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

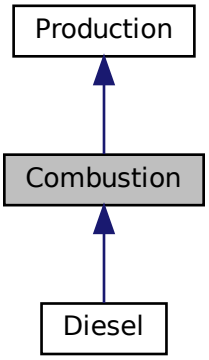
Class Documentation

4.1 Combustion Class Reference

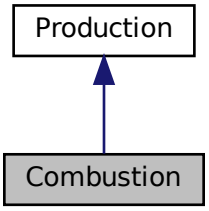
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

`#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:

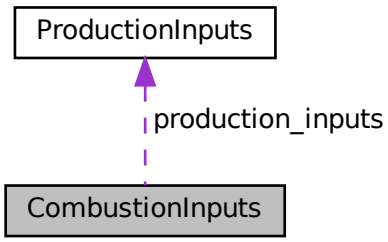
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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.
- ~Controller (void)

  Destructor for the Controller class.

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.

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

4.3.2.2 ~Controller()

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

Destructor for the Controller class.

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

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

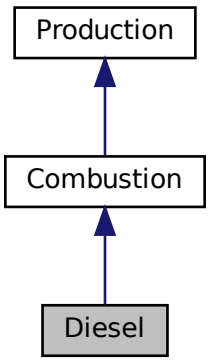
- header/Controller.h
- source/Controller.cpp

4.4 Diesel Class Reference

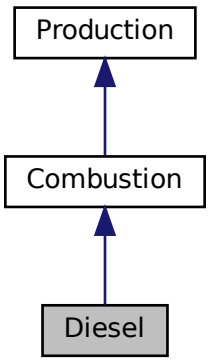
A derived class of the Combustion branch of Production which models production using a diesel generator.

`#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.

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

4.4.2.2 Diesel() [2/2]

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

```
326                                                     :
327 Combustion(n_points, diesel_inputs.combustion_inputs)
328 {
329     //  1. check inputs
330     this->__checkInputs(diesel_inputs);
331 
332     //  2. set attributes
333     this->type = CombustionType :: 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:

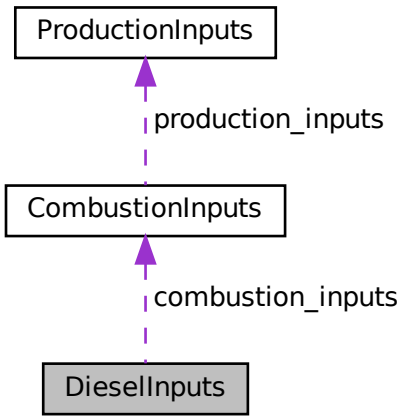
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

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 for the ElectricalLoad class.
- ~ElectricalLoad (void)
  Destructor for the ElectricalLoad class.

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()

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

Constructor for the ElectricalLoad class.

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

4.6.2.2 ~ElectricalLoad()

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

Destructor for the ElectricalLoad class.

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

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:

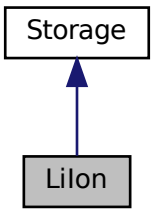
- header/Production/Combustion/Combustion.h

4.8 LiIon Class Reference

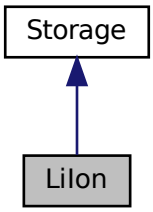
A derived class of Storage which models energy storage by way of lithium-ion batteries.

`#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:

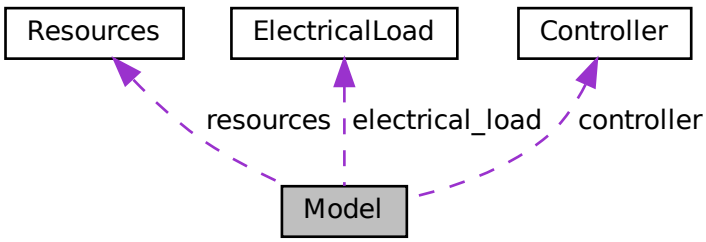
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

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 for the Model class.
- ∼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()

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

Constructor for the Model class.

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

4.9.2.2 ~Model()

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

Destructor for the Model class.

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

4.9.3 Member Data Documentation

4.9.3.1 combustion_ptr_vec

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

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

4.9.3.2 controller

```
Controller Model::controller
```

Controller component of Model.

4.9.3.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

ElectricalLoad component of Model.

4.9.3.4 renewable_ptr_vec

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

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

4.9.3.5 resources

```
Resources Model::resources
```

Resources component of Model.

4.9.3.6 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:

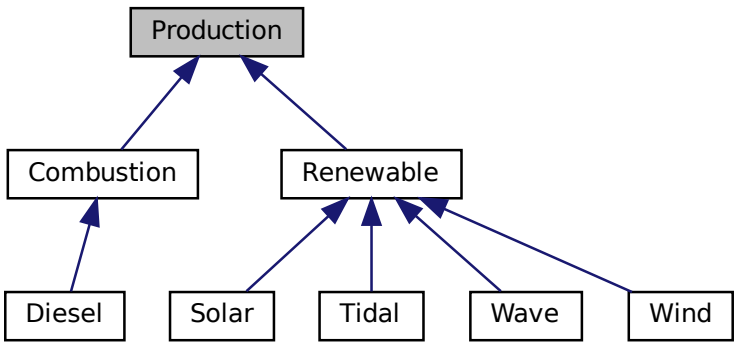
- header/Model.h
- source/Model.cpp

4.10 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::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).

4.10.1 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.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 Production() [1/2]

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

Constructor (dummy) for the Production class.

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

4.10.2.2 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() */
```

4.10.2.3 ~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() */
```

4.10.3 Member Function Documentation

4.10.3.1 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 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() */
```

4.10.4 Member Data Documentation

4.10.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.10.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.10.4.3 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).

4.10.4.4 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.

4.10.4.5 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.

4.10.4.6 is_running

```
bool Production::is_running
```

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

4.10.4.7 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.

4.10.4.8 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).

4.10.4.9 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.

4.10.4.10 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.10.4.11 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.10.4.12 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.10.4.13 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.10.4.14 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.

4.10.4.15 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).

4.10.4.16 print_flag

```
bool Production::print_flag
```

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

4.10.4.17 production_vec_kW

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

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

4.10.4.18 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.

4.10.4.19 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.10.4.20 running_hours

```
double Production::running_hours
```

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

4.10.4.21 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.

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

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

4.11 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.

4.11.1 Detailed Description

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

4.11.2 Member Data Documentation

4.11.2.1 capacity_kW

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

The rated production capacity [kW] of the asset.

4.11.2.2 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).

4.11.2.3 nominal_discount_annual

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

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

4.11.2.4 nominal_inflation_annual

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

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

4.11.2.5 print_flag

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

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

4.11.2.6 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:

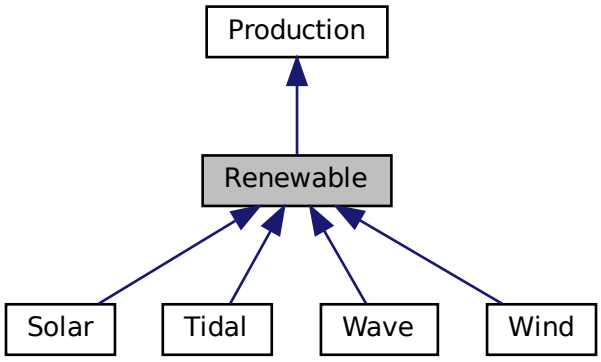
- header/Production/Production.h

4.12 Renewable Class Reference

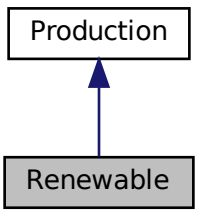
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#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 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.

4.12.1 Detailed Description

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

4.12.2 Constructor & Destructor Documentation

4.12.2.1 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.

```
58 {
59     //...
60
61     return;
62 }   /* Renewable() */
```

4.12.2.2 Renewable() [2/2]

```
Renewable::Renewable (
              int n_points,
              RenewableInputs renewable_inputs )
80                                                                    :
81 Production(n_points, renewable_inputs.production_inputs)
82 {
83     //  1. check inputs
84     this->__checkInputs(renewable_inputs);
85
86     //  2. set attributes
87     //...
88
89     //  3. construction print
90     if (this->print_flag) {
91         std::cout « "Renewable object constructed at " « this « std::endl;
92     }
93
94     return;
95 }   /* Renewable() */
```

4.12.2.3 ~Renewable()

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

Destructor for the Renewable class.

```
159 {
160     //  1. destruction print
161     if (this->print_flag) {
162         std::cout « "Renewable object at " « this « " destroyed" « std::endl;
163     }
164
165     return;
166 }   /* ~Renewable() */
```

4.12.3 Member Function Documentation

4.12.3.1 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 Solar.

```
131 {
132     //  1. invoke base class method
133     load_kW = Production :: commit(
134         timestep,
135         dt_hrs,
136         production_kW,
137         load_kW
138     );
139
140
141     //...
142
143     return load_kW;
144 }   /* commit() */
```

4.12.3.2 computeProductionkW()

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

Reimplemented in Solar.

```
83 {return 0;}
```

4.12.4 Member Data Documentation

4.12.4.1 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.

