# PGMcpp: PRIMED Grid Modelling (in C++)

Generated by Doxygen 1.9.1

# Chapter 1

# Hierarchical Index

## 1.1 Class Hierarchy

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

# Chapter 2

# Class Index

## 2.1 Class List

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

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all files with brief descriptions:

# Chapter 4

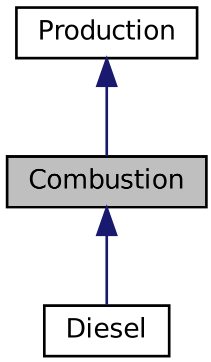
# Class Documentation

## 4.1 Combustion Class Reference

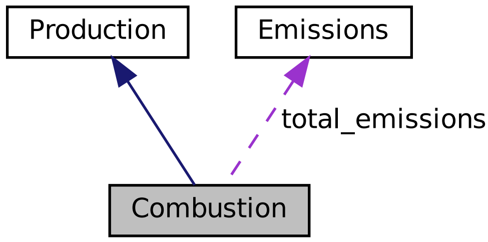
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

```
#include <Combustion.h>
```

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:




## Public Member Functions

- Combustion (void)

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

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

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

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

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

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

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

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

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

  *Destructor for the Combustion class.*

## Public Attributes

- CombustionType type

  *The type (CombustionType) of the asset.*
- double fuel_cost_L

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

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

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

- double linear_fuel_slope_LkWh

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

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

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

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

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

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

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

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

  *The total fuel consumed [L] over a model run.*
- Emissions total_emissions

  *An Emissions structure for holding total emissions [kg].*
- std::vector< double > fuel_consumption_vec_L

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

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

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

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

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

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

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

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

## Private Member Functions

- void __checkInputs (CombustionInputs)

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

### 4.1.1 Detailed Description

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

### 4.1.2 Constructor & Destructor Documentation

#### 4.1.2.1 Combustion() [1/2]

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

Constructor (dummy) for the Combustion class.

```
{
    return;
}   /* Combustion() */
```

#### 4.1.2.2 Combustion() [2/2]

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

Constructor (intended) for the Combustion class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *combustion_inputs* | A structure of Combustion constructor inputs. |

```
  :
Production(
    n_points,
    n_years,
    combustion_inputs.production_inputs
)
{
    //  1. check inputs
    this->__checkInputs(combustion_inputs);

    //  2. set attributes
    this->fuel_cost_L = 0;
    this->nominal_fuel_escalation_annual =
        combustion_inputs.nominal_fuel_escalation_annual;

    this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
        combustion_inputs.nominal_fuel_escalation_annual,
        combustion_inputs.production_inputs.nominal_discount_annual
    );

    this->linear_fuel_slope_LkWh = 0;
    this->linear_fuel_intercept_LkWh = 0;

    this->CO2_emissions_intensity_kgL = 0;
```

```
    this->CO_emissions_intensity_kgL = 0;
    this->NOx_emissions_intensity_kgL = 0;
    this->SOx_emissions_intensity_kgL = 0;
    this->CH4_emissions_intensity_kgL = 0;
    this->PM_emissions_intensity_kgL = 0;

    this->total_fuel_consumed_L = 0;

    this->fuel_consumption_vec_L.resize(this->n_points, 0);
    this->fuel_cost_vec.resize(this->n_points, 0);

    this->CO2_emissions_vec_kg.resize(this->n_points, 0);
    this->CO_emissions_vec_kg.resize(this->n_points, 0);
    this->NOx_emissions_vec_kg.resize(this->n_points, 0);
    this->SOx_emissions_vec_kg.resize(this->n_points, 0);
    this->CH4_emissions_vec_kg.resize(this->n_points, 0);
    this->PM_emissions_vec_kg.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Combustion object constructed at " « this « std::endl;
    }

    return;
}   /* Combustion() */
```

#### 4.1.2.3 ∼Combustion()

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

Destructor for the Combustion class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Combustion object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Combustion() */
```

### 4.1.3 Member Function Documentation

#### 4.1.3.1 __checkInputs()

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

Helper method to check inputs to the Combustion constructor.

**Parameters**

| | |
|---|---|
| *combustion_inputs* | A structure of Combustion constructor inputs. |

```
{
    // ...

    return;
}   /* __checkInputs() */
```

#### 4.1.3.2 __writeSummary()

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

Reimplemented in Diesel.

```
{return;}
```

#### 4.1.3.3 __writeTimeSeries()

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

Reimplemented in Diesel.

```
            {return;}
```

#### 4.1.3.4 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Production.

Reimplemented in Diesel.

```
{
    //  1. invoke base class method
    load_kW = Production :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
```

```
    );


    if (this->is_running) {
        //  2. compute and record fuel consumption
        double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
        this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;

        //  3. compute and record emissions
        Emissions emissions = this->getEmissionskg(fuel_consumed_L);
        this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
        this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
        this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
        this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
        this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
        this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;

        //  4. incur fuel costs
        this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
    }

    return load_kW;
}   /* commit() */
```

#### 4.1.3.5 computeEconomics()

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

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

Ref: HOMER [2023b]

**Parameters**

| | |
|---|---|
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |

Reimplemented from Production.

