4.1.3 Member Function Documentation

4.1.3.1 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.

```
150 {
151     //  1. invoke base class method
152     load_kW = Production :: commit(
153         timestep,
154         dt_hrs,
155         production_kW,
156         load_kW
157     );
158 
159 
160     if (this->is_running) {
161         //  2. compute and record fuel consumption
162         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
163         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
164 
165         //  3. compute and record emissions
166         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
167         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
168         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
169         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
170         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
171         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
172         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
173 
174         //  4. incur fuel costs
175         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
176     }
177 
178     return load_kW;
179 }   /* commit() */
```

4.1.3.2 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.

```
226                                                         {
227     Emissions emissions;
228
229     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
230     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
231     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
232     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
233     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
234     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
235
236     return emissions;
237 }   /* getEmissionskg() */
```

4.1.3.3 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].

```
201 {
202     double fuel_consumed_L = (
203         this->linear_fuel_slope_LkWh * production_kW +
204         this->linear_fuel_intercept_LkWh * this->capacity_kW
205     ) * dt_hrs;
206
207     return fuel_consumed_L;
208 } /* getFuelConsumptionL() */
```

4.1.3.4 requestProductionkW()

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

Reimplemented in Diesel.

```
117 {return 0;}
```

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 nominal 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 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.13 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.

4.1.4.14 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.15 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.

4.1.4.16 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.17 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.

4.1.4.18 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

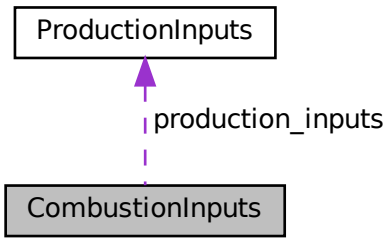
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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.

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 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 clear (void)

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

  Destructor for the Controller class.

Public Attributes

- ControlMode control_mode

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.

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

4.3.2.2 ~Controller()

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

Destructor for the Controller class.

```
73 {
74     this->clear();
75
76     return;
77 }   /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 clear()

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

Method to clear all attributes of the Controller object.

```
54 {
55     //...
56
57     return;
58 }   /* clear() */
```

4.3.4 Member Data Documentation

4.3.4.1 control_mode

```
ControlMode Controller::control_mode
```

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

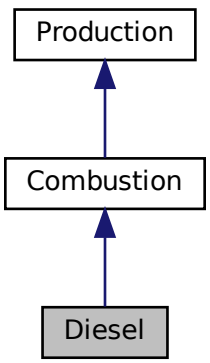
- header/Controller.h
- source/Controller.cpp

4.4 Diesel Class Reference

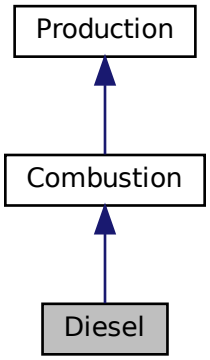
A derived class of the Combustion branch of Production which models production using a diesel generator.

`#include <Diesel.h>`

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

- Diesel (void)

  Constructor (dummy) for the Diesel class.
- Diesel (int, DieselInputs)
- 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.

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.

Constructor (intended) for the Diesel class.

Parameters

n_points	The number of points in the modelling time series.
diesel_inputs	A structure of Diesel constructor inputs.

```
305 {
306     return;
307 }   /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

```
Diesel::Diesel (
            int n_points,
            DieselInputs diesel_inputs )
325                                                 :
326 Combustion(n_points, diesel_inputs.combustion_inputs)
327 {
328     //  1. check inputs
329     this->__checkInputs(diesel_inputs);
330 
331     //  2. set attributes
332     this->type = CombustionType :: DIESEL;
333     this->type_str = "DIESEL";
334 
335     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
336 
337     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
338 
339     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
340     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
341     this->time_since_last_start_hrs = 0;
342 
343     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
344     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
345     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
346     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
347     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
348     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
349 
350     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
351         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
352 
353     }
354 
355     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
356         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
357     }
358 
359     if (diesel_inputs.capital_cost < 0) {
360         this->capital_cost = this->__getGenericCapitalCost();
361     }
362 
```

```
363     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
364         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
365     }
366
367     if (this->is_sunk) {
368         this->capital_cost_vec[0] = this->capital_cost;
369     }
370
371     //  3. construction print
372     if (this->print_flag) {
373         std::cout << "Diesel object constructed at " << this << std::endl;
374     }
375
376     return;
377 }   /* Diesel() */
```

4.4.2.3 ~Diesel()

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

Destructor for the Diesel class.

```
502 {
503     //  1. destruction print
504     if (this->print_flag) {
505         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
506     }
507
508     return;
509 }   /* ~Diesel() */
```

4.4.3 Member Function Documentation

4.4.3.1 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.

Reimplemented from Combustion.

```
460 {
461     //  1. handle start/stop, enforce minimum runtime constraint
```

```
462     this->__handleStartStop(timestep, dt_hrs, production_kW);
463
464     //  2. invoke base class method
465     load_kW = Combustion :: commit(
466         timestep,
467         dt_hrs,
468         production_kW,
469         load_kW
470     );
471
472     if (this->is_running) {
473         //  3. log time since last start
474         this->time_since_last_start_hrs += dt_hrs;
475
476         //  4. correct operation and maintenance costs (should be non-zero if idling)
477         if (production_kW <= 0) {
478             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
479
480             double operation_maintenance_cost =
481                 this->operation_maintenance_cost_kWh * produced_kWh;
482             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
483         }
484     }
485
486     return load_kW;
487 }   /* commit() */
```

4.4.3.2 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].

Reimplemented from Combustion.

```
407 {
408     //  1. return on request of zero
409     if (request_kW <= 0) {
410         return 0;
411     }
412
413     double deliver_kW = request_kW;
414
415     //  2. enforce capacity constraint
416     if (deliver_kW > this->capacity_kW) {
417         deliver_kW = this->capacity_kW;
418     }
419
420     //  3. enforce minimum load ratio
421     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
422         deliver_kW = this->minimum_load_ratio * this->capacity_kW;
423     }
424
425     return deliver_kW;
426 }   /* 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:

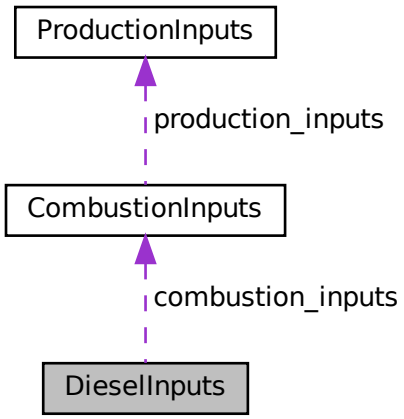
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

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.

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:

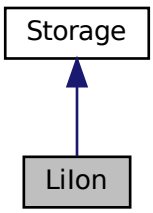
- header/Production/Combustion/Combustion.h

4.8 LiIon Class Reference

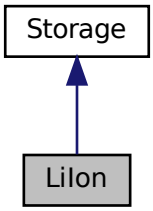
A derived class of Storage which models energy storage by way of lithium-ion batteries.

```
#include <LiIon.h>
```

Inheritance diagram for LiIon:



Collaboration diagram for LiIon:



Public Member Functions

- LiIon (void)

  Constructor for the LiIon class.
- ~LiIon (void)

  Destructor for the LiIon class.

4.8.1 Detailed Description

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

4.8.2 Constructor & Destructor Documentation

4.8.2.1 LiIon()

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

Constructor for the LiIon class.

```
35                    :
36 Storage()
37 {
38     //...
39
40     return;
41 }   /* LiIon() */
```

4.8.2.2 ~LiIon()

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

Destructor for the LiIon class.

```
64 {
65     //...
66
67     return;
68 }   /* ~LiIon() */
```

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

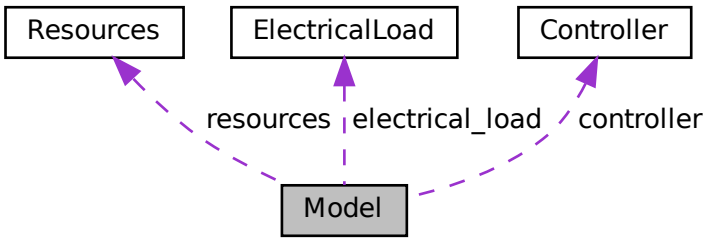
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

4.9 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 addResource (RenewableType, std::string, int)

  A method to add a renewable resource time series to 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; it leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.
- void clear (void)

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

  Destructor for the Model class.

Public Attributes

- 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.

4.9.1 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.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 Model() [1/2]

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

Constructor (dummy) for the Model class.

```
55 {
56     return;
57 }   /* Model() */
```

4.9.2.2 Model() [2/2]

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

Constructor (intended) for the Model class.

Parameters


```
74 {
75     //  1. check inputs
76     this->__checkInputs(model_inputs);
```

```
77
78      //  2. read in electrical load data
79      this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
80
81      //  3. set control mode
82      this->controller.control_mode = model_inputs.control_mode;
83
84      return;
85  }   /* Model() */
```

4.9.2.3 ~Model()

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

Destructor for the Model class.

```
212 {
213     this->clear();
214     return;
215 }   /* ~Model() */
```

4.9.3 Member Function Documentation

4.9.3.1 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.

```
113 {
114     switch (renewable_type) {
115         case (RenewableType :: WAVE): {
116             resources.addResource2D(
117                 renewable_type,
118                 path_2_resource_data,
119                 resource_key,
120                 &(this->electrical_load)
121             );
122
123             break;
124         }
125
126         default: {
127             resources.addResource1D(
128                 renewable_type,
129                 path_2_resource_data,
130                 resource_key,
```

```
131                 &(this->electrical_load)
132             );
133
134             break;
135         }
136     }
137
138     return;
139 }   /* addResource() */
```

4.9.3.2 clear()

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

Method to clear all attributes of the Model object.

```
188 {
189     //  1. reset
190     this->reset();
191
192     //  2. clear remaining attributes
193     controller.clear();
194     electrical_load.clear();
195     resources.clear();
196
197     return;
198 }   /* clear() */
```

4.9.3.3 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; it leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

```
154 {
155     //  1. clear combustion_ptr_vec
156     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
157         delete this->combustion_ptr_vec[i];
158     }
159     this->combustion_ptr_vec.clear();
160
161     //  2. clear renewable_ptr_vec
162     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
163         delete this->renewable_ptr_vec[i];
164     }
165     this->renewable_ptr_vec.clear();
166
167     //  3. clear storage_ptr_vec
168     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
169         delete this->storage_ptr_vec[i];
170     }
171     this->storage_ptr_vec.clear();
172
173     return;
174 }   /* reset() */
```

4.9.4 Member Data Documentation

4.9.4.1 combustion_ptr_vec

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

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

4.9.4.2 controller

```
Controller Model::controller
```

Controller component of Model.

4.9.4.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

ElectricalLoad component of Model.

4.9.4.4 renewable_ptr_vec

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

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

4.9.4.5 resources

```
Resources Model::resources
```

Resources component of Model.

4.9.4.6 storage_ptr_vec

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

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

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

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

4.10 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.

4.10.1 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).

4.10.2 Member Data Documentation

4.10.2.1 control_mode

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

The control mode to be applied by the Controller object.

4.10.2.2 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:

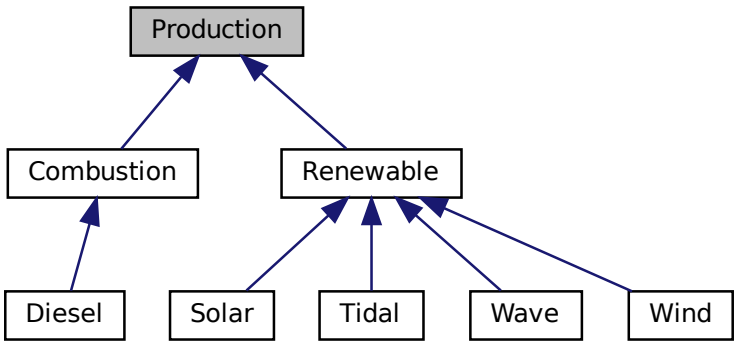
- header/Model.h

4.11 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:



Public Member Functions

- Production (void)

  Constructor (dummy) for the Production class.
- Production (int, ProductionInputs)

  Constructor (intended) for the Production class.
- 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

- 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 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 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 levellized_cost_of_energy_kWh

  The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatched and stored energy.
- 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 nominal 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 nominal costs).

4.11.1 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.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 Production() [1/2]

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

Constructor (dummy) for the Production class.

```
143 {
144     return;
145 }   /* Production() */
```

4.11.2.2 Production() [2/2]

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

Constructor (intended) for the Production class.

Parameters

n_points	The number of points in the modelling time series.
production_inputs	A structure of Production constructor inputs.