4.12.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

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

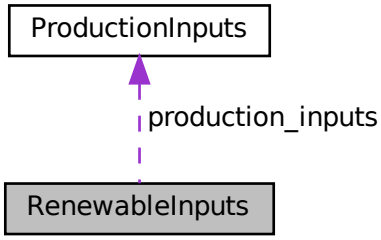
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

4.13 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.

4.13.1 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.

4.13.2 Member Data Documentation

4.13.2.1 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

4.14 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.
- ~Resources (void)

  Destructor for the Resources class.

4.14.1 Detailed Description

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

4.14.2 Constructor & Destructor Documentation

4.14.2.1 Resources()

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

Constructor for the Resources class.

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

4.14.2.2 ~Resources()

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

Destructor for the Resources class.

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

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

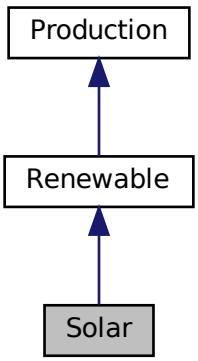
- header/Resources.h
- source/Resources.cpp

4.15 Solar Class Reference

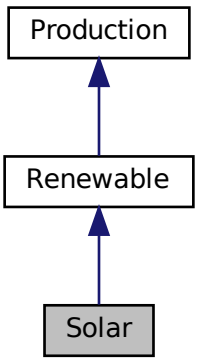
A derived class of the Renewable branch of Production which models solar production.

`#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.).

4.15.1 Detailed Description

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

4.15.2 Constructor & Destructor Documentation

4.15.2.1 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.

```
141 {
142     //...
143
144     return;
145 }   /* Solar() */
```

4.15.2.2 Solar() [2/2]

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

```
163                                                 :
164 Renewable(n_points, solar_inputs.renewable_inputs)
165 {
166     //  1. check inputs
167     this->__checkInputs(solar_inputs);
168 
169     //  2. set attributes
170     this->type = RenewableType :: SOLAR;
171 
172     this->resource_key = solar_inputs.resource_key;
173 
174     this->derating = solar_inputs.derating;
175 
176     if (solar_inputs.capital_cost < 0) {
177         this->capital_cost = this->__getGenericCapitalCost();
178     }
179 
180     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
181         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
182     }
183 
184     if (this->is_sunk) {
185         this->capital_cost_vec[0] = this->capital_cost;
186     }
187 
188     //  3. construction print
189     if (this->print_flag) {
190         std::cout << "Solar object constructed at " << this << std::endl;
191     }
192 
193     return;
194 }   /* Renewable() */
```

4.15.2.3 ~Solar()

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

Destructor for the Solar class.

```
303 {
304     //  1. destruction print
305     if (this->print_flag) {
306         std::cout << "Solar object at " << this << " destroyed" << std::endl;
307     }
308 
309     return;
310 }   /* ~Solar() */
```

4.15.3 Member Function Documentation

4.15.3.1 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.

```
272 {
273     //  1. handle start/stop
274     this->__handleStartStop(timestep, dt_hrs, production_kW);
275 
276     //  2. invoke base class method
277     load_kW = Renewable :: commit(
278         timestep,
279         dt_hrs,
280         production_kW,
281         load_kW
282     );
283 
284 
285     //...
286 
287     return load_kW;
288 }   /* commit() */
```

4.15.3.2 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.

```
228 {
```

```
229     if (solar_resource_kWm2 <= 0) {
230         return 0;
231     }
232
233     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
234
235     return production_kW;
236 }   /* computeProductionkW() */
```

4.15.4 Member Data Documentation

4.15.4.1 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:

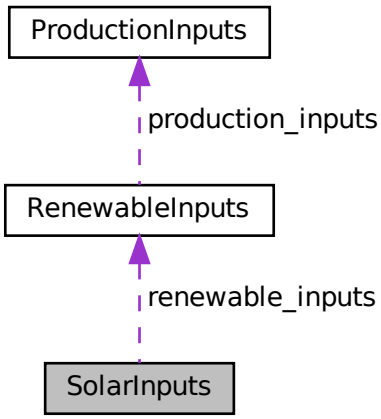
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

4.16 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.).

4.16.1 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.

4.16.2 Member Data Documentation

4.16.2.1 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].

4.16.2.2 derating

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

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

4.16.2.3 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].

4.16.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.16.2.5 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:

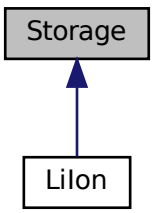
- header/Production/Renewable/Solar.h

4.17 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.

4.17.1 Detailed Description

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

4.17.2 Constructor & Destructor Documentation

4.17.2.1 Storage()

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

Constructor for the Storage class.

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

4.17.2.2 ~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:

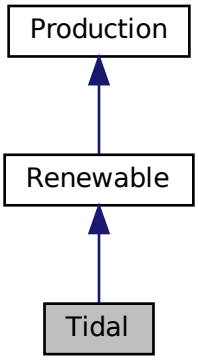
- header/Storage/Storage.h
- source/Storage/Storage.cpp

4.18 Tidal Class Reference

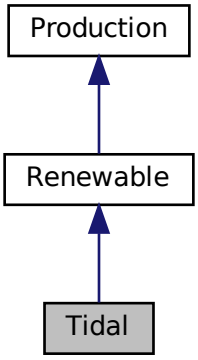
A derived class of the Renewable branch of Production which models tidal production.

`#include <Tidal.h>`

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- Tidal (void)

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

  Destructor for the Tidal class.

Additional Inherited Members

4.18.1 Detailed Description

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

4.18.2 Constructor & Destructor Documentation

4.18.2.1 Tidal()

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

Constructor for the Tidal class.

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

4.18.2.2 ~Tidal()

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

Destructor for the Tidal class.

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

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

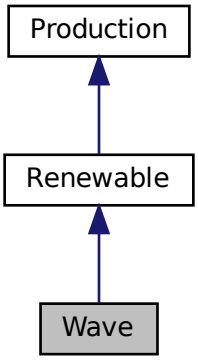
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.19 Wave Class Reference

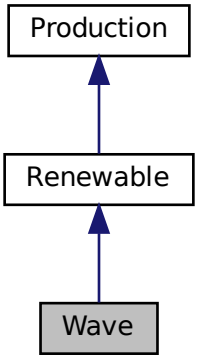
A derived class of the Renewable branch of Production which models wave production.

`#include <Wave.h>`

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

- Wave (void)

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

  Destructor for the Wave class.

Additional Inherited Members

4.19.1 Detailed Description

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

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Wave()

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

Constructor for the Wave class.

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

4.19.2.2 ∼Wave()

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

Destructor for the Wave class.

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

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

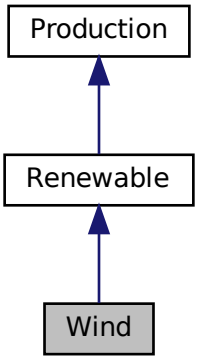
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

4.20 Wind Class Reference

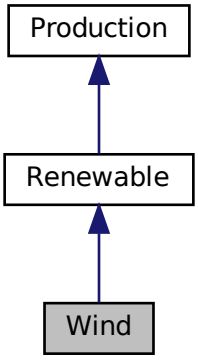
A derived class of the Renewable branch of Production which models wind production.

`#include <Wind.h>`

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- Wind (void)

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

  Destructor for the Wind class.

Additional Inherited Members

4.20.1 Detailed Description

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

4.20.2 Constructor & Destructor Documentation

4.20.2.1 Wind()

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

Constructor for the Wind class.

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

4.20.2.2 ∼Wind()

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

Destructor for the Wind class.