```
{
    //  1. account for fuel costs in net present cost
    double t_hrs = 0;
    double real_fuel_escalation_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_fuel_escalation_scalar = 1.0 / pow(
            1 + this->real_fuel_escalation_annual,
            t_hrs / 8760
        );

        this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
    }

    //  2. invoke base class method
    Production :: computeEconomics(time_vec_hrs_ptr);

    return;
}   /* computeEconomics() */
```

#### 4.1.3.6 computeFuelAndEmissions()

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

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

```
{
    for (int i = 0; i < n_points; i++) {
        this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];

        this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
        this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
        this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
        this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
        this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
        this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
    }

    return;
}   /* computeFuelAndEmissions() */
```

#### 4.1.3.7 getEmissionskg()

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

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

**Parameters**

| | |
|---|---|
| *fuel_consumed←_L* | The volume of fuel consumed [L]. |

**Returns**

A structure containing the mass spectrum of resulting emissions.

```
                                                        {
    Emissions emissions;

    emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
    emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
    emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
    emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
    emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
    emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;

    return emissions;
}   /* getEmissionskg() */
```

#### 4.1.3.8 getFuelConsumptionL()

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

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

**Parameters**

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

**Returns**

The volume of fuel consumed [L].

```
{
    double fuel_consumed_L = (
        this->linear_fuel_slope_LkWh * production_kW +
        this->linear_fuel_intercept_LkWh * this->capacity_kW
    ) * dt_hrs;

    return fuel_consumed_L;
} /* getFuelConsumptionL() */
```

#### 4.1.3.9 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Production.

Reimplemented in Diesel.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Production :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

#### 4.1.3.10 requestProductionkW()

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

Reimplemented in Diesel.

```
{return 0;}
```

#### 4.1.3.11 writeResults()

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

Method which writes Combustion results to an output directory.

**Parameters**

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

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Production/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += "Combustion/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->capacity_kW)));
    write_path += "kW_idx";
    write_path += std::to_string(combustion_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
    }

    return;
}   /* writeResults() */
```

### 4.1.4 Member Data Documentation

#### 4.1.4.1 CH4_emissions_intensity_kgL

`double Combustion::CH4_emissions_intensity_kgL`

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

#### 4.1.4.2 CH4_emissions_vec_kg

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

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

#### 4.1.4.3 CO2_emissions_intensity_kgL

```
double Combustion::CO2_emissions_intensity_kgL
```

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

#### 4.1.4.4 CO2_emissions_vec_kg

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

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

#### 4.1.4.5 CO_emissions_intensity_kgL

```
double Combustion::CO_emissions_intensity_kgL
```

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

#### 4.1.4.6 CO_emissions_vec_kg

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

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

#### 4.1.4.7 fuel_consumption_vec_L

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

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

#### 4.1.4.8 fuel_cost_L

```
double Combustion::fuel_cost_L
```

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

#### 4.1.4.9 fuel_cost_vec

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

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

#### 4.1.4.10 linear_fuel_intercept_LkWh

```
double Combustion::linear_fuel_intercept_LkWh
```

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

#### 4.1.4.11 linear_fuel_slope_LkWh

```
double Combustion::linear_fuel_slope_LkWh
```

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

#### 4.1.4.12 nominal_fuel_escalation_annual

```
double Combustion::nominal_fuel_escalation_annual
```

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

#### 4.1.4.13 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

#### 4.1.4.14 NOx_emissions_vec_kg

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

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

#### 4.1.4.15 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

#### 4.1.4.16 PM_emissions_vec_kg

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

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

#### 4.1.4.17 real_fuel_escalation_annual

```
double Combustion::real_fuel_escalation_annual
```

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

#### 4.1.4.18 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

#### 4.1.4.19 SOx_emissions_vec_kg

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

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

#### 4.1.4.20 total_emissions

```
Emissions Combustion::total_emissions
```

An Emissions structure for holding total emissions [kg].

#### 4.1.4.21 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

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

#### 4.1.4.22 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

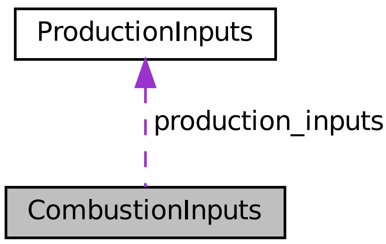
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

## 4.2 CombustionInputs Struct Reference

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

```
#include <Combustion.h>
```

Collaboration diagram for CombustionInputs:




### Public Attributes

- ProductionInputs production_inputs

  *An encapsulated ProductionInputs instance.*
- double nominal_fuel_escalation_annual = 0.05

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

### 4.2.1 Detailed Description

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

### 4.2.2 Member Data Documentation

#### 4.2.2.1 nominal_fuel_escalation_annual

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

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

#### 4.2.2.2 production_inputs

```
ProductionInputs CombustionInputs::production_inputs
```

An encapsulated ProductionInputs instance.