```
167 {
168     //  1. check inputs
169     this->__checkInputs(n_points, production_inputs);
170 
171     //  2. set attributes
172     this->print_flag = production_inputs.print_flag;
173     this->is_running = false;
174 
175     this->n_points = n_points;
176     this->n_starts = 0;
177 
178     this->running_hours = 0;
179     this->replace_running_hrs = production_inputs.replace_running_hrs;
180 
181     this->capacity_kW = production_inputs.capacity_kW;
182 
183     this->real_discount_annual = this->__computeRealDiscountAnnual(
184         production_inputs.nominal_inflation_annual,
185         production_inputs.nominal_discount_annual
186     );
187     this->capital_cost = 0;
188     this->operation_maintenance_cost_kWh = 0;
189     this->net_present_cost = 0;
190     this->levellized_cost_of_energy_kWh = 0;
191 
192     this->is_running_vec.resize(this->n_points, 0);
193 
194     this->production_vec_kW.resize(this->n_points, 0);
195     this->dispatch_vec_kW.resize(this->n_points, 0);
196     this->storage_vec_kW.resize(this->n_points, 0);
197     this->curtailment_vec_kW.resize(this->n_points, 0);
198 
199     this->capital_cost_vec.resize(this->n_points, 0);
200     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
201 
202     //  3. construction print
203     if (this->print_flag) {
204         std::cout << "Production object constructed at " << this << std::endl;
205     }
206 
207     return;
208 }   /* Production() */
```

4.11.2.3 ~Production()

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

Destructor for the Production class.

```
300 {
301     //  1. destruction print
302     if (this->print_flag) {
303         std::cout « "Production object at " « this « " destroyed" « std::endl;
304     }
305
306     return;
307 }   /* ~Production() */
```

4.11.3 Member Function Documentation

4.11.3.1 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.

```
244 {
245     //  1. record production
246     this->production_vec_kW[timestep] = production_kW;
247
248     //  2. compute and record dispatch and curtailment
249     double dispatch_kW = 0;
250     double curtailment_kW = 0;
251
252     if (production_kW > load_kW) {
253         dispatch_kW = load_kW;
254         curtailment_kW = production_kW - dispatch_kW;
255     }
256
257     else {
258         dispatch_kW = production_kW;
259     }
260
```

```
261     this->dispatch_vec_kW[timestep] = dispatch_kW;
262     this->curtailment_vec_kW[timestep] = curtailment_kW;
263
264     //  3. update load
265     load_kW -= dispatch_kW;
266
267     if (this->is_running) {
268         //  4. log running state, running hours
269         this->is_running_vec[timestep] = this->is_running;
270         this->running_hours += dt_hrs;
271
272         //  5. incur operation and maintenance costs
273         double produced_kWh = production_kW * dt_hrs;
274
275         double operation_maintenance_cost =
276             this->operation_maintenance_cost_kWh * produced_kWh;
277         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
278
279         //  6. incur capital costs (i.e., handle replacement)
280         this->__handleReplacement(timestep);
281     }
282
283
284     return load_kW;
285 }   /* commit() */
```

4.11.4 Member Data Documentation

4.11.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.11.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.11.4.3 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 nominal costs).

4.11.4.4 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.

4.11.4.5 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.

4.11.4.6 is_running

```
bool Production::is_running
```

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

4.11.4.7 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.

4.11.4.8 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).

4.11.4.9 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 dispatched and stored energy.

4.11.4.10 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.11.4.11 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.11.4.12 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.11.4.13 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.11.4.14 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.

4.11.4.15 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 nominal costs).

4.11.4.16 print_flag

```
bool Production::print_flag
```

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

4.11.4.17 production_vec_kW

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

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

4.11.4.18 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.

4.11.4.19 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.11.4.20 running_hours

```
double Production::running_hours
```

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

4.11.4.21 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.

4.11.4.22 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

4.12 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.

4.12.1 Detailed Description

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

4.12.2 Member Data Documentation

4.12.2.1 capacity_kW

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

The rated production capacity [kW] of the asset.

4.12.2.2 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).

4.12.2.3 nominal_discount_annual

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

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

4.12.2.4 nominal_inflation_annual

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

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

4.12.2.5 print_flag

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

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

4.12.2.6 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:

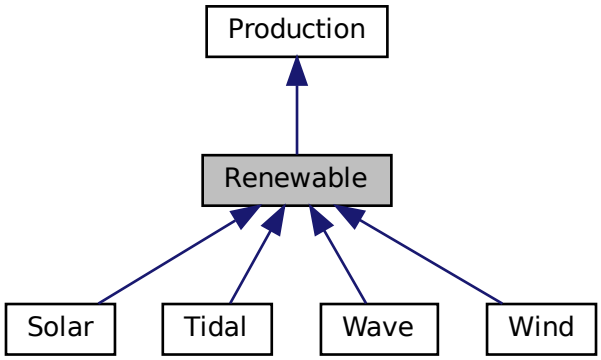
- header/Production/Production.h

4.13 Renewable Class Reference

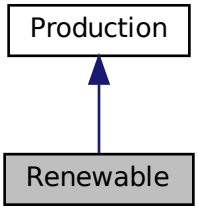
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

`#include <Renewable.h>`

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



Public Member Functions

- Renewable (void)

  Constructor (dummy) for the Renewable class.
- Renewable (int, RenewableInputs)
- 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.
- 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.

4.13.1 Detailed Description

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

4.13.2 Constructor & Destructor Documentation

4.13.2.1 Renewable() [1/2]

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

Constructor (dummy) for the Renewable class.

Constructor (intended) for the Renewable class.

Parameters

n_points	The number of points in the modelling time series.
renewable_inputs	A structure of Renewable constructor inputs.

```
88 {
89     //...
90
91     return;
92 }   /* Renewable() */
```

4.13.2.2 Renewable() [2/2]

```
Renewable::Renewable (
            int n_points,
            RenewableInputs renewable_inputs )
110                                                                    :
111 Production(n_points, renewable_inputs.production_inputs)
112 {
113     //  1. check inputs
114     this->__checkInputs(renewable_inputs);
115
116     //  2. set attributes
117     //...
118
119     //  3. construction print
120     if (this->print_flag) {
```

```
121         std::cout « "Renewable object constructed at " « this « std::endl;
122     }
123
124     return;
125 }   /* Renewable() */
```

4.13.2.3 ~Renewable()

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

Destructor for the Renewable class.

```
192 {
193     //  1. destruction print
194     if (this->print_flag) {
195         std::cout « "Renewable object at " « this « " destroyed" « std::endl;
196     }
197
198     return;
199 }   /* ~Renewable() */
```

4.13.3 Member Function Documentation

4.13.3.1 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.

```
161 {
162     //  1. handle start/stop
```

```
163     this->__handleStartStop(timestep, dt_hrs, production_kW);
164
165     //  2. invoke base class method
166     load_kW = Production :: commit(
167         timestep,
168         dt_hrs,
169         production_kW,
170         load_kW
171     );
172
173
174     //...
175
176     return load_kW;
177 }   /* commit() */
```

4.13.3.2 computeProductionkW() [1/2]

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

Reimplemented in Wind, Tidal, and Solar.

```
84 {return 0;}
```

4.13.3.3 computeProductionkW() [2/2]

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

Reimplemented in Wave.

```
85 {return 0;}
```

4.13.4 Member Data Documentation

4.13.4.1 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.

4.13.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

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

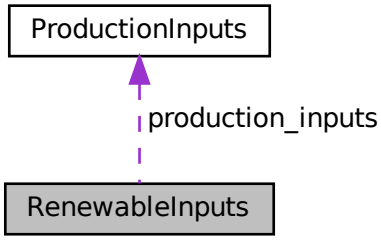
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

4.14 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.

4.14.1 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.

4.14.2 Member Data Documentation

4.14.2.1 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

4.15 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 addResource1D (RenewableType, std::string, int, ElectricalLoad *)

  A method to add a 1D renewable resource time series to Resources. Checks 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 addResource2D (RenewableType, std::string, int, ElectricalLoad *)

  A method to add a 2D renewable resource time series to Resources. Checks 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
- std::map< int, std::string > path_map_1D
- std::map< int, std::vector< std::vector< double > > > resource_map_2D
- std::map< int, std::string > path_map_2D

4.15.1 Detailed Description

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

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Resources()

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

Constructor for the Resources class.

```
93 {
94     return;
95 }   /* Resources() */
```

4.15.2.2 ~Resources()

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

Destructor for the Resources class.

```
216 {
217     this->clear();
218     return;
219 }   /* ~Resources() */
```

4.15.3 Member Function Documentation

4.15.3.1 addResource1D()

```
void Resources::addResource1D (
            RenewableType renewable_type,
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )
```

A method to add a 1D renewable resource time series to Resources. Checks 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.

```
131 {
132     //  1. check resource key
133     this->__checkResourceKey1D(resource_key);
134 
135     //...
136 
137     return;
138 }   /* addResource1D() */
```

4.15.3.2 addResource2D()

```
void Resources::addResource2D (
            RenewableType renewable_type,
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )
```

A method to add a 2D renewable resource time series to Resources. Checks 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.

```
173 {
174     //  1. check resource key
175     this->__checkResourceKey2D(resource_key);
176 
177     //...
178 
179     return;
180 }   /* addResource2D() */
```

4.15.3.3 clear()

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

Method to clear all attributes of the Resources object.

```
194 {
195     this->resource_map_1D.clear();
196     this->path_map_1D.clear();
197 
198     this->resource_map_2D.clear();
199     this->path_map_2D.clear();
200 
201     return;
202 }   /* clear() */
```

4.15.4 Member Data Documentation

4.15.4.1 path_map_1D

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

4.15.4.2 path_map_2D

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

4.15.4.3 resource_map_1D

```
std::map<int, std::vector<double> > Resources::resource_map_1D
```

4.15.4.4 resource_map_2D

```
std::map<int, std::vector<std::vector<double> > > Resources::resource_map_2D
```

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

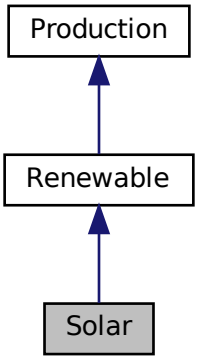
- header/Resources.h
- source/Resources.cpp

4.16 Solar Class Reference

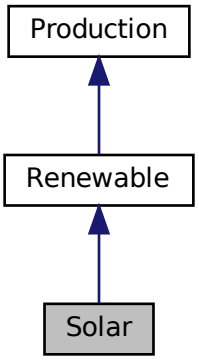
A derived class of the Renewable branch of Production which models solar production.

```
#include <Solar.h>
```

Inheritance diagram for Solar:



Collaboration diagram for Solar:



Public Member Functions

- Solar (void)

  Constructor (dummy) for the Solar class.
- Solar (int, SolarInputs)
- 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.).

4.16.1 Detailed Description

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

4.16.2 Constructor & Destructor Documentation

4.16.2.1 Solar() [1/2]

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

Constructor (dummy) for the Solar class.

Constructor (intended) for the Solar class.

Parameters

n_points	The number of points in the modelling time series.
solar_inputs	A structure of Solar constructor inputs.

```
113 {
114     //...
115 
116     return;
117 }   /* Solar() */
```

4.16.2.2 Solar() [2/2]

```
Solar::Solar (
             int n_points,
             SolarInputs solar_inputs )
135                                                 :
136 Renewable(n_points, solar_inputs.renewable_inputs)
137 {
138     //  1. check inputs
139     this->__checkInputs(solar_inputs);
140 
141     //  2. set attributes
142     this->type = RenewableType :: SOLAR;
143     this->type_str = "SOLAR";
144 
145     this->resource_key = solar_inputs.resource_key;
146 
147     this->derating = solar_inputs.derating;
148 
149     if (solar_inputs.capital_cost < 0) {
150         this->capital_cost = this->__getGenericCapitalCost();
151     }
152 
153     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
154         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
155     }
156 
157     if (this->is_sunk) {
158         this->capital_cost_vec[0] = this->capital_cost;
159     }
160 
161     //  3. construction print
162     if (this->print_flag) {
163         std::cout << "Solar object constructed at " << this << std::endl;
164     }
165 
166     return;
167 }   /* Renewable() */
```

4.16.2.3 ~Solar()

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

Destructor for the Solar class.

```
280 {
281     //  1. destruction print
282     if (this->print_flag) {
283         std::cout << "Solar object at " << this << " destroyed" << std::endl;
284     }
285 
286     return;
287 }   /* ~Solar() */
```

4.16.3 Member Function Documentation

4.16.3.1 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.

```
252 {
253     //  1. invoke base class method
254     load_kW = Renewable :: commit(
255         timestep,
256         dt_hrs,
257         production_kW,
258         load_kW
259     );
260 
261 
262     //...
263 
264     return load_kW;
265 }   /* commit() */
```

4.16.3.2 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: https://www.homerenergy.com/products/pro/docs/3.11/how_homer_calculates_the_pv_array_power_output.html

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.

```
201 {
202     // check if no resource
203     if (solar_resource_kWm2 <= 0) {
204         return 0;
205     }
206 
207     // compute production
208     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
209 
210     // cap production at capacity
211     if (production_kW > this->capacity_kW) {
212         production_kW = this->capacity_kW;
213     }
214 
215     return production_kW;
216 }   /* computeProductionkW() */
```

4.16.4 Member Data Documentation

4.16.4.1 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:

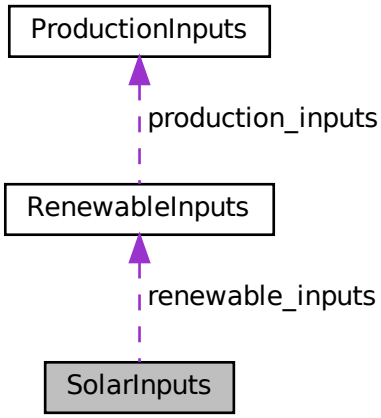
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

4.17 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.).

4.17.1 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.

4.17.2 Member Data Documentation

4.17.2.1 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].

4.17.2.2 derating

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

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

4.17.2.3 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].

4.17.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.17.2.5 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:

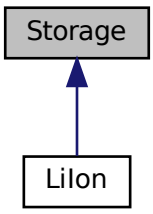
- header/Production/Renewable/Solar.h

4.18 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:



Public Member Functions

- Storage (void)

  Constructor for the Storage class.
- virtual ~Storage (void)

  Destructor for the Storage class.

4.18.1 Detailed Description

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

4.18.2 Constructor & Destructor Documentation

4.18.2.1 Storage()

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

Constructor for the Storage class.