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

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

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

Chapter 5

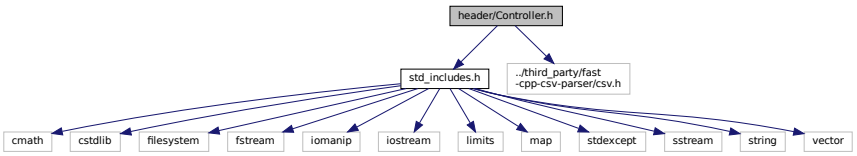
File Documentation

5.1 header/Controller.h File Reference

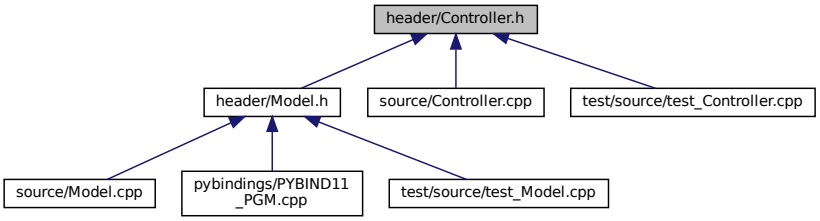
Header file the Controller class.

```
#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.

5.1.1 Detailed Description

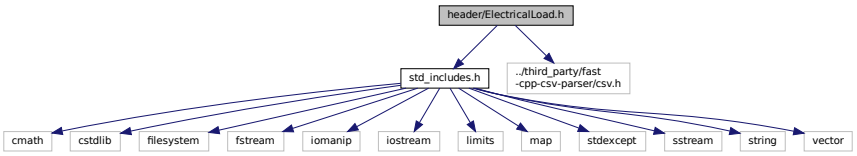
Header file the Controller class.

5.2 header/ElectricalLoad.h File Reference

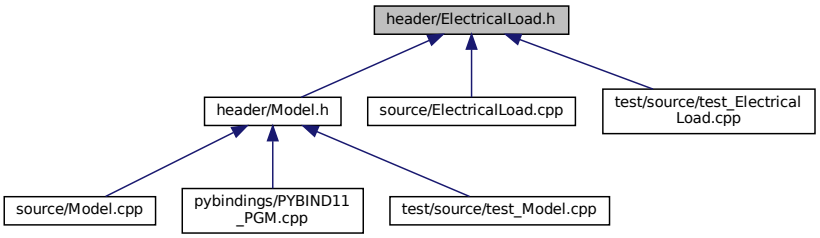
Header file the ElectricalLoad class.

```
#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

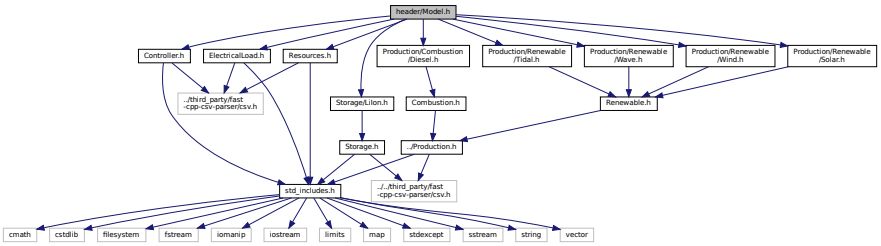
Header file the ElectricalLoad class.

5.3 header/Model.h File Reference

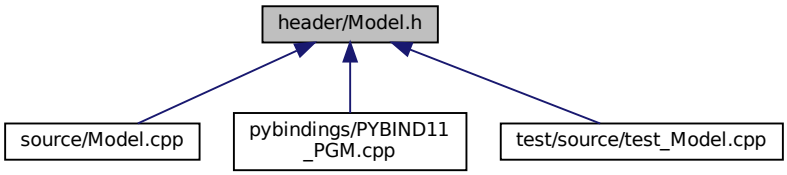
Header file the Model class.

```
#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

- 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.

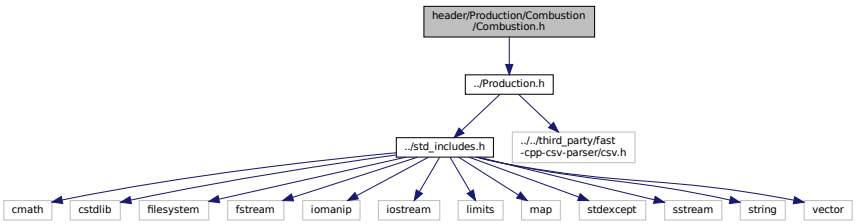
5.3.1 Detailed Description

Header file the Model class.

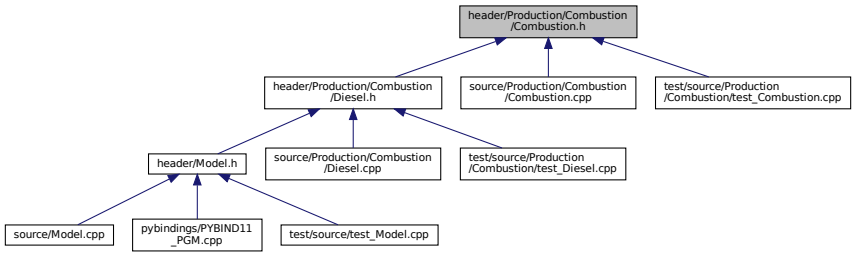
5.4 header/Production/Combustion/Combustion.h File Reference

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.

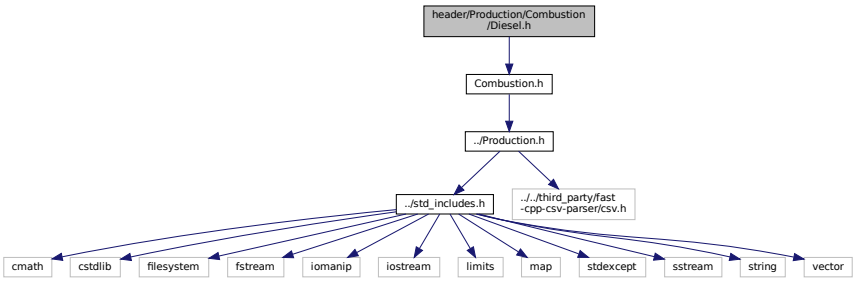
```
33                  {
34     DIESEL,
35     N_COMBUSTION_TYPES
36 };
```

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

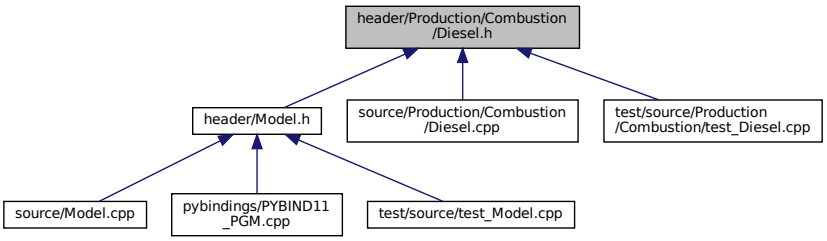
Header file the Diesel class.

```
#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

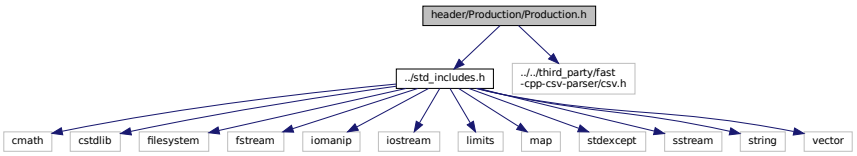
Header file the Diesel class.

5.6 header/Production/Production.h File Reference

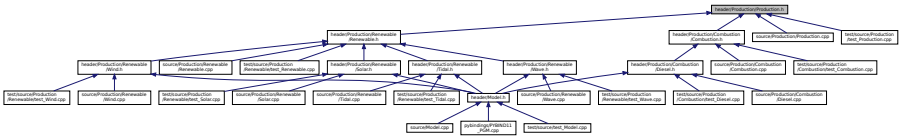
Header file the Production class.

```
#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

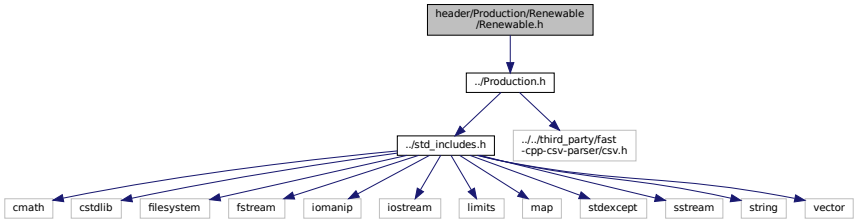
Header file the Production class.

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

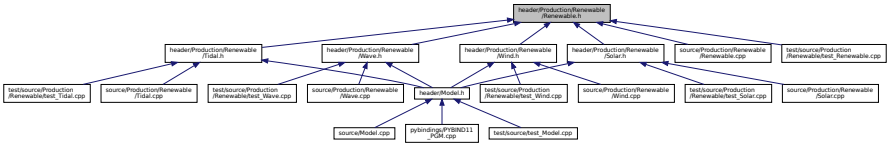
Header file the Renewable class.

```
#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

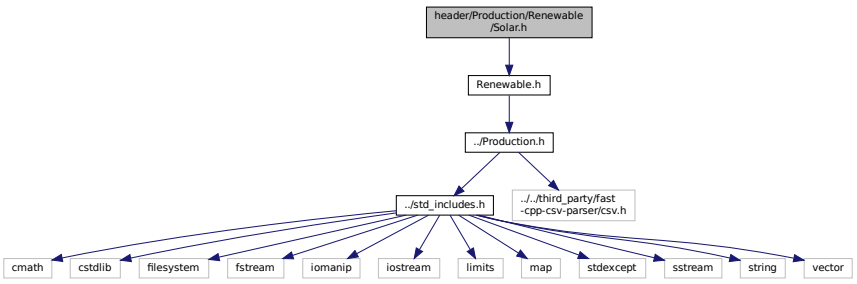
SOLAR	A solar photovoltaic (PV) array.
TIDAL	A tidal stream turbine (or tidal energy converter, TEC)
WAVE	A wave energy converter (WEC)
WIND	A wind turbine.
N_RENEWABLE_TYPES	A simple hack to get the number of elements in RenewableType.

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

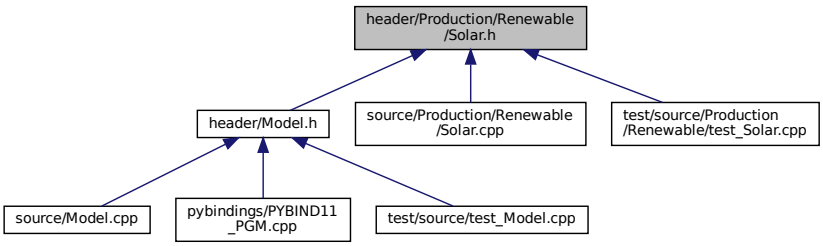
5.8 header/Production/Renewable/Solar.h File Reference

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

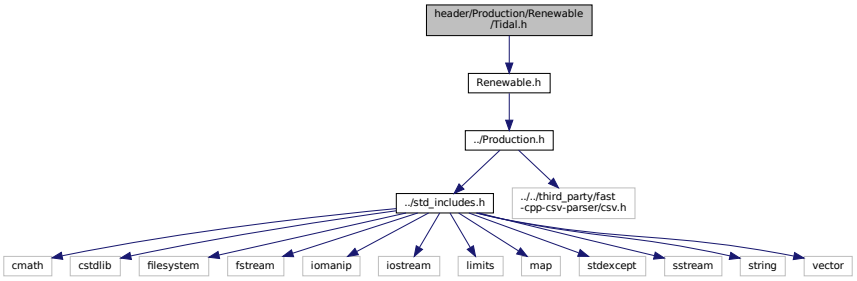
Header file the Solar class.

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

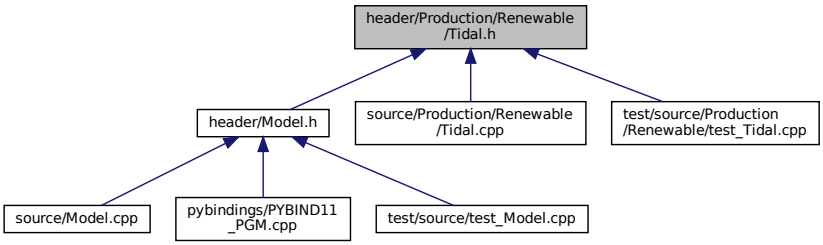
Header file the Tidal class.

```
#include "Renewable.h"
```

Include dependency graph for Tidal.h:



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



Classes

- class Tidal

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

5.9.1 Detailed Description

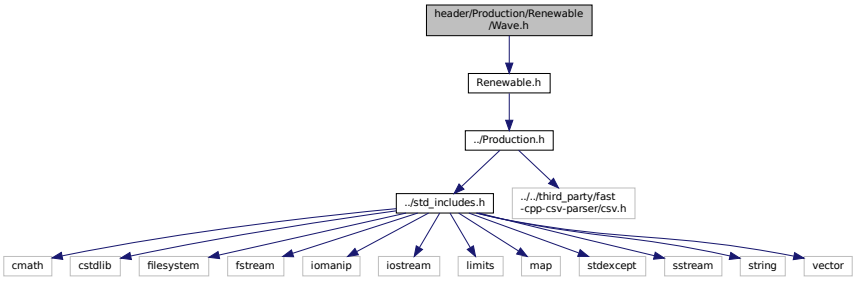
Header file the Tidal class.

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

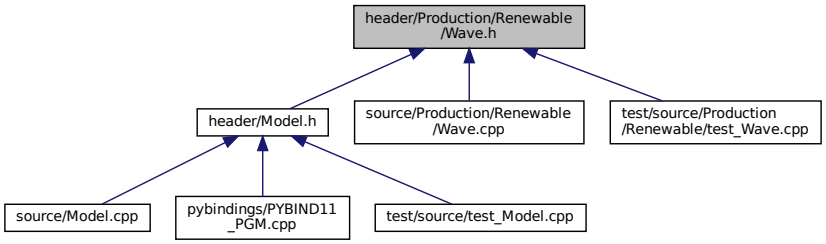
Header file the Wave class.

```
#include "Renewable.h"
```

Include dependency graph for Wave.h:



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



Classes

- class Wave

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

5.10.1 Detailed Description

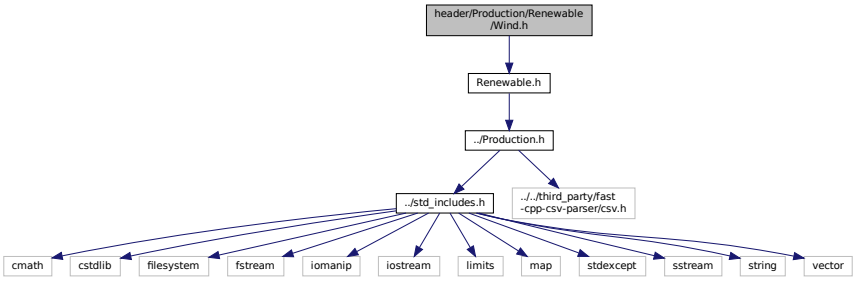
Header file the Wave class.

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

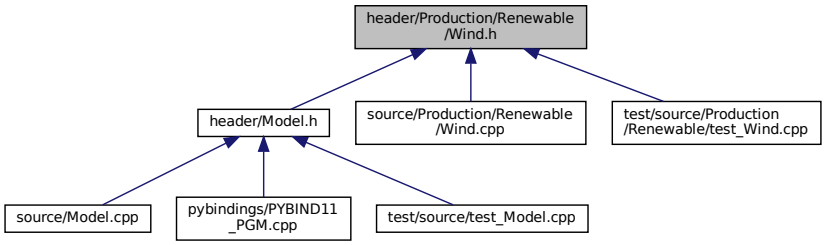
Header file the Wind class.

```
#include "Renewable.h"
```

Include dependency graph for Wind.h:



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



Classes

- class Wind

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

5.11.1 Detailed Description

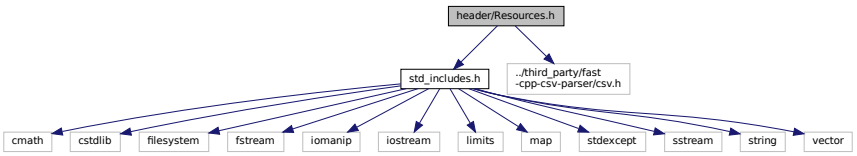
Header file the Wind class.

5.12 header/Resources.h File Reference

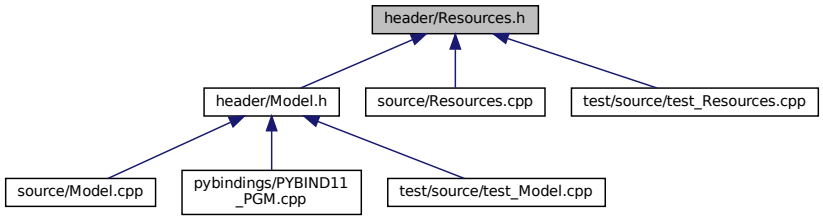
Header file the Resources class.

```
#include "std_includes.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.

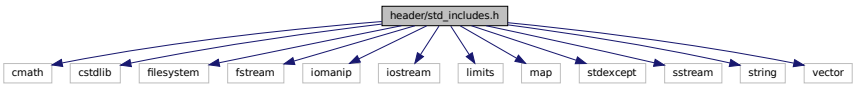
5.13 header/std_includes.h File Reference

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

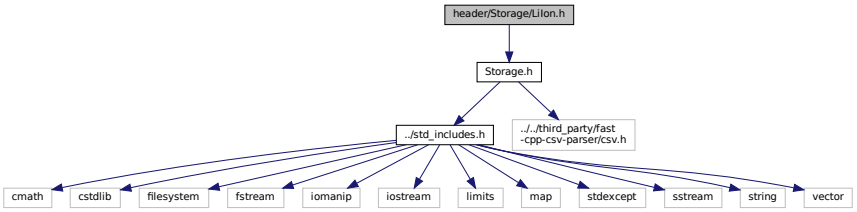
Header file which simply batches together the usual, standard includes.

5.14 header/Storage/LiIon.h File Reference

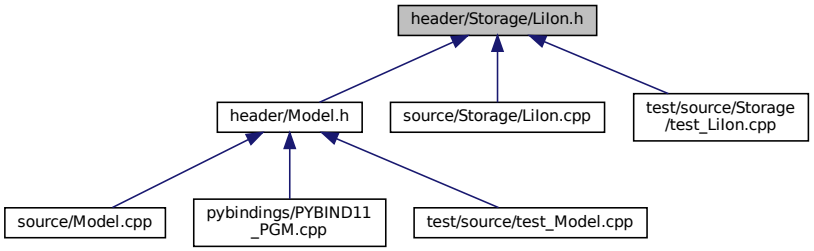
Header file the LiIon class.