```
36 {
37     //...
38
39     return;
40 }   /* Storage() */
```

4.18.2.2 ~Storage()

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

Destructor for the Storage class.

```
63 {
64     //...
65
66     return;
67 }   /* ~Storage() */
```

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

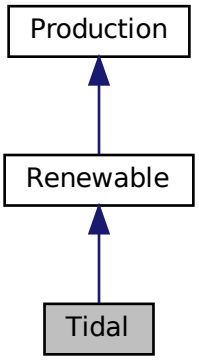
- header/Storage/Storage.h
- source/Storage/Storage.cpp

4.19 Tidal Class Reference

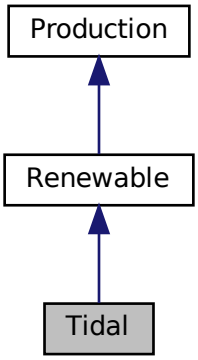
A derived class of the Renewable branch of Production which models tidal production.

```
#include <Tidal.h>
```

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- Tidal (void)

  Constructor (dummy) for the Tidal class.
- Tidal (int, TidalInputs)
- 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.

4.19.1 Detailed Description

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

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Tidal() [1/2]

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

Constructor (dummy) for the Tidal class.

Constructor (intended) for the Tidal class.

Parameters

n_points	The number of points in the modelling time series.
tidal_inputs	A structure of Tidal constructor inputs.

```
226 {
227     return;
228 }   /* Tidal() */
```

4.19.2.2 Tidal() [2/2]

```
Tidal::Tidal (
            int n_points,
            TidalInputs tidal_inputs )
246                                                 :
247 Renewable(n_points, tidal_inputs.renewable_inputs)
248 {
249     //  1. check inputs
250     this->__checkInputs(tidal_inputs);
251 
252     //  2. set attributes
253     this->type = RenewableType :: TIDAL;
254     this->type_str = "TIDAL";
255 
256     this->resource_key = tidal_inputs.resource_key;
257 
258     this->design_speed_ms = tidal_inputs.design_speed_ms;
259 
260     this->power_model = tidal_inputs.power_model;
261 
262     if (tidal_inputs.capital_cost < 0) {
263         this->capital_cost = this->__getGenericCapitalCost();
264     }
265 
266     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
267         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
268     }
269 
270     if (this->is_sunk) {
271         this->capital_cost_vec[0] = this->capital_cost;
272     }
273 
274     //  3. construction print
275     if (this->print_flag) {
276         std::cout « "Tidal object constructed at " « this « std::endl;
277     }
278 
279     return;
280 }   /* Renewable() */
```

4.19.2.3 ~Tidal()

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

Destructor for the Tidal class.

```
418 {
419     //  1. destruction print
420     if (this->print_flag) {
421         std::cout « "Tidal object at " « this « " destroyed" « std::endl;
422     }
423 
424     return;
425 }   /* ~Tidal() */
```

4.19.3 Member Function Documentation

4.19.3.1 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.

```
390 {
391     //  1. invoke base class method
392     load_kW = Renewable :: commit(
393         timestep,
394         dt_hrs,
395         production_kW,
396         load_kW
397     );
398 
399 
400     //...
401 
402     return load_kW;
403 }   /* commit() */
```

4.19.3.2 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.

```
312 {
313     // check if no resource
314     if (tidal_resource_ms <= 0) {
315         return 0;
316     }
317 
318     // compute production
319     double production_kW = 0;
320 
321     switch (this->power_model) {
322         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
323             production_kW = this->__computeExponentialProductionkW(
324                 timestep,
325                 dt_hrs,
326                 tidal_resource_ms
327             );
328 
329             break;
330         }
331 
332         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
333             production_kW = this->__computeLookupProductionkW(
334                 timestep,
335                 dt_hrs,
336                 tidal_resource_ms
337             );
338 
339             break;
340         }
341 
342         default: {   // default to CUBIC
343             production_kW = this->__computeCubicProductionkW(
344                 timestep,
345                 dt_hrs,
346                 tidal_resource_ms
347             );
348 
349             break;
350         }
351     }
352 
353     return production_kW;
354 }   /* computeProductionkW() */
```

4.19.4 Member Data Documentation

4.19.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

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

4.19.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

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

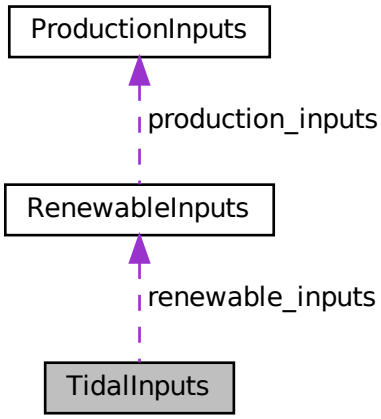
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.20 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.

4.20.1 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.

4.20.2 Member Data Documentation

4.20.2.1 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].

4.20.2.2 design_speed_ms

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

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

4.20.2.3 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].

4.20.2.4 power_model

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

The tidal power production model to be applied.

4.20.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.20.2.6 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:

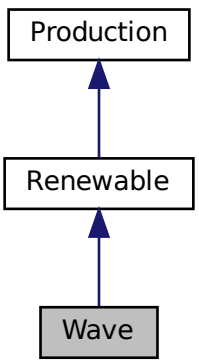
- header/Production/Renewable/Tidal.h

4.21 Wave Class Reference

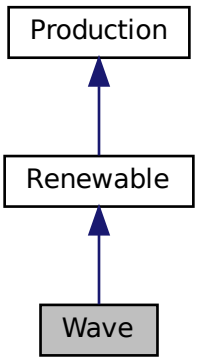
A derived class of the Renewable branch of Production which models wave production.

```
#include <Wave.h>
```

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

- Wave (void)

  Constructor (dummy) for the Wave class.
- Wave (int, WaveInputs)
- 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.

4.21.1 Detailed Description

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

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Wave() [1/2]

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

Constructor (dummy) for the Wave class.

Constructor (intended) for the Wave class.

Parameters

n_points	The number of points in the modelling time series.
wave_inputs	A structure of Wave constructor inputs.

```
237 {
238     return;
239 }   /* Wave() */
```

4.21.2.2 Wave() [2/2]

```
Wave::Wave (
            int n_points,
            WaveInputs wave_inputs )
257                                                 :
258 Renewable(n_points, wave_inputs.renewable_inputs)
259 {
260     //  1. check inputs
261     this->__checkInputs(wave_inputs);
262 
263     //  2. set attributes
264     this->type = RenewableType :: WAVE;
265     this->type_str = "WAVE";
266 
267     this->resource_key = wave_inputs.resource_key;
268 
269     this->design_significant_wave_height_m =
270         wave_inputs.design_significant_wave_height_m;
271     this->design_energy_period_s = wave_inputs.design_energy_period_s;
272 
273     this->power_model = wave_inputs.power_model;
274 
275     if (wave_inputs.capital_cost < 0) {
276         this->capital_cost = this->__getGenericCapitalCost();
277     }
278 
279     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
280         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
281     }
282 
283     if (this->is_sunk) {
284         this->capital_cost_vec[0] = this->capital_cost;
285     }
286 
287     //  3. construction print
288     if (this->print_flag) {
289         std::cout « "Wave object constructed at " « this « std::endl;
290     }
291 
292     return;
293 }   /* Renewable() */
```

4.21.2.3 ~Wave()

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

Destructor for the Wave class.

```
438 {
439     //  1. destruction print
440     if (this->print_flag) {
441         std::cout « "Wave object at " « this « " destroyed" « std::endl;
442     }
443
444     return;
445 }   /* ~Wave() */
```

4.21.3 Member Function Documentation

4.21.3.1 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.

```
410 {
411     //  1. invoke base class method
412     load_kW = Renewable :: commit(
413         timestep,
414         dt_hrs,
415         production_kW,
416         load_kW
417     );
418
419
420     //...
421
422     return load_kW;
423 }   /* commit() */
```

4.21.3.2 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.

```
329 {
330     // check if no resource
331     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
332         return 0;
333     }
334 
335     // compute production
336     double production_kW = 0;
337 
338     switch (this->power_model) {
339         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
340             production_kW = this->__computeGaussianProductionkW(
341                 timestep,
342                 dt_hrs,
343                 significant_wave_height_m,
344                 energy_period_s
345             );
346 
347             break;
348         }
349 
350         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
351             production_kW = this->__computeLookupProductionkW(
352                 timestep,
353                 dt_hrs,
354                 significant_wave_height_m,
355                 energy_period_s
356             );
357 
358             break;
359         }
360 
361         default: {   // default to PARABOLOID
362             production_kW = this->__computeParaboloidProductionkW(
363                 timestep,
364                 dt_hrs,
365                 significant_wave_height_m,
366                 energy_period_s
367             );
368 
369             break;
370         }
371     }
372 
373     return production_kW;
374 }   /* computeProductionkW() */
```

4.21.4 Member Data Documentation

4.21.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

4.21.4.2 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.

4.21.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

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

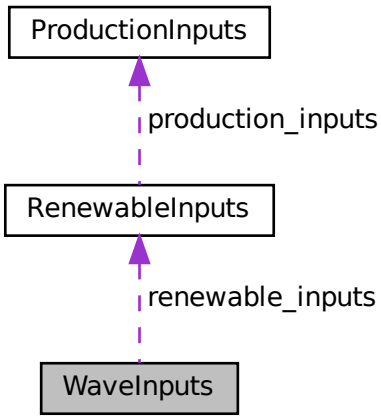
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

4.22 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.

4.22.1 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.

4.22.2 Member Data Documentation

4.22.2.1 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].

4.22.2.2 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.

4.22.2.3 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.

4.22.2.4 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].

4.22.2.5 power_model

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

The wave power production model to be applied.

4.22.2.6 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.22.2.7 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:

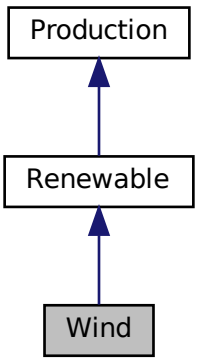
- header/Production/Renewable/Wave.h

4.23 Wind Class Reference

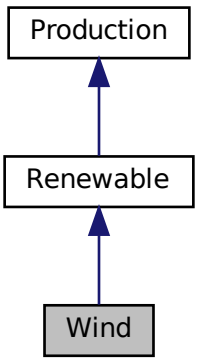
A derived class of the Renewable branch of Production which models wind production.

`#include <Wind.h>`

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- Wind (void)

  Constructor (dummy) for the Wind class.
- Wind (int, WindInputs)

- 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.

4.23.1 Detailed Description

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

4.23.2 Constructor & Destructor Documentation

4.23.2.1 Wind() [1/2]

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

Constructor (dummy) for the Wind class.

Constructor (intended) for the Wind class.

Parameters

n_points	The number of points in the modelling time series.
wind_inputs	A structure of Wind constructor inputs.

```
172 {
173     return;
174 }   /* Wind() */
```

4.23.2.2 Wind() [2/2]

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

```
192                                                 :
193 Renewable(n_points, wind_inputs.renewable_inputs)
194 {
195     //  1. check inputs
196     this->__checkInputs(wind_inputs);
197 
198     //  2. set attributes
199     this->type = RenewableType :: WIND;
200     this->type_str = "WIND";
201 
202     this->resource_key = wind_inputs.resource_key;
203 
204     this->design_speed_ms = wind_inputs.design_speed_ms;
205 
206     this->power_model = wind_inputs.power_model;
207 
208     if (wind_inputs.capital_cost < 0) {
209         this->capital_cost = this->__getGenericCapitalCost();
210     }
211 
212     if (wind_inputs.operation_maintenance_cost_kWh < 0) {
213         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
214     }
215 
216     if (this->is_sunk) {
217         this->capital_cost_vec[0] = this->capital_cost;
218     }
219 
220     //  3. construction print
221     if (this->print_flag) {
222         std::cout « "Wind object constructed at " « this « std::endl;
223     }
224 
225     return;
226 }   /* Renewable() */
```

4.23.2.3 ~Wind()

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

Destructor for the Wind class.

```
354 {
355     //  1. destruction print
356     if (this->print_flag) {
357         std::cout « "Wind object at " « this « " destroyed" « std::endl;
358     }
359 
360     return;
361 }   /* ~Wind() */
```

4.23.3 Member Function Documentation

4.23.3.1 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.

```
326 {
327     //  1. invoke base class method
328     load_kW = Renewable :: commit(
329         timestep,
330         dt_hrs,
331         production_kW,
332         load_kW
333     );
334 
335 
336     //...
337 
338     return load_kW;
339 }   /* commit() */
```

4.23.3.2 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.