```
#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

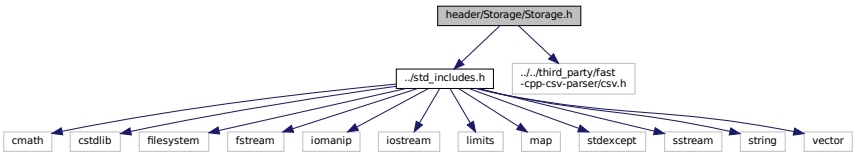
Header file the LiIon class.

5.15 header/Storage/Storage.h File Reference

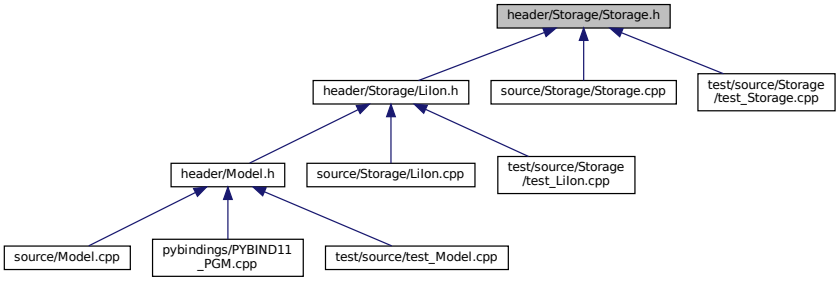
Header file the Storage class.

```
#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

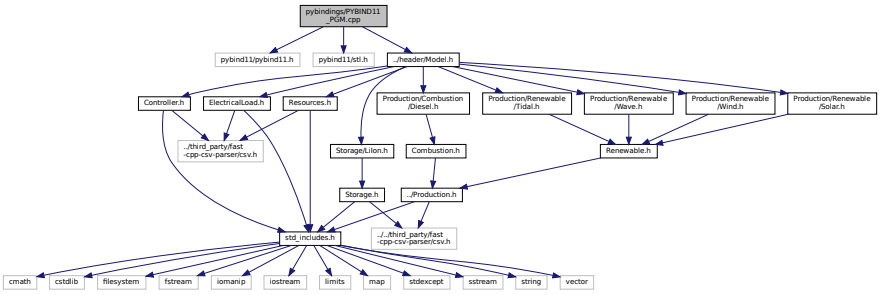
Header file the Storage class.

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

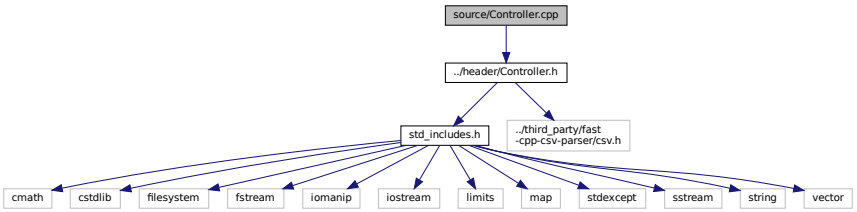
5.16.2.1 PYBIND11_MODULE()

```
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

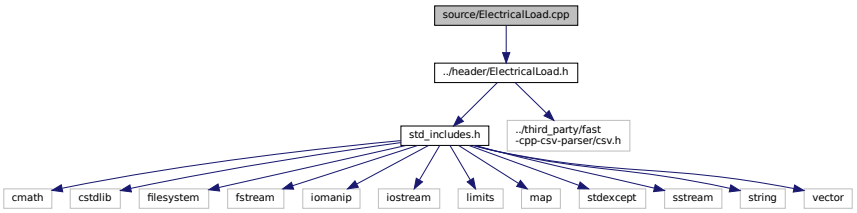
Implementation file for the Controller class.

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

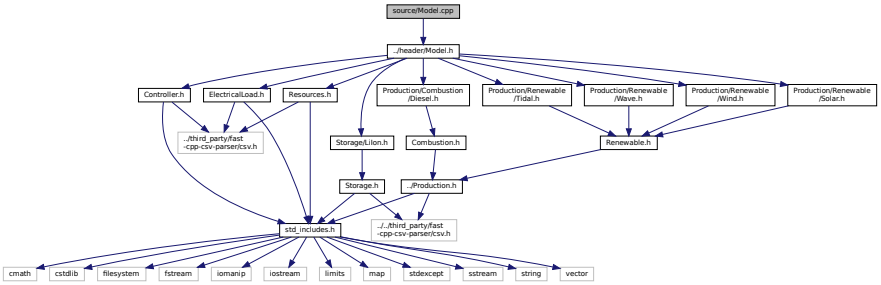
Implementation file for the ElectricalLoad class.

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

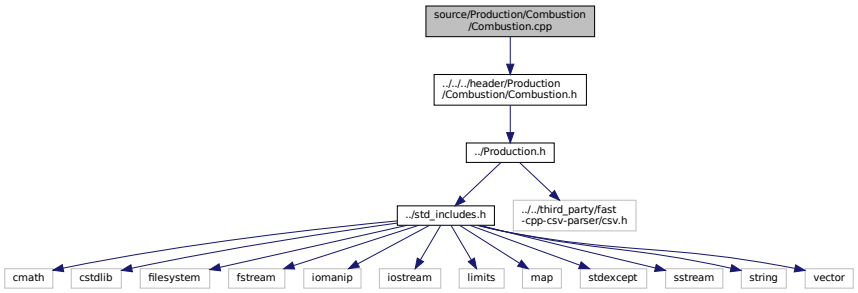
Implementation file for the Model class.

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

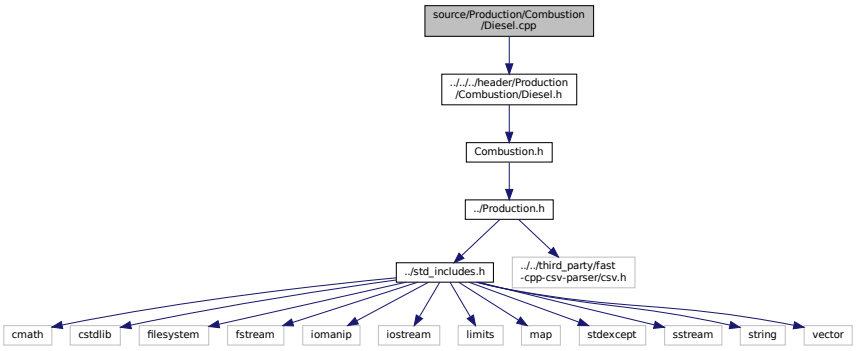
Implementation file for the Combustion class.

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

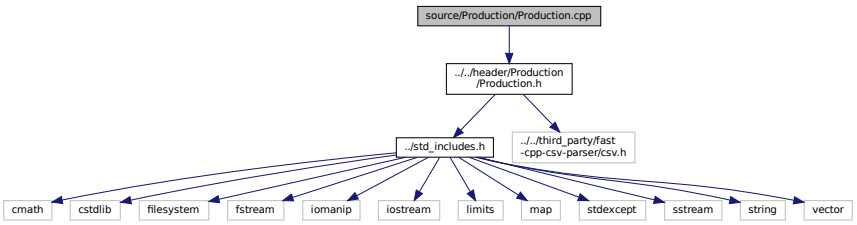
Implementation file for the Diesel class.

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

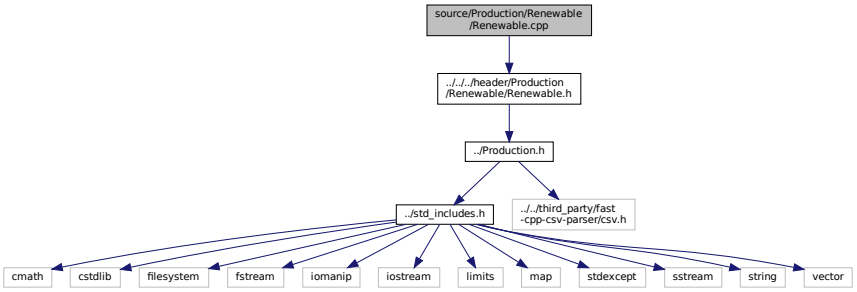
Implementation file for the Production class.

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

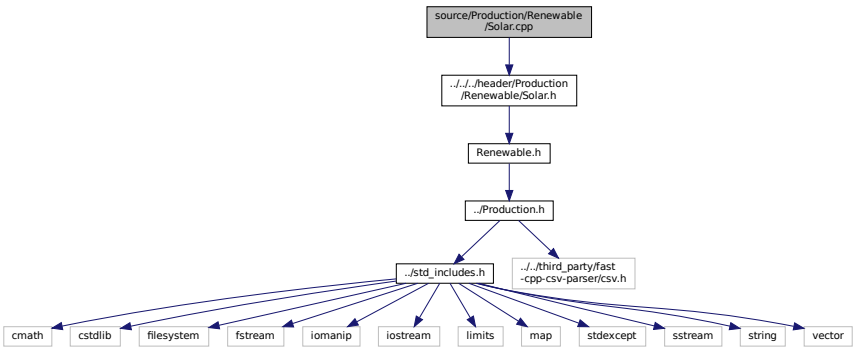
Implementation file for the Renewable class.

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

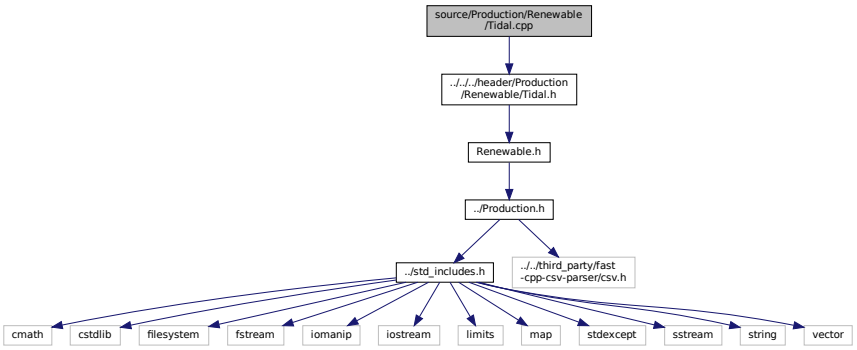
Implementation file for the Solar class.

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

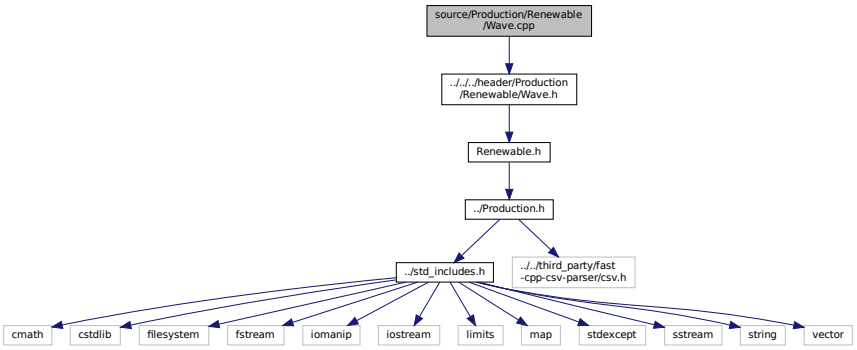
Implementation file for the Tidal class.

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

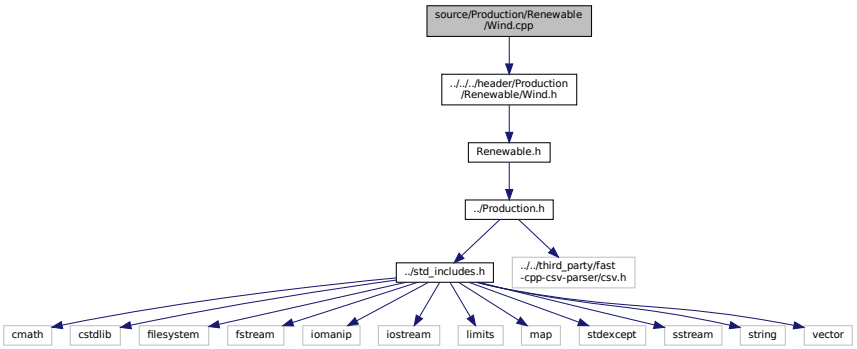
Implementation file for the Wave class.

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

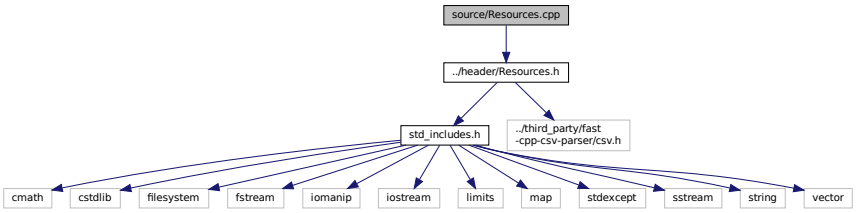
Implementation file for the Wind class.

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

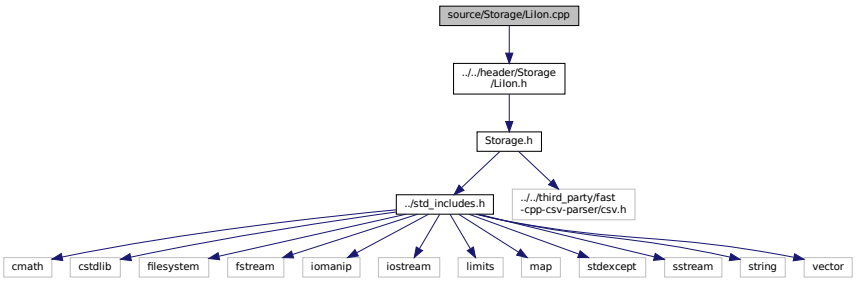
Implementation file for the Resources class.

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

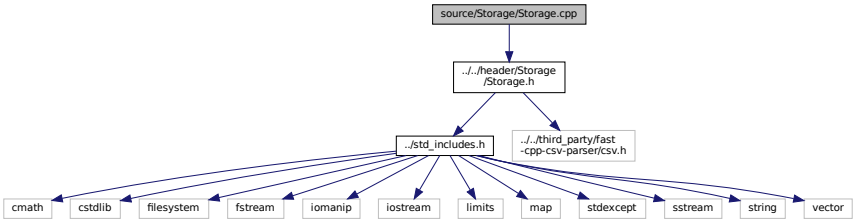
Implementation file for the LiIon class.

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.

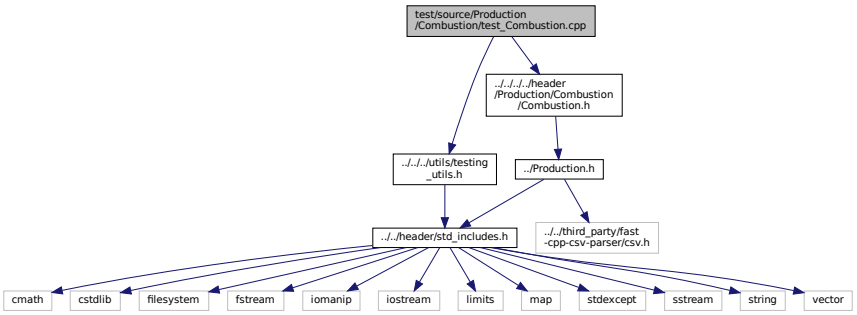
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

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 )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

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

    srand(time(NULL));


try {

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

CombustionInputs combustion_inputs;

Combustion test_combustion(8760, combustion_inputs);

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


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

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

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

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

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

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

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

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

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

testFloatEquals(
```

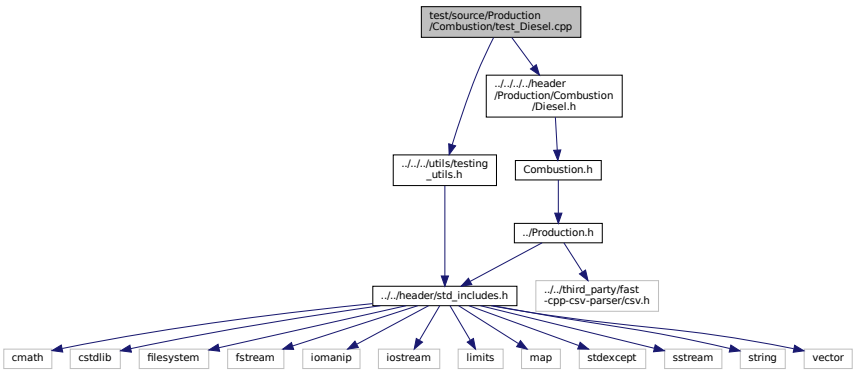
```
107     test_combustion.PM_emissions_vec_kg.size(),
108     8760,
109     __FILE__,
110     __LINE__
111 );
112
113 // ======== END ATTRIBUTES ============================================================ //
114
115 }   /* try */
116
117
118 catch (...) {
119     //...
120
121     printGold(" ..............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

5.32 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 testFloatEquals(
84     test_diesel_ptr->linear_fuel_slope_LkWh,
85     0.265675,
86     __FILE__,
87     __LINE__
88 );
89
90 testFloatEquals(
91     test_diesel_ptr->linear_fuel_intercept_LkWh,
92     0.026676,
93     __FILE__,
94     __LINE__
95 );
96
97 testFloatEquals(
```