```
258 {
259     // check if no resource
260     if (wind_resource_ms <= 0) {
261         return 0;
262     }
263 
264     // compute production
265     double production_kW = 0;
```

```
266
267     switch (this->power_model) {
268         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
269             production_kW = this->__computeLookupProductionkW(
270                 timestep,
271                 dt_hrs,
272                 wind_resource_ms
273             );
274
275             break;
276         }
277
278         default: {   // default to WIND_POWER_EXPONENTIAL
279             production_kW = this->__computeExponentialProductionkW(
280                 timestep,
281                 dt_hrs,
282                 wind_resource_ms
283             );
284
285             break;
286         }
287     }
288
289     return production_kW;
290 }   /* computeProductionkW() */
```

4.23.4 Member Data Documentation

4.23.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

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

4.23.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

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

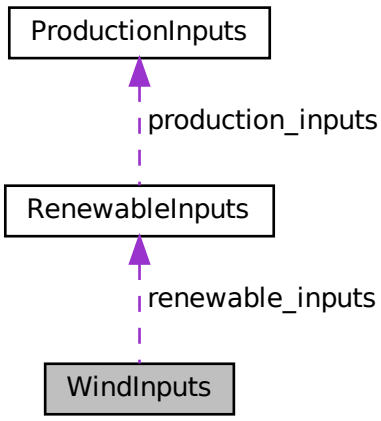
- header/Production/Renewable/Wind.h
- source/Production/Renewable/Wind.cpp

4.24 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.

4.24.1 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.

4.24.2 Member Data Documentation

4.24.2.1 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].

4.24.2.2 design_speed_ms

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

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

4.24.2.3 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].

4.24.2.4 power_model

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

The wind power production model to be applied.

4.24.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.24.2.6 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

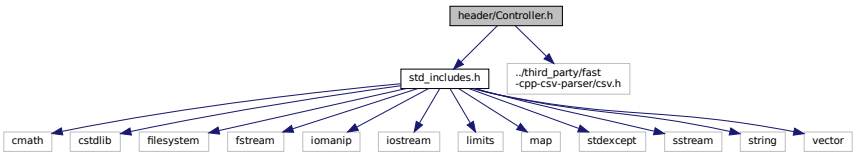
File Documentation

5.1 header/Controller.h File Reference

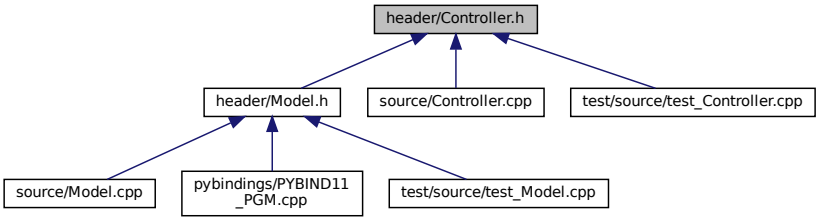
Header file the Controller class.

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

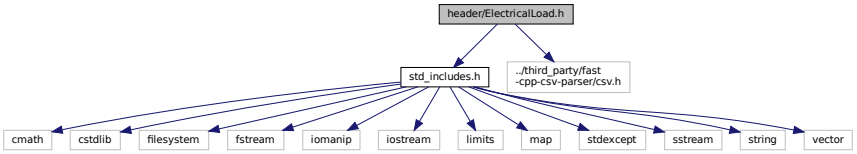
```
34                  {
35     LOAD_FOLLOWING,
36     CYCLE_CHARGING,
37     N_CONTROL_MODES
38 };
```

5.2 header/ElectricalLoad.h File Reference

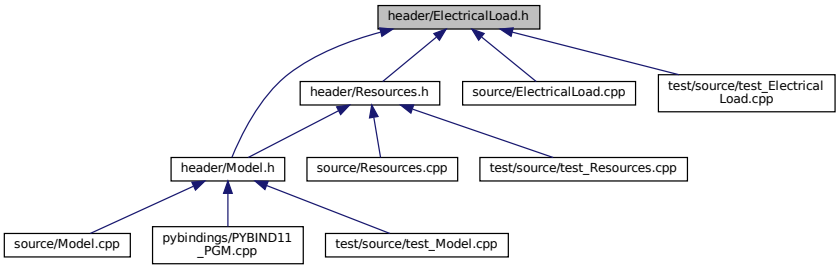
Header file the ElectricalLoad class.

```
#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

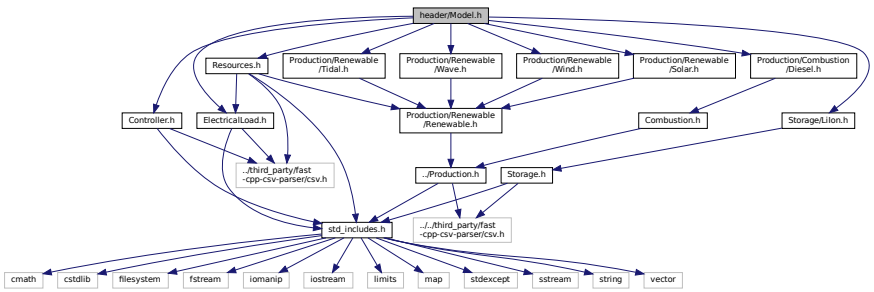
Header file the ElectricalLoad class.

5.3 header/Model.h File Reference

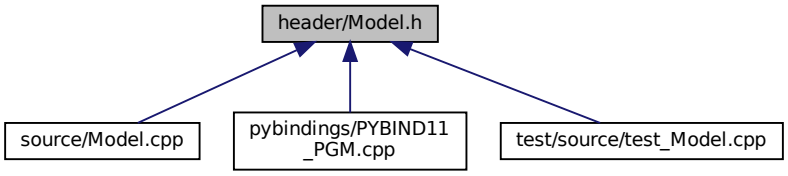
Header file the Model class.

```
#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.

5.3.1 Detailed Description

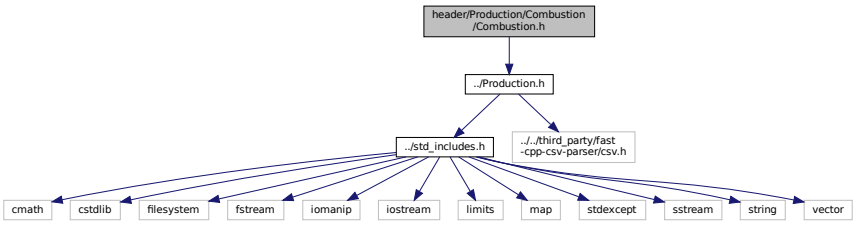
Header file the Model class.

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

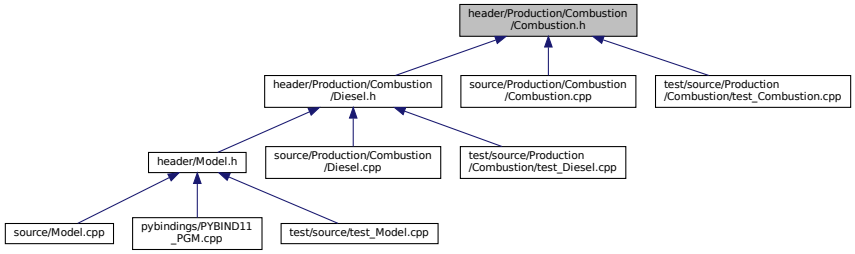
Header file the Combustion class.

```
#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.

5.4.1 Detailed Description

Header file the Combustion class.

5.4.2 Enumeration Type Documentation

5.4.2.1 CombustionType

`enum CombustionType`

An enumeration of the types of Combustion asset supported by PGMcpp.

Enumerator

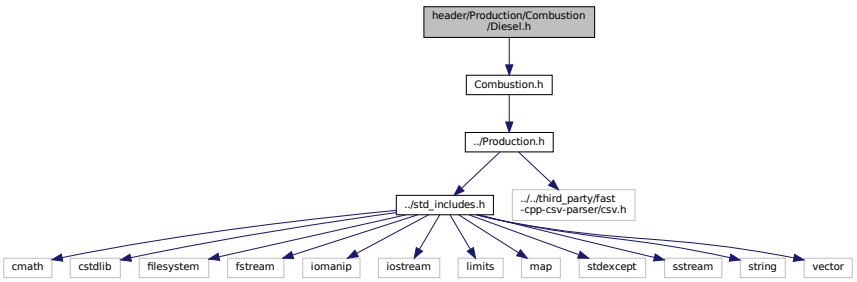
DIESEL	A diesel generator.
N_COMBUSTION_TYPES	A simple hack to get the number of elements in CombustionType.

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

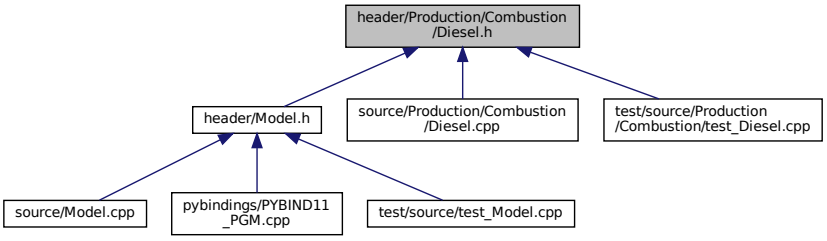
5.5 header/Production/Combustion/Diesel.h File Reference

Header file 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.

5.5.1 Detailed Description

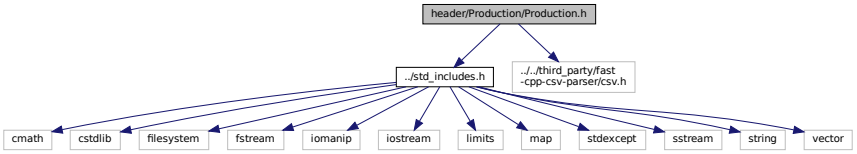
Header file the Diesel class.

5.6 header/Production/Production.h File Reference

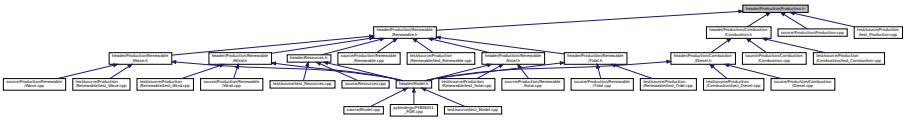
Header file the Production class.

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

5.6.1 Detailed Description

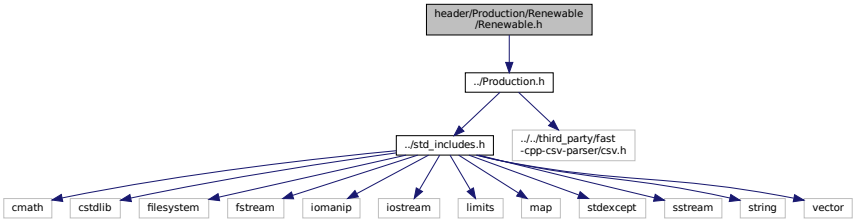
Header file the Production class.

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

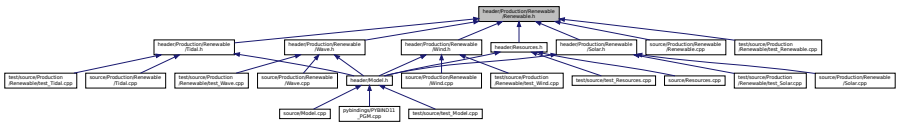
Header file the Renewable class.

```
#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.

5.7.1 Detailed Description

Header file the Renewable class.

5.7.2 Enumeration Type Documentation

5.7.2.1 RenewableType

```
enum RenewableType
```

An enumeration of the types of Renewable asset supported by PGMcpp.

Enumerator

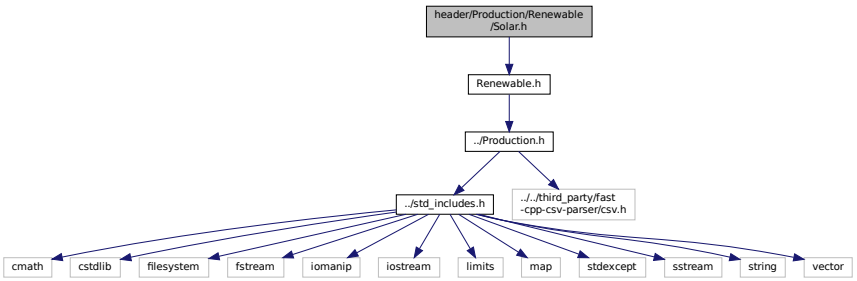
SOLAR	A solar photovoltaic (PV) array.
TIDAL	A tidal stream turbine (or tidal energy converter, TEC)
WAVE	A wave energy converter (WEC)
WIND	A wind turbine.
N_RENEWABLE_TYPES	A simple hack to get the number of elements in RenewableType.

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

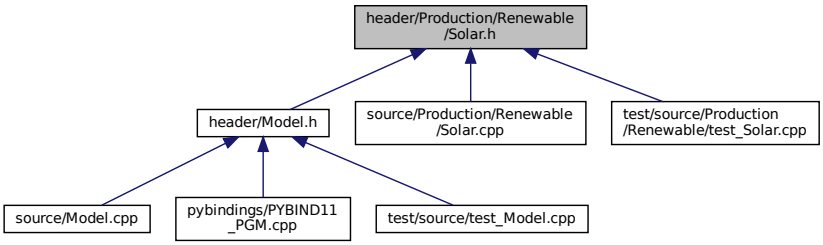
5.8 header/Production/Renewable/Solar.h File Reference

Header file 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.

5.8.1 Detailed Description

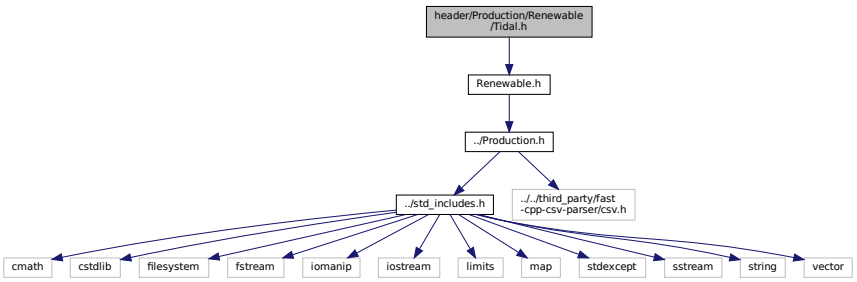
Header file the Solar class.

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

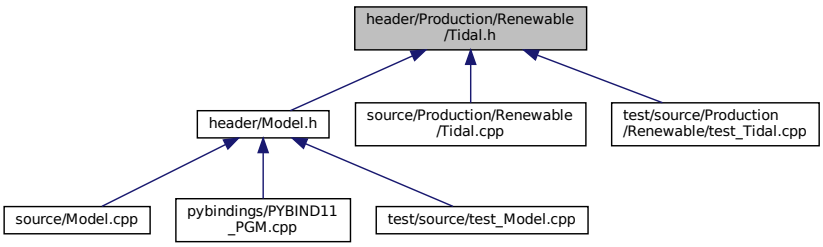
Header file the Tidal class.

```
#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 }

5.9.1 Detailed Description

Header file the Tidal class.

5.9.2 Enumeration Type Documentation

5.9.2.1 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.

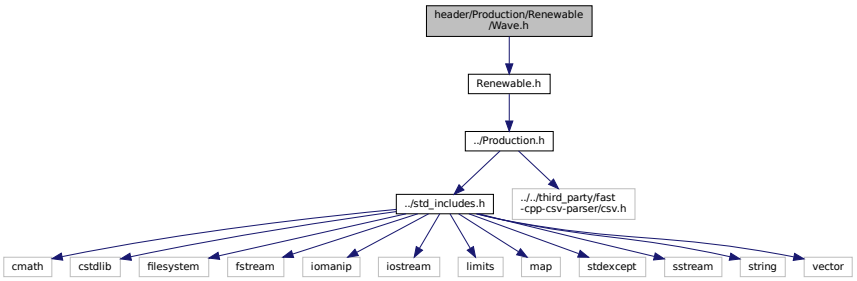
```
34                                {
35     TIDAL_POWER_CUBIC,
36     TIDAL_POWER_EXPONENTIAL,
37     TIDAL_POWER_LOOKUP,
38     N_TIDAL_POWER_PRODUCTION_MODELS
39 };
```

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

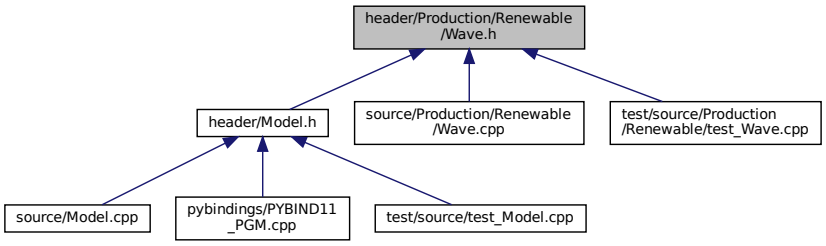
Header file the Wave class.

```
#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 }

5.10.1 Detailed Description

Header file the Wave class.

5.10.2 Enumeration Type Documentation

5.10.2.1 WavePowerProductionModel

```
enum WavePowerProductionModel
```

Enumerator

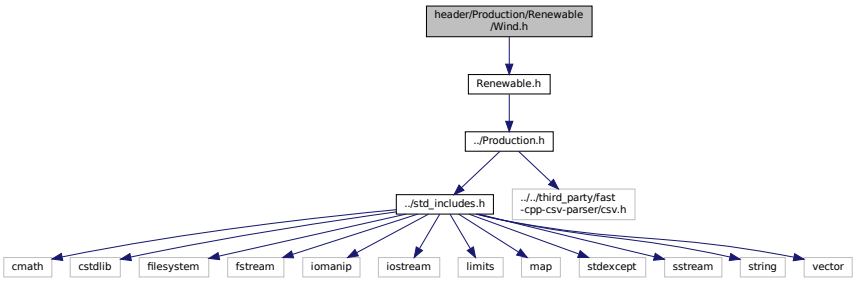
WAVE_POWER_GAUSSIAN	A Gaussian power production model.
WAVE_POWER_PARABOLOID	A paraboloid power production model.
WAVE_POWER_LOOKUP	Lookup from a given performance matrix.
N_WAVE_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in WavePowerProductionModel.

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

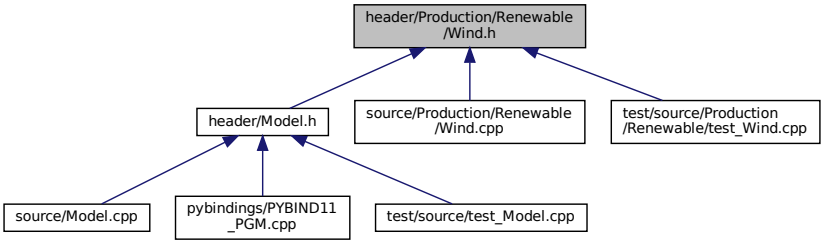
5.11 header/Production/Renewable/Wind.h File Reference

Header file 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 }

5.11.1 Detailed Description

Header file the Wind class.

5.11.2 Enumeration Type Documentation

5.11.2.1 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.

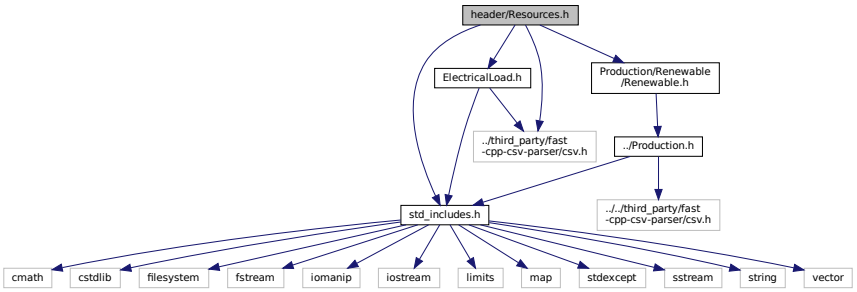
```
34                              {
35     WIND_POWER_EXPONENTIAL,
36     WIND_POWER_LOOKUP,
37     N_WIND_POWER_PRODUCTION_MODELS
38 };
```

5.12 header/Resources.h File Reference

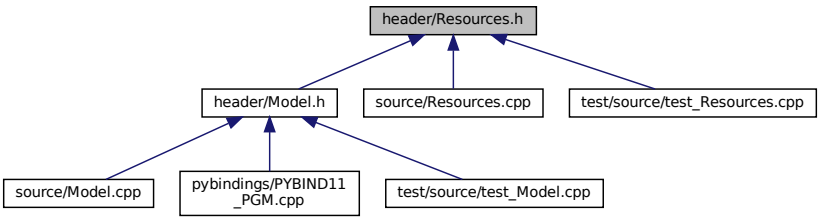
Header file the Resources class.

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

5.12.1 Detailed Description

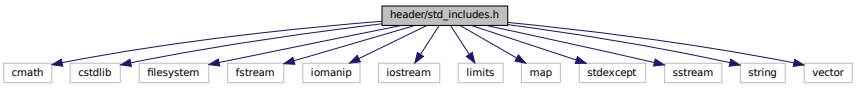
Header file the Resources class.

5.13 header/std_includes.h File Reference

Header file which simply batches together the usual, standard includes.

```
#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#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:

5.13.1 Detailed Description

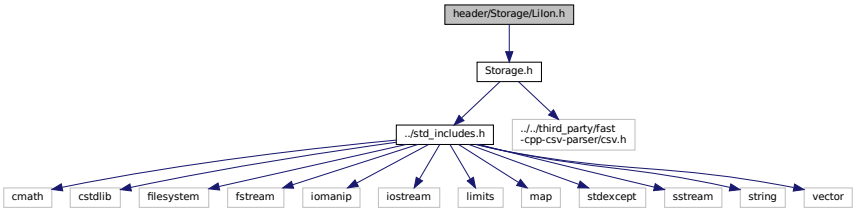
Header file which simply batches together the usual, standard includes.

5.14 header/Storage/LiIon.h File Reference

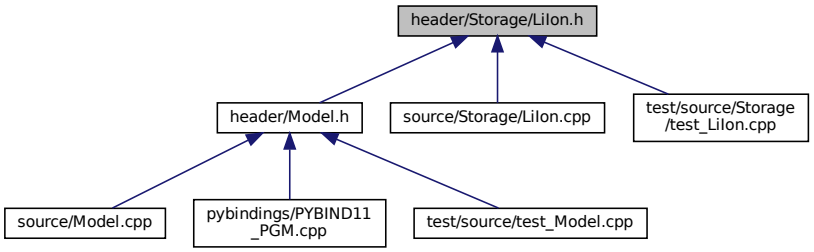
Header file the LiIon class.

```
#include "Storage.h"
```

Include dependency graph for LiIon.h:



This graph shows which files directly or indirectly include this file:



Classes

- class LiIon

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

5.14.1 Detailed Description

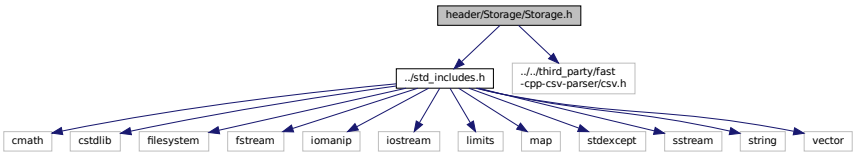
Header file the LiIon class.

5.15 header/Storage/Storage.h File Reference

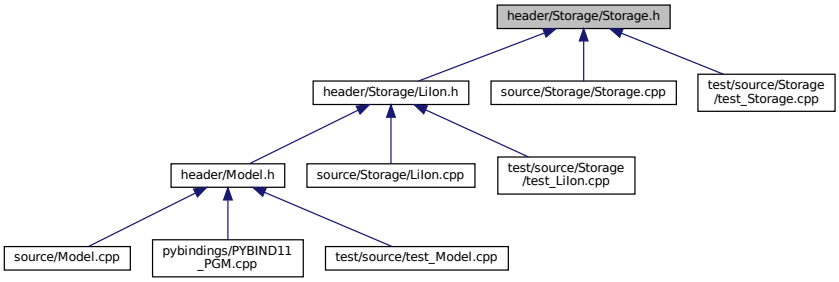
Header file the Storage class.

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

Include dependency graph for Storage.h:



This graph shows which files directly or indirectly include this file:



Classes

- class Storage

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

5.15.1 Detailed Description

Header file the Storage class.

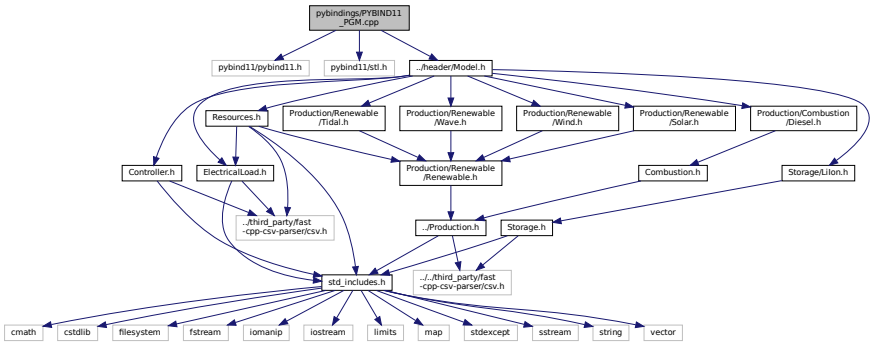
5.16 pybindings/PYBIND11_PGM.cpp File Reference

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)

5.16.1 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/

5.16.2 Function Documentation

5.16.2.1 PYBIND11_MODULE()

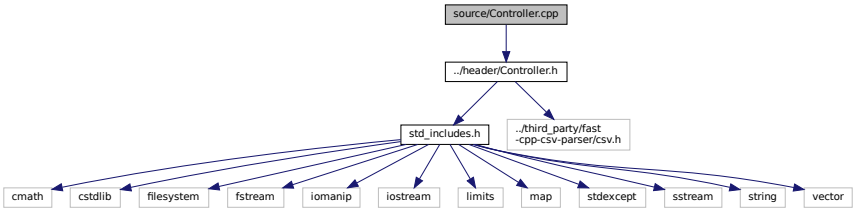
```
PYBIND11_MODULE (
             PGMcpp ,
             m   )
30                           {
31
32 // ================ Controller ================ //
33 /*
34 pybind11::class_<Controller>(m, "Controller")
35     .def(pybind11::init());
36 */
37 // ================ END Controller ================ //
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() */
```

5.17 source/Controller.cpp File Reference

Implementation file for the Controller class.

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



5.17.1 Detailed Description

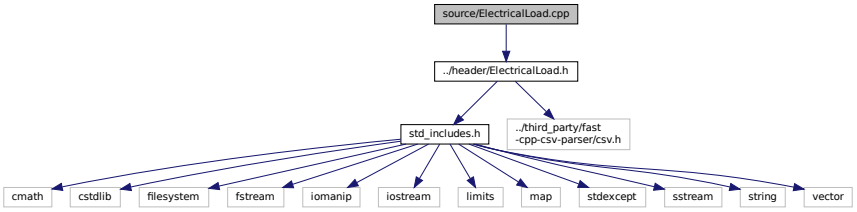
Implementation file for the Controller class.