```
98      test_diesel_ptr->capital_cost,
99      67846.467018,
100     __FILE__,
101     __LINE__
102 );
103
104 testFloatEquals(
105     test_diesel_ptr->operation_maintenance_cost_kWh,
106     0.038027,
107     __FILE__,
108     __LINE__
109 );
110
111 testFloatEquals(
112     ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
113     0.2,
114     __FILE__,
115     __LINE__
116 );
117
118 testFloatEquals(
119     ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
120     4,
121     __FILE__,
122     __LINE__
123 );
124
125 testFloatEquals(
126     test_diesel_ptr->replace_running_hrs,
127     30000,
128     __FILE__,
129     __LINE__
130 );
131
132 // ======== END ATTRIBUTES ============================================================ //
133
134
135
136 // ======== METHODS ================================================================== //
137
138 //  test capacity constraint
139 testFloatEquals(
140     test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
141     test_diesel_ptr->capacity_kW,
142     __FILE__,
143     __LINE__
144 );
145
146 //  test minimum load ratio constraint
147 testFloatEquals(
148     test_diesel_ptr->requestProductionkW(
149         0,
150         1,
151         0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
152             test_diesel_ptr->capacity_kW
153     ),
154     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
155     __FILE__,
156     __LINE__
157 );
158
159 //  test commit()
160 std::vector<double> dt_vec_hrs (48, 1);
161
162 std::vector<double> load_vec_kW = {
163     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
164     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
165     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
166     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
167 };
168
169 std::vector<bool> expected_is_running_vec = {
170     1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
171     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
172     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
173     1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
174 };
175
176 double roll = 0;
177 double production_kW = 0;
178 double load_kW = 0;
179
180 for (int i = 0; i < 48; i++) {
181     roll = (double)rand() / RAND_MAX;
182
183     if (roll >= 0.95) {
184         roll = 1.25;
```

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

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

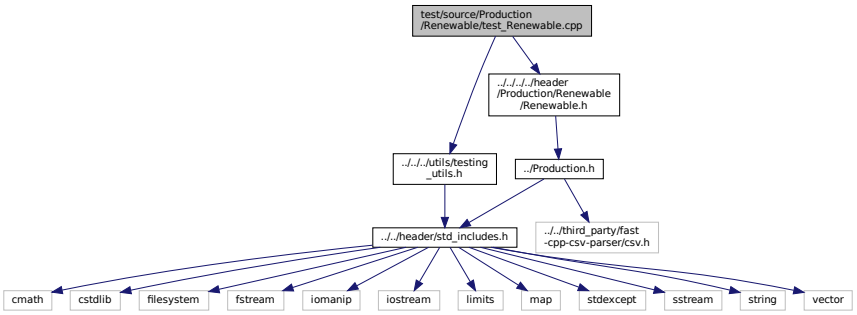
```
359         testFloatEquals(
360             test_diesel_ptr->NOx_emissions_vec_kg[i],
361             0,
362             __FILE__,
363             __LINE__
364         );
365 
366         testFloatEquals(
367             test_diesel_ptr->SOx_emissions_vec_kg[i],
368             0,
369             __FILE__,
370             __LINE__
371         );
372 
373         testFloatEquals(
374             test_diesel_ptr->CH4_emissions_vec_kg[i],
375             0,
376             __FILE__,
377             __LINE__
378         );
379 
380         testFloatEquals(
381             test_diesel_ptr->PM_emissions_vec_kg[i],
382             0,
383             __FILE__,
384             __LINE__
385         );
386     }
387 }
388 
389 // ======== END METHODS ============================================================ //
390 
391 }   /* try */
392 
393 
394 catch (...) {
395     delete test_diesel_ptr;
396 
397     printGold(" ... ");
398     printRed("FAIL");
399     std::cout « std::endl;
400     throw;
401 }
402 
403 
404 delete test_diesel_ptr;
405 
406 printGold(" ... ");
407 printGreen("PASS");
408 std::cout « std::endl;
409 return 0;
410 
411 }   /* 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 )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ======== CONSTRUCTION ========================================================== //
40
41 RenewableInputs renewable_inputs;
42
43 Renewable test_renewable(8760, renewable_inputs);
44
45 // ======== END CONSTRUCTION ====================================================== //
46
47
48
49 // ======== ATTRIBUTES ============================================================ //
50
```

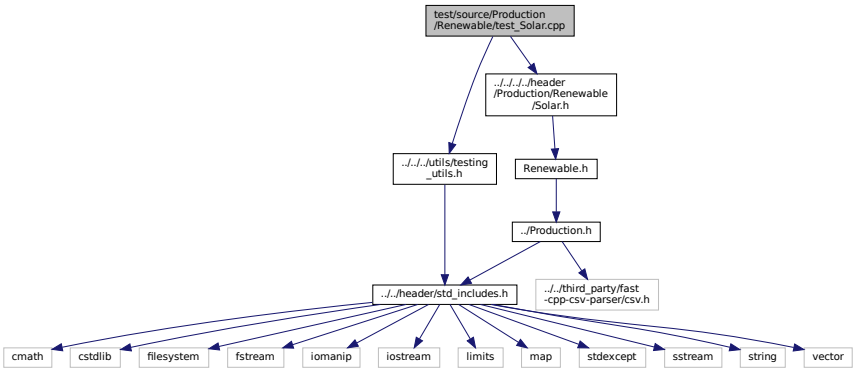
```
51 testTruth(
52     not renewable_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56
57 // ======== END ATTRIBUTES ============================================================ //
58
59 }   /* try */
60
61
62 catch (...) {
63     //...
64
65     printGold(" ...............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

5.34 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 testFloatEquals(
82     test_solar_ptr->capital_cost,
83     3000 * 100,
84     __FILE__,
85     __LINE__
86 );
87
88 testFloatEquals(
89     test_solar_ptr->operation_maintenance_cost_kWh,
90     0.01,
91     __FILE__,
92     __LINE__
93 );
94
95 // ======== END ATTRIBUTES ========================================================== //
96
97
```

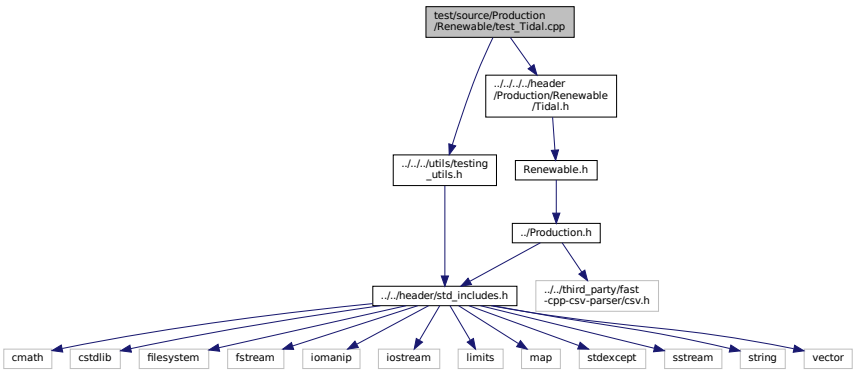
```
98
99 // ======== METHODS ========================================================== //
100
101 //...
102
103 // ======== END METHODS ====================================================== //
104
105 }   /* try */
106
107
108 catch (...) {
109     delete test_solar_ptr;
110
111     printGold(" ..... ");
112     printRed("FAIL");
113     std::cout « std::endl;
114     throw;
115 }
116
117
118 delete test_solar_ptr;
119
120 printGold(" ..... ");
121 printGreen("PASS");
122 std::cout « std::endl;
123 return 0;
124 }   /* 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()

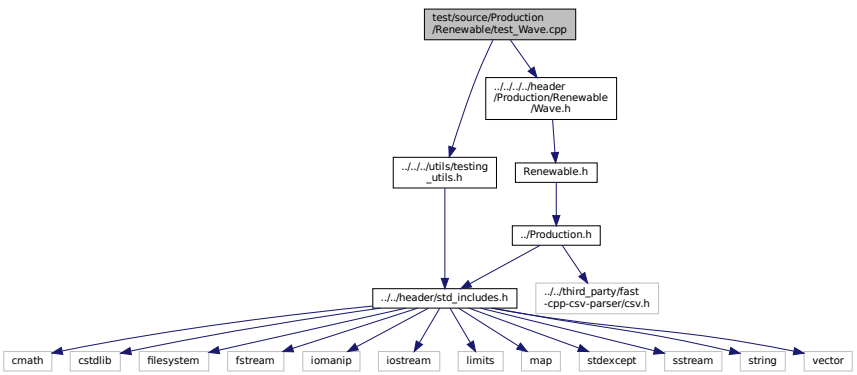
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Tidal");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
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() */
```

5.36 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()

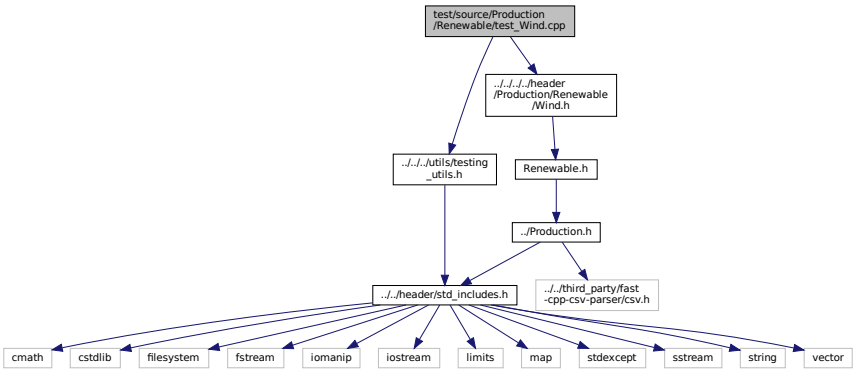
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wave");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
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() */
```