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

5.18 source/ElectricalLoad.cpp File Reference

Implementation file for the ElectricalLoad class.

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



5.18.1 Detailed Description

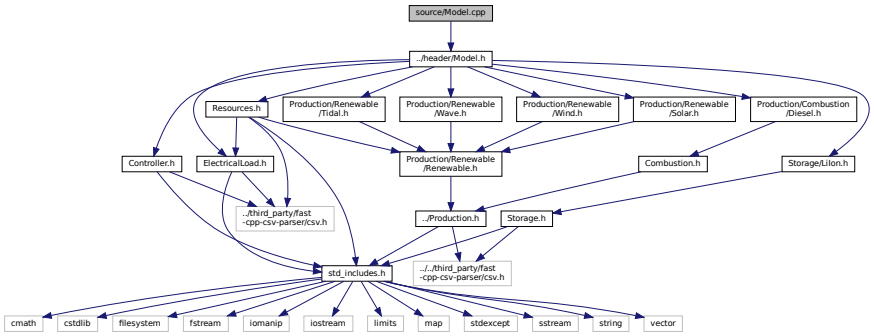
Implementation file for the ElectricalLoad class.

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

5.19 source/Model.cpp File Reference

Implementation file for the Model class.

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



5.19.1 Detailed Description

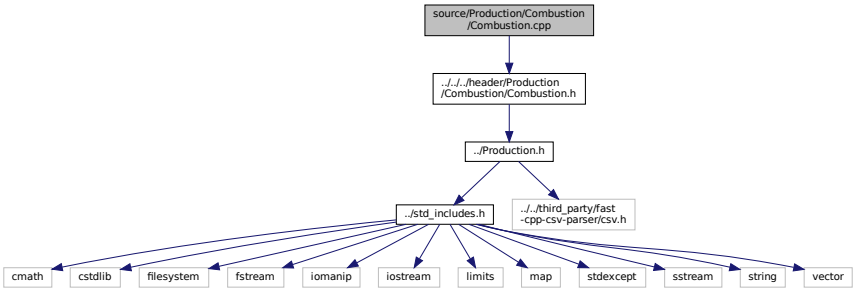
Implementation file for the Model class.

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.

5.20 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:



5.20.1 Detailed Description

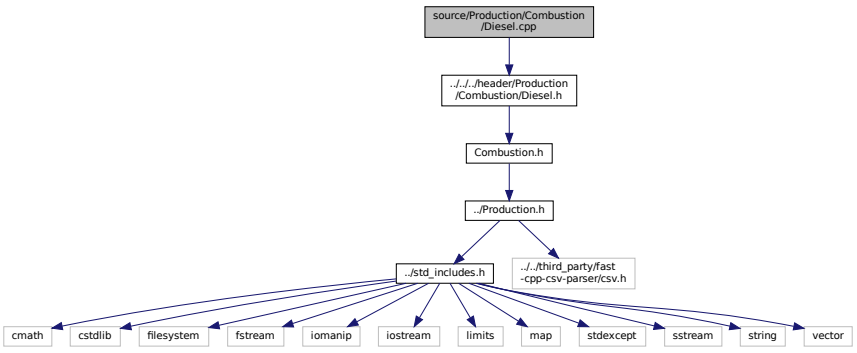
Implementation file for the Combustion class.

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.

5.21 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:



5.21.1 Detailed Description

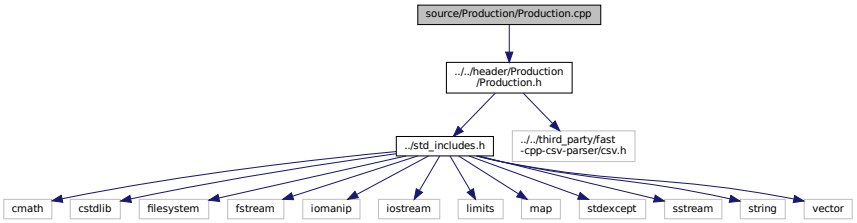
Implementation file for the Diesel class.

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

5.22 source/Production/Production.cpp File Reference

Implementation file for the Production class.

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



5.22.1 Detailed Description

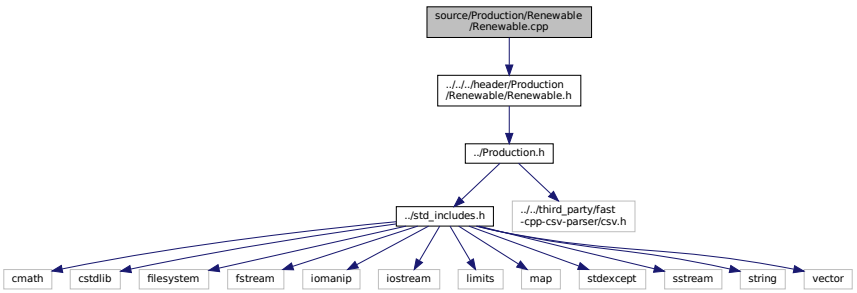
Implementation file for the Production class.

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

5.23 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:



5.23.1 Detailed Description

Implementation file for the Renewable class.

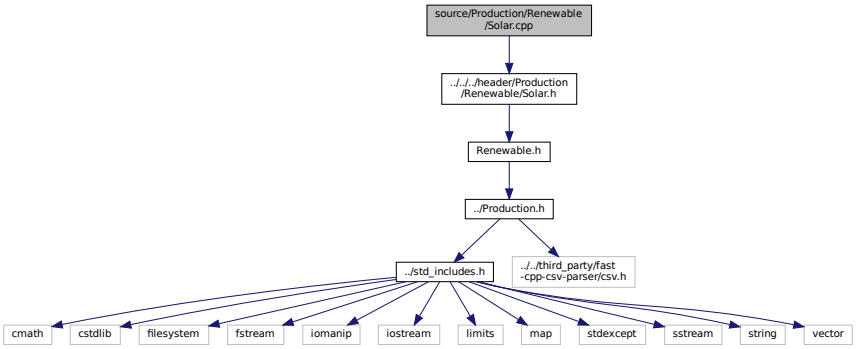
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

5.24 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:



5.24.1 Detailed Description

Implementation file for the Solar class.

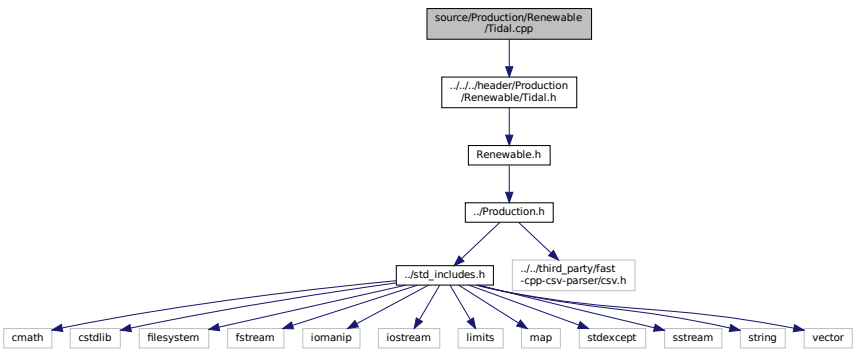
A derived class of the Renewable branch of Production which models solar production.

5.25 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:



5.25.1 Detailed Description

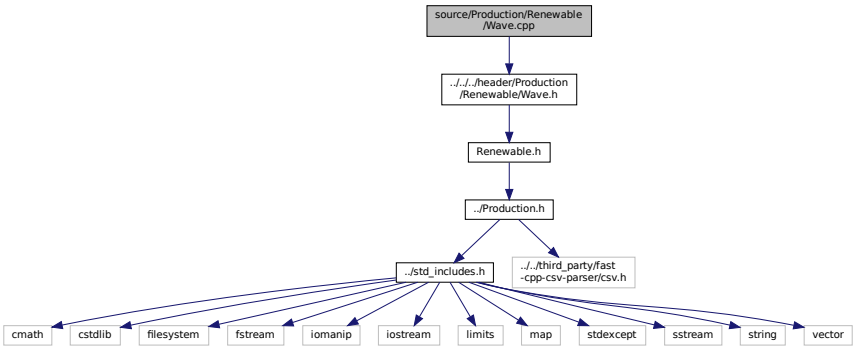
Implementation file for the Tidal class.

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

5.26 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:



5.26.1 Detailed Description

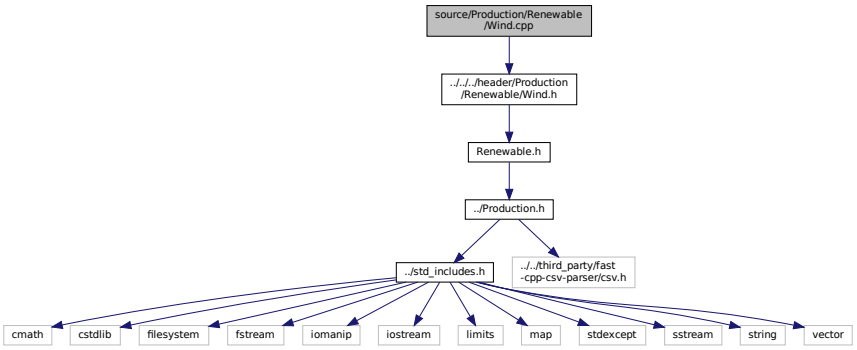
Implementation file for the Wave class.

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

5.27 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:



5.27.1 Detailed Description

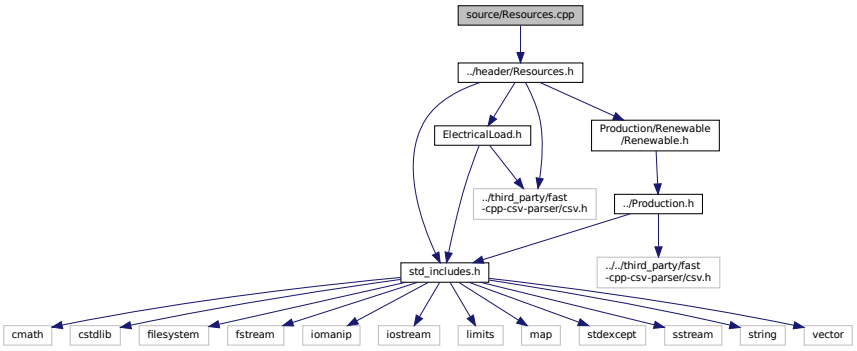
Implementation file for the Wind class.

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

5.28 source/Resources.cpp File Reference

Implementation file for the Resources class.

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



5.28.1 Detailed Description

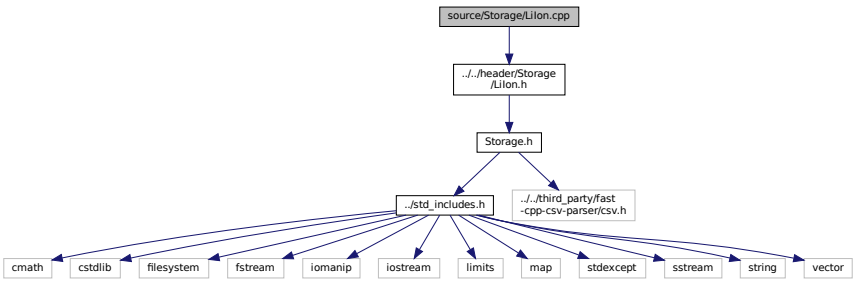
Implementation file for the Resources class.

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

5.29 source/Storage/LiIon.cpp File Reference

Implementation file for the LiIon class.

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



5.29.1 Detailed Description

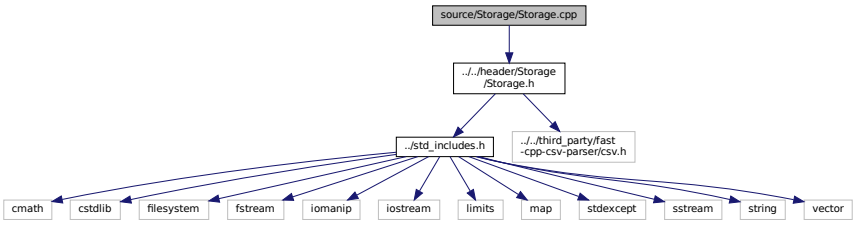
Implementation file for the LiIon class.

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

5.30 source/Storage/Storage.cpp File Reference

Implementation file for the Storage class.

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



5.30.1 Detailed Description

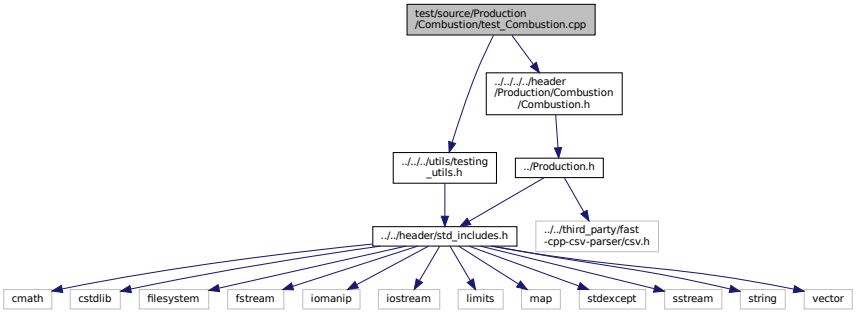
Implementation file for the Storage class.

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

5.31 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)

5.31.1 Detailed Description

Testing suite for Combustion class.

A suite of tests for the Combustion class.