5.37 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 )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wind");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
42         //...
43
44         printGold(" ..... ");
45         printRed("FAIL");
```

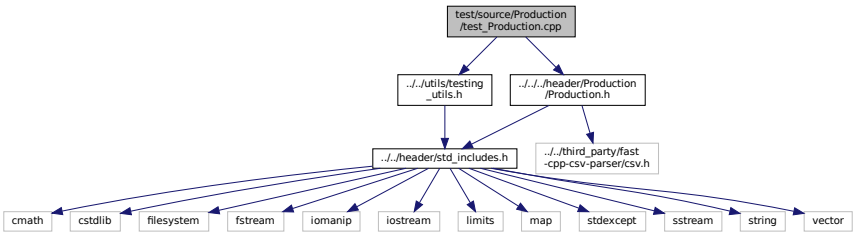
```
46          std::cout « std::endl;
47          throw;
48      }
49
50
51      printGold(" ..... ");
52      printGreen("PASS");
53      std::cout « std::endl;
54      return 0;
55  }   /* main() */
```

5.38 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
```

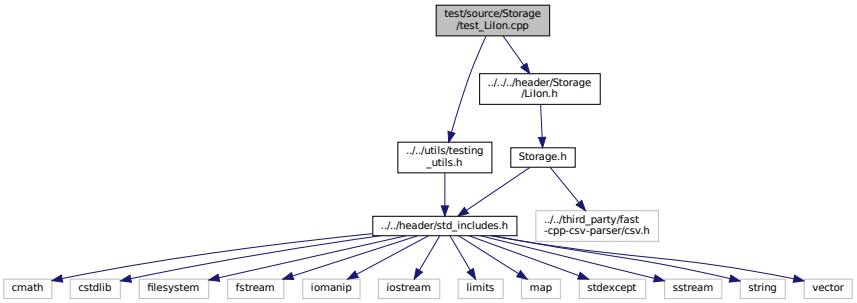
```
107 testFloatEquals(
108     test_production.production_vec_kW.size(),
109     8760,
110     __FILE__,
111     __LINE__
112 );
113
114 testFloatEquals(
115     test_production.dispatch_vec_kW.size(),
116     8760,
117     __FILE__,
118     __LINE__
119 );
120
121 testFloatEquals(
122     test_production.storage_vec_kW.size(),
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()

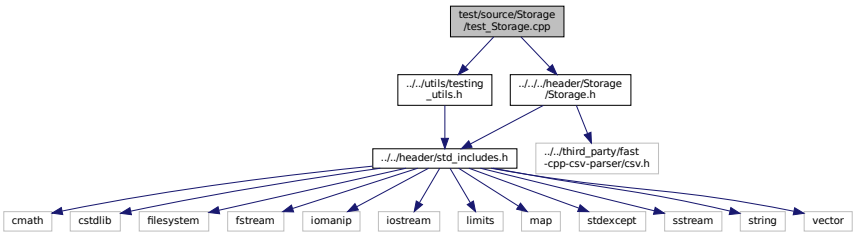
```
int main (
            int argc,
            char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Storage <-- LiIon");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
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(" ...................... ");
```

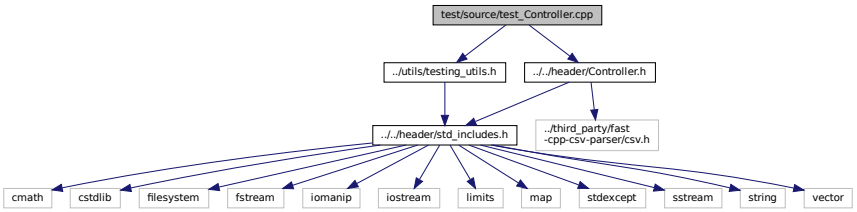
```
45          printRed("FAIL");
46          std::cout « std::endl;
47          throw;
48      }
49
50
51      printGold(" ............................... ");
52      printGreen("PASS");
53      std::cout « std::endl;
54      return 0;
55  }   /* main() */
```

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()

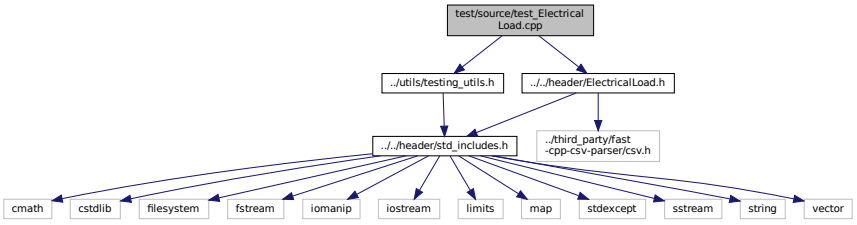
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Controller");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
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()

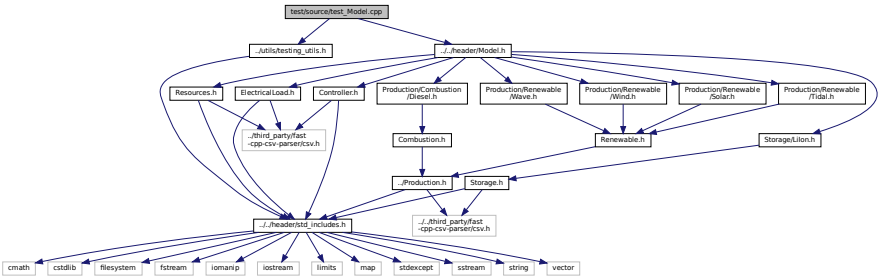
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting ElectricalLoad");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
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() */
```

5.43 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()

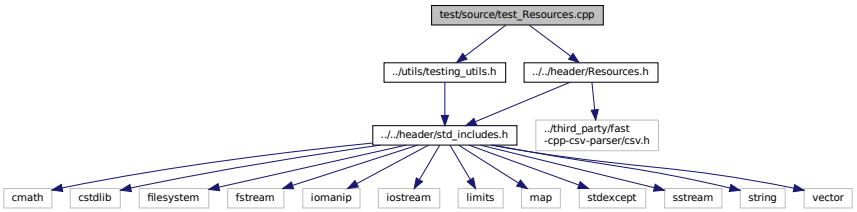
```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Model");
33
34     srand(time(NULL));
35
36
37     try {
38         //...
39     }
40
41     catch (...) {
42         //...
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() */
```

5.44 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

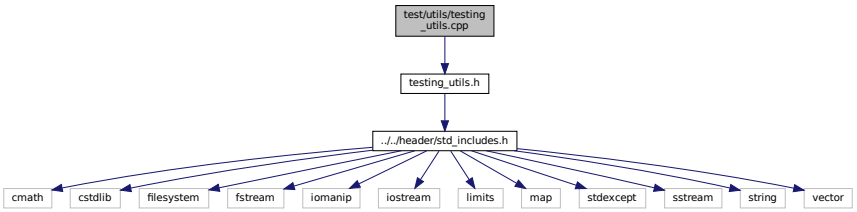
5.44.2.1 main()

```
int main (
              int argc,
              char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif  /* _WIN32 */
31
32     printGold("\tTesting Resources");
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() */
```

5.45 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.
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__").

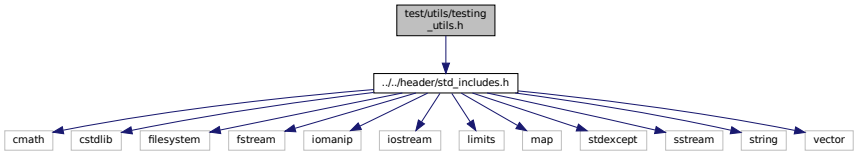
```
392 {
393     if (statement) {
394         return;
395     }
396 
397     std::string error_str = "ERROR: testTruth():\t in ";
398     error_str += file;
399     error_str += "\tline ";
400     error_str += std::to_string(line);
401     error_str += ":\t\n";
402     error_str += "Given statement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

Parameters

y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142 
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154 
155     #ifdef _WIN32
156         std::cout << error_str << std::endl;
157     #endif
158 
159     throw std::runtime_error(error_str);
160     return;
161 }   /* testFloatEquals() */
```

5.46.3.6 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