5.31.2 Function Documentation

5.31.2.1 main()

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

```
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31 
32     printGold("\tTesting Production <-- Combustion");
33 
34     srand(time(NULL));
35 
36 
37 try {
38 
39 // ======== CONSTRUCTION ========================================================== //
40 
41 CombustionInputs combustion_inputs;
42 
43 Combustion test_combustion(8760, combustion_inputs);
44 
45 // ======== END CONSTRUCTION ====================================================== //
46 
47 
48 
49 // ======== ATTRIBUTES ============================================================ //
50 
51 testTruth(
52     not combustion_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56 
57 testFloatEquals(
58     test_combustion.fuel_consumption_vec_L.size(),
59     8760,
60     __FILE__,
61     __LINE__
62 );
63 
64 testFloatEquals(
65     test_combustion.fuel_cost_vec.size(),
66     8760,
67     __FILE__,
68     __LINE__
69 );
70 
71 testFloatEquals(
72     test_combustion.CO2_emissions_vec_kg.size(),
73     8760,
74     __FILE__,
75     __LINE__
76 );
77 
```

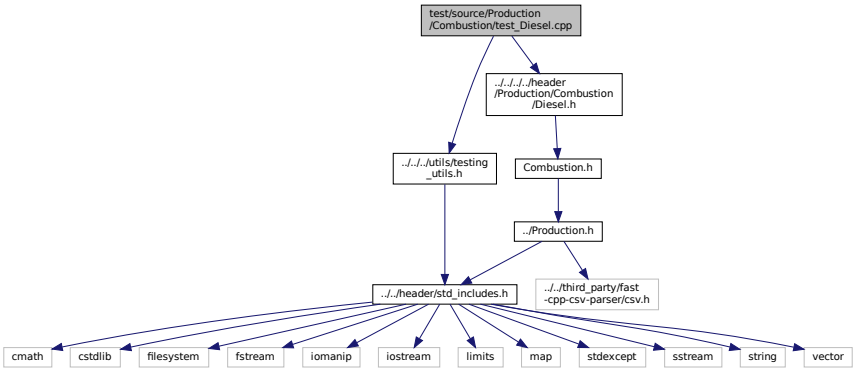
```
78 testFloatEquals(
79     test_combustion.CO_emissions_vec_kg.size(),
80     8760,
81     __FILE__,
82     __LINE__
83 );
84
85 testFloatEquals(
86     test_combustion.NOx_emissions_vec_kg.size(),
87     8760,
88     __FILE__,
89     __LINE__
90 );
91
92 testFloatEquals(
93     test_combustion.SOx_emissions_vec_kg.size(),
94     8760,
95     __FILE__,
96     __LINE__
97 );
98
99 testFloatEquals(
100     test_combustion.CH4_emissions_vec_kg.size(),
101     8760,
102     __FILE__,
103     __LINE__
104 );
105
106 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() */
```

5.32 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)

5.32.1 Detailed Description

Testing suite for Diesel class.

A suite of tests for the Diesel class.

5.32.2 Function Documentation

5.32.2.1 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, 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, 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
```

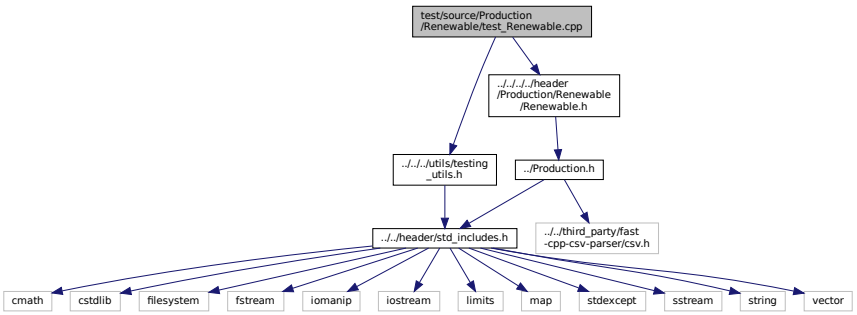
```
397 }   /* try */
398
399
400 catch (...) {
401     delete test_diesel_ptr;
402
403     printGold(" ... ");
404     printRed("FAIL");
405     std::cout « std::endl;
406     throw;
407 }
408
409
410 delete test_diesel_ptr;
411
412 printGold(" ... ");
413 printGreen("PASS");
414 std::cout « std::endl;
415 return 0;
416
417 }   /* main() */
```

5.33 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)

5.33.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

5.33.2 Function Documentation

5.33.2.1 main()

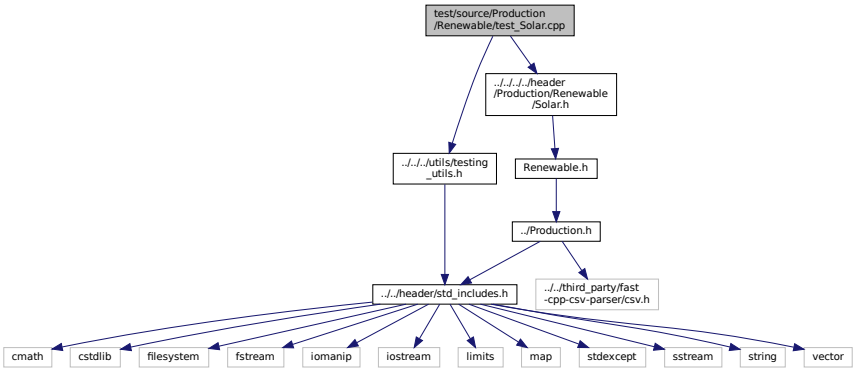
```
int main (
              int argc,
              char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

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

    srand(time(NULL));


try {

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

RenewableInputs renewable_inputs;

Renewable test_renewable(8760, renewable_inputs);

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



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

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

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

}   /* try */


catch (...) {
    //...

    printGold(" ............... ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ............... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

5.34 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)

5.34.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar 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 <-- 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, 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, 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, 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(
```

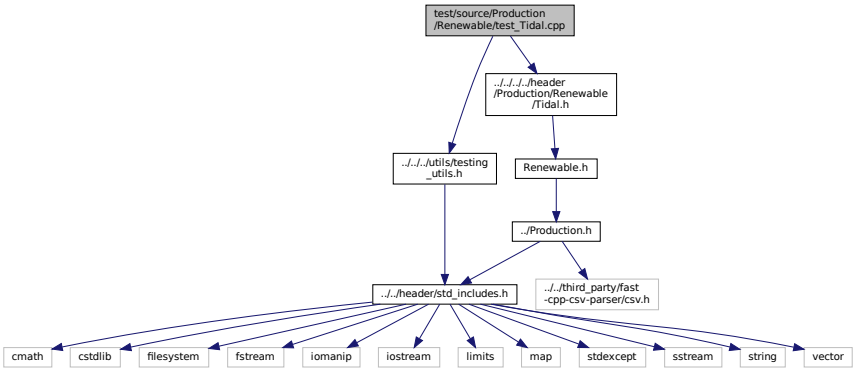
```
223             solar_resource_kWm2,
224             0,
225             __FILE__,
226             __LINE__
227         );
228
229         testGreaterThan(
230             test_solar_ptr->operation_maintenance_cost_vec[i],
231             0,
232             __FILE__,
233             __LINE__
234         );
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() */
```

5.35 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)

5.35.1 Detailed Description

Testing suite for Tidal class.

A suite of tests for the Tidal 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 <-- Tidal");
33 
34     srand(time(NULL));
35 
36     Renewable* test_tidal_ptr;
37 
38 try {
39 
40 // ======== CONSTRUCTION ============================================================ //
41 
42 bool error_flag = true;
43 
44 try {
45     TidalInputs bad_tidal_inputs;
46     bad_tidal_inputs.design_speed_ms = -1;
47 
48     Tidal bad_tidal(8760, bad_tidal_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 TidalInputs tidal_inputs;
59
60 test_tidal_ptr = new Tidal(8760, tidal_inputs);
61
62 // ======== END CONSTRUCTION ============================================================ //
63
64
65
66 // ======== ATTRIBUTES ================================================================== //
67
68 testTruth(
69     not tidal_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73
74 testFloatEquals(
75     test_tidal_ptr->type,
76     RenewableType :: TIDAL,
77     __FILE__,
78     __LINE__
79 );
80
81 testTruth(
82     test_tidal_ptr->type_str == "TIDAL",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_tidal_ptr->capital_cost,
89     500237.446725,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_tidal_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_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) {
```

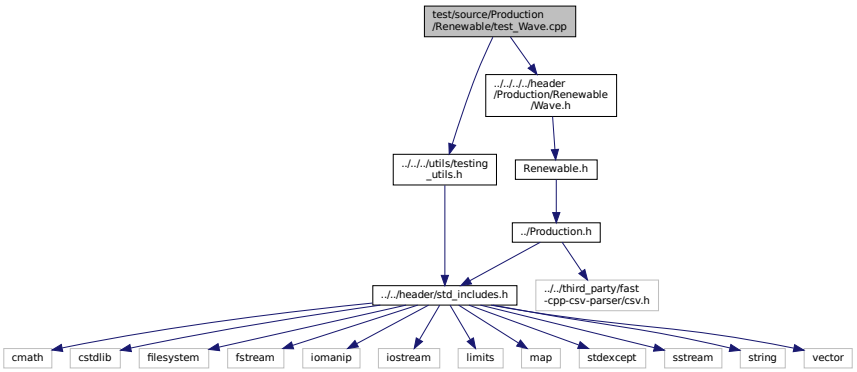
```
223         testGreaterThan(
224             tidal_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229 
230         testGreaterThan(
231             test_tidal_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __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() */
```

5.36 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)

5.36.1 Detailed Description

Testing suite for Wave class.

A suite of tests for the Wave 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 <-- 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, 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, 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, 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 */
```

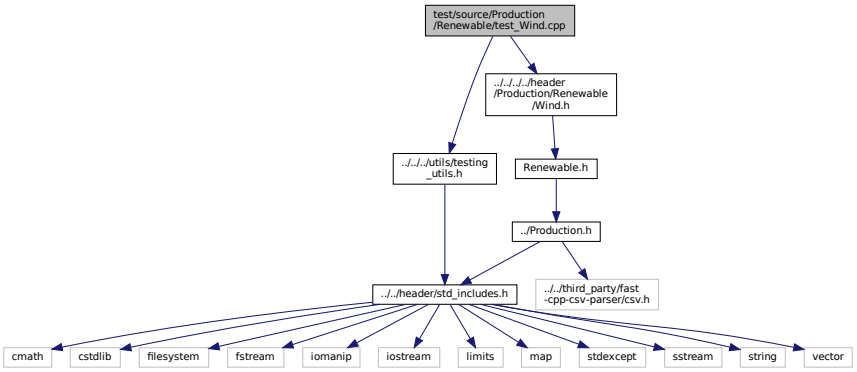
```
252
253
254 catch (...) {
255     delete test_wave_ptr;
256
257     printGold(" ..... ");
258     printRed("FAIL");
259     std::cout « std::endl;
260     throw;
261 }
262
263
264 delete test_wave_ptr;
265
266 printGold(" ..... ");
267 printGreen("PASS");
268 std::cout « std::endl;
269 return 0;
270 }   /* main() */
```

5.37 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)

5.37.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind 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 <-- Wind");

    srand(time(NULL));

    Renewable* test_wind_ptr;

try {

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

bool error_flag = true;

try {
    WindInputs bad_wind_inputs;
    bad_wind_inputs.design_speed_ms = -1;

    Wind bad_wind(8760, bad_wind_inputs);

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

WindInputs wind_inputs;

test_wind_ptr = new Wind(8760, wind_inputs);

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


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

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

testFloatEquals(
    test_wind_ptr->type,
    RenewableType :: WIND,
    __FILE__,
    __LINE__
);

testTruth(
    test_wind_ptr->type_str == "WIND",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->capital_cost,
    450356.170088,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->operation_maintenance_cost_kWh,
    0.034953,
    __FILE__,
    __LINE__
);

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


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

```

```
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, 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
```

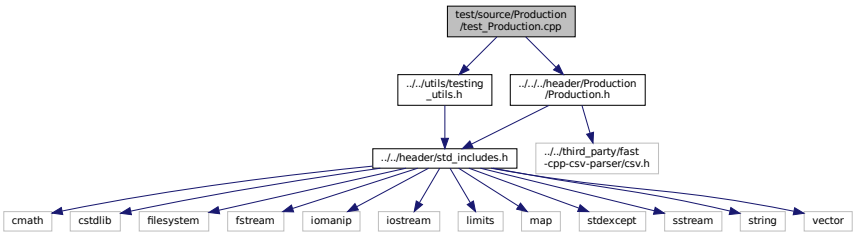
```
194     else {
195         testTruth(
196             not test_wind_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201 
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
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() */
```

5.38 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)

5.38.1 Detailed Description

Testing suite for Production class.

A suite of tests for the Production 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("\n\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, 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, 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__,
```

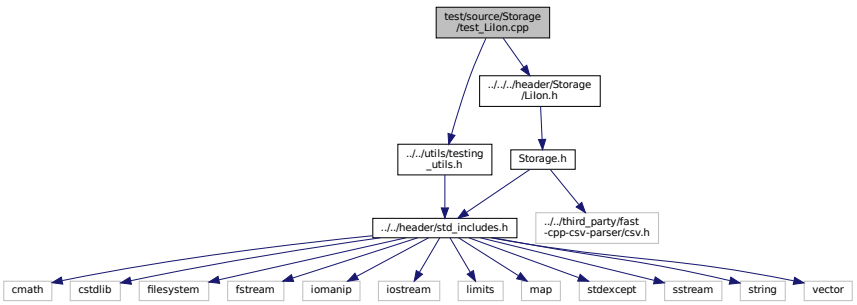
```
139     __LINE__
140 );
141
142 testFloatEquals(
143     test_production.operation_maintenance_cost_vec.size(),
144     8760,
145     __FILE__,
146     __LINE__
147 );
148
149 // ======== END ATTRIBUTES ============================================================ //
150
151 }   /* try */
152
153
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() */
```

5.39 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)

5.39.1 Detailed Description

Testing suite for LiIon class.

A suite of tests for the LiIon class.

5.39.2 Function Documentation

5.39.2.1 main()

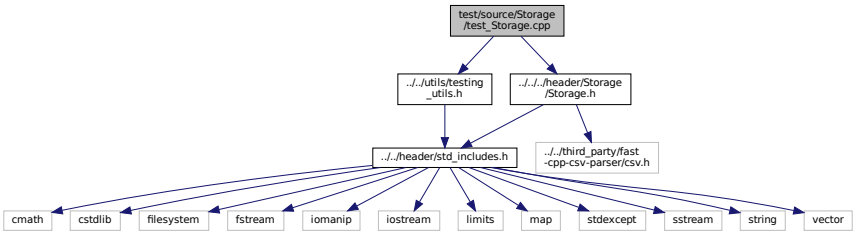
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Storage <-- LiIon");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
42         //...
43
44         printGold(" ..................... ");
45         printRed("FAIL");
46         std::cout « std::endl;
47         throw;
48     }
49
50
51     printGold(" ..................... ");
52     printGreen("PASS");
53     std::cout « std::endl;
54     return 0;
55 }   /* main() */
```

5.40 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)

5.40.1 Detailed Description

Testing suite for Storage class.

A suite of tests for the Storage class.

5.40.2 Function Documentation

5.40.2.1 main()

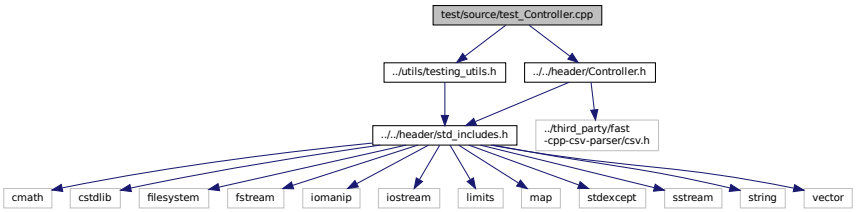
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Storage");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
42         //...
43
44         printGold(" ..................... ");
45         printRed("FAIL");
46         std::cout « std::endl;
47         throw;
48     }
49
50
51     printGold(" ............................... ");
52     printGreen("PASS");
53     std::cout « std::endl;
54     return 0;
55 }   /* main() */
```

5.41 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)

5.41.1 Detailed Description

Testing suite for Controller class.

A suite of tests for the Controller class.

5.41.2 Function Documentation

5.41.2.1 main()

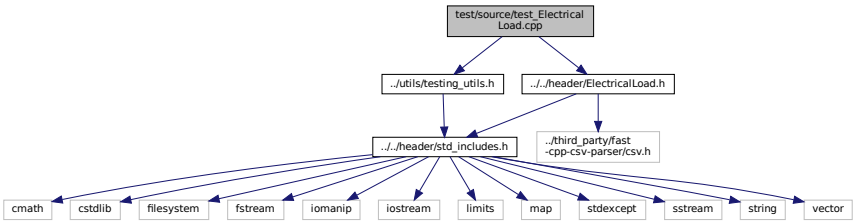
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Controller");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
42         //...
43
44         printGold(" ............................. ");
45         printRed("FAIL");
46         std::cout « std::endl;
47         throw;
48     }
49
50
51     printGold(" ............................. ");
52     printGreen("PASS");
53     std::cout « std::endl;
54     return 0;
55 }   /* main() */
```

5.42 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)

5.42.1 Detailed Description

Testing suite for ElectricalLoad class.

A suite of tests for the ElectricalLoad 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 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_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,
```

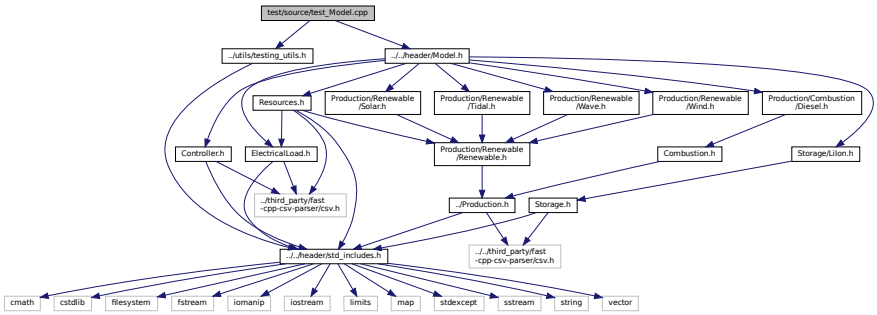
```
140     480.225974100847,
141     457.318764401085,
142     418.177339948609,
143     414.399018364126,
144     409.678420185754,
145     404.768766016563,
146     401.699589920585,
147     402.44339040654,
148     398.138372541906,
149     396.010498627646,
150     390.165117432277,
151     375.850429417013,
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() */
```

5.43 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)

5.43.1 Detailed Description

Testing suite for Model class.

A suite of tests for the Model 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 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;
45     bad_model_inputs.path_2_electrical_load_time_series =
46         "data/test/bad_path_240984069830.csv";
47
48     Model bad_model(bad_model_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  std::string path_2_electrical_load_time_series =
59      "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
60
61  ModelInputs test_model_inputs;
62  test_model_inputs.path_2_electrical_load_time_series =
63      path_2_electrical_load_time_series;
64
65  Model test_model(test_model_inputs);
66
67  // ======== END CONSTRUCTION ============================================================ //
68
69
70  // ======== ATTRIBUTES ================================================================== //
71
72  testTruth(
73      test_model.electrical_load.path_2_electrical_load_time_series ==
74      path_2_electrical_load_time_series,
75      __FILE__,
76      __LINE__
77  );
78
79  testFloatEquals(
80      test_model.electrical_load.n_points,
81      8760,
82      __FILE__,
83      __LINE__
84  );
85
86  testFloatEquals(
87      test_model.electrical_load.n_years,
88      0.999886,
89      __FILE__,
90      __LINE__
91  );
92
93  testFloatEquals(
94      test_model.electrical_load.min_load_kW,
95      82.1211213927802,
96      __FILE__,
97      __LINE__
98  );
99
100 testFloatEquals(
101     test_model.electrical_load.mean_load_kW,
102     258.373472633202,
103     __FILE__,
104     __LINE__
105 );
106
107
108 testFloatEquals(
109     test_model.electrical_load.max_load_kW,
110     500,
111     __FILE__,
112     __LINE__
113 );
114
115
116 std::vector<double> expected_dt_vec_hrs (48, 1);
117
118 std::vector<double> expected_time_vec_hrs = {
119      0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
120     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
121     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
122     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
123 };
124
125 std::vector<double> expected_load_vec_kW = {
126     360.253836463674,
127     355.171277826775,
128     353.776453532298,
129     353.75405737934,
130     346.592867404975,
131     340.132411175118,
132     337.354867340578,
133     340.644115618736,
134     363.639028500678,
135     378.787797779238,
136     372.215798201712,
137     395.093925731298,
138     402.325427142659,
```

```
139     386.907725462306,
140     380.709170928091,
141     372.062070914977,
142     372.328646856954,
143     391.841444284136,
144     394.029351759596,
145     383.369407765254,
146     381.093099675206,
147     382.604158946193,
148     390.744843709034,
149     383.13949492437,
150     368.150393976985,
151     364.629744480226,
152     363.572736804082,
153     359.854924202248,
154     355.207590170267,
155     349.094656012401,
156     354.365935871597,
157     343.380608328546,
158     404.673065729266,
159     486.296896820126,
160     480.225974100847,
161     457.318764401085,
162     418.177339948609,
163     414.399018364126,
164     409.678420185754,
165     404.768766016563,
166     401.699589920585,
167     402.44339040654,
168     398.138372541906,
169     396.010498627646,
170     390.165117432277,
171     375.850429417013,
172     365.567100746484,
173     365.429624610923
174 };
175
176 for (int i = 0; i < 48; i++) {
177     testFloatEquals(
178         test_model.electrical_load.dt_vec_hrs[i],
179         expected_dt_vec_hrs[i],
180         __FILE__,
181         __LINE__
182     );
183
184     testFloatEquals(
185         test_model.electrical_load.time_vec_hrs[i],
186         expected_time_vec_hrs[i],
187         __FILE__,
188         __LINE__
189     );
190
191     testFloatEquals(
192         test_model.electrical_load.load_vec_kW[i],
193         expected_load_vec_kW[i],
194         __FILE__,
195         __LINE__
196     );
197 }
198
199 // ======== END ATTRIBUTES ============================================================ //
200
201
202
203 // ======== METHODS ================================================================== //
204
205 //...
206
207 // ======== END METHODS ============================================================== //
208
209 }   /* try */
210
211
212 catch (...) {
213     //...
214
215     printGold(" ................................. ");
216     printRed("FAIL");
217     std::cout « std::endl;
218     throw;
219 }
220
221
222 printGold(" ................................. ");
223 printGreen("PASS");
224 std::cout « std::endl;
225 return 0;
```

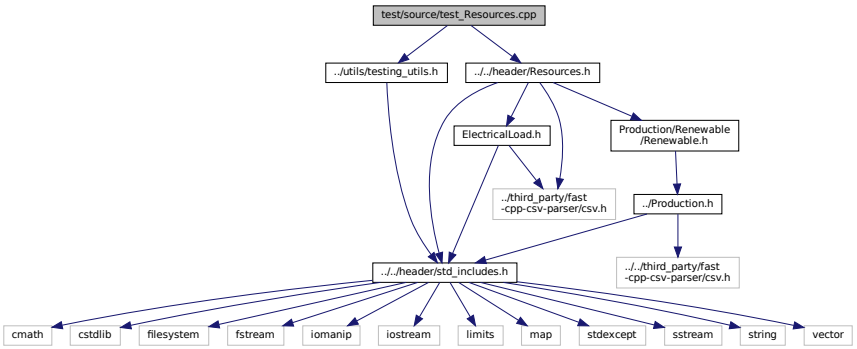
```
226 }   /* main() */
```

5.44 test/source/test_Resources.cpp File Reference

Testing suite for Resources class.

```
#include "../utils/testing_utils.h"
#include "../../header/Resources.h"
```

Include dependency graph for test_Resources.cpp:



Functions

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

5.44.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

5.44.2 Function Documentation

5.44.2.1 main()

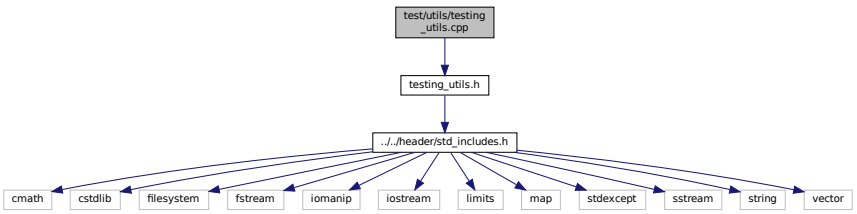
```
int main (
              int argc,
              char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Resources");

    srand(time(NULL));


try {

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

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 =============================================================== //

//...

// ======== END METHODS =========================================================== //

}   /* try */


catch (...) {
    printGold(" ............................. ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ............................. ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

5.45 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.

5.45.1 Detailed Description

Header file for various PGMcpp testing utilities.

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

5.45.2 Function Documentation

5.45.2.1 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() */
```

5.45.2.2 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() */
```

5.45.2.3 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() */
```

5.45.2.4 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() */
```

5.45.2.5 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() */
```

5.45.2.6 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() */
```

5.45.2.7 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.

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__").

```
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() */
```

5.45.2.8 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() */
```

5.45.2.9 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() */
```

5.45.2.10 testTruth()

```
void testTruth (
            bool statement,
            std::string file,
            int line )
```

Tests if the given statement is true.

Parameters

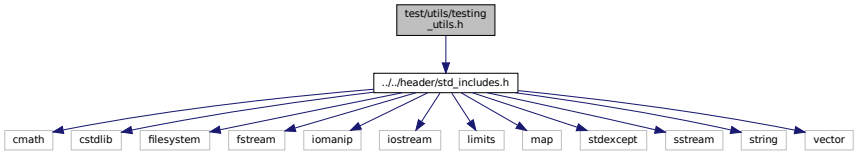
statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
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() */
```

5.46 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.

5.46.1 Detailed Description

Header file for various PGMcpp testing utilities.

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

5.46.2 Macro Definition Documentation

5.46.2.1 FLOAT_TOLERANCE

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

A tolerance for application to floating point equality tests.

5.46.3 Function Documentation

5.46.3.1 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() */
```

5.46.3.2 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() */
```

5.46.3.3 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() */
```

5.46.3.4 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() */
```

5.46.3.5 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.
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() */
```

5.46.3.6 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() */
```

5.46.3.7 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() */
```

5.46.3.8 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() */
```

5.46.3.9 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() */
```

5.46.3.10 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() */
```

Index