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

- header/Production/Combustion/Combustion.h

## 4.3 Controller Class Reference

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

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

### Public Member Functions

- Controller (void)

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

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

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

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

  *Destructor for the Controller class.*

## Public Attributes

- ControlMode control_mode

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

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

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

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

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

## Private Member Functions

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

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

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

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

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

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

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

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

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

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

  *Helper method to handle the optimal dispatch of Combustion assets. Dispatches for 1.2x the received net load, so as to ensure a "20% spinning reserve". Dispatches a minimum number of Combustion assets, which then share the load proportional to their rated capacities.*
- double __handleStorageDischarging (int, double, double, std::list< Storage ∗ >)

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

### 4.3.1 Detailed Description

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

## 4.3.2 Constructor & Destructor Documentation

### 4.3.2.1 Controller()

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

Constructor for the Controller class.

```
{
    return;
}   /* Controller() */
```

### 4.3.2.2 ∼Controller()

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

Destructor for the Controller class.

```
{
    this->clear();

    return;
}   /* ~Controller() */
```

## 4.3.3 Member Function Documentation

### 4.3.3.1 __applyCycleChargingControl_CHARGING()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *electrical_load_ptr* | A pointer to the ElectricalLoad component of the Model. |
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |
| *renewable_ptr_vec_ptr* | A pointer to the Renewable pointer vector of the Model. |
| *storage_ptr_vec_ptr* | A pointer to the Storage pointer vector of the Model. |

```
{
    // 1. default to load following
    this->__applyLoadFollowingControl_CHARGING(
        timestep,
        electrical_load_ptr,
        combustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr,
        storage_ptr_vec_ptr
    );

    return;
}   /* __applyCycleChargingControl_CHARGING() */
```

#### 4.3.3.2 __applyCycleChargingControl_DISCHARGING()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *electrical_load_ptr* | A pointer to the ElectricalLoad component of the Model. |
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |
| *renewable_ptr_vec_ptr* | A pointer to the Renewable pointer vector of the Model. |
| *storage_ptr_vec_ptr* | A pointer to the Storage pointer vector of the Model. |

curtailment

```
{
    // 1. get dt_hrs, net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = this->net_load_vec_kW[timestep];

    // 2. partition Storage assets into depleted and non-depleted
    std::list<Storage*> depleted_storage_ptr_list;
    std::list<Storage*> nondepleted_storage_ptr_list;

    Storage* storage_ptr;
    for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
        storage_ptr = storage_ptr_vec_ptr->at(i);

        if (storage_ptr->is_depleted) {
            depleted_storage_ptr_list.push_back(storage_ptr);
        }

        else {
            nondepleted_storage_ptr_list.push_back(storage_ptr);
        }
    }

    // 3. discharge non-depleted storage assets
    net_load_kW = this->__handleStorageDischarging(
        timestep,
        dt_hrs,
        net_load_kW,
        nondepleted_storage_ptr_list
    );

    // 4. request optimal production from all Combustion assets
    //    default to load following if no depleted storage
    if (depleted_storage_ptr_list.empty()) {
```

```
        net_load_kW = this->__handleCombustionDispatch(
            timestep,
            dt_hrs,
            net_load_kW,
            combustion_ptr_vec_ptr,
            false   // is_cycle_charging
        );
    }

    else {
        net_load_kW = this->__handleCombustionDispatch(
            timestep,
            dt_hrs,
            net_load_kW,
            combustion_ptr_vec_ptr,
            true    // is_cycle_charging
        );
    }

    //  5. attempt to charge depleted Storage assets using any and all available
    //     charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        depleted_storage_ptr_list,
        combustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    //  6. record any missed load
    if (net_load_kW > 1e-6) {
        this->missed_load_vec_kW[timestep] = net_load_kW;
    }

    return;
}   /* __applyCycleChargingControl_DISCHARGING() */
```

#### 4.3.3.3 __applyLoadFollowingControl_CHARGING()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *electrical_load_ptr* | A pointer to the ElectricalLoad component of the Model. |
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |
| *renewable_ptr_vec_ptr* | A pointer to the Renewable pointer vector of the Model. |
| *storage_ptr_vec_ptr* | A pointer to the Storage pointer vector of the Model. |

```
{
    //  1. get dt_hrs, set net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = 0;

    //  2. request zero production from all Combustion assets
    this->__handleCombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        combustion_ptr_vec_ptr,
        false   // is_cycle_charging
    );
```

```
    //  3. attempt to charge all Storage assets using any and all available curtailment
    //     charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        storage_ptr_vec_ptr,
        combustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    return;
}   /* __applyLoadFollowingControl_CHARGING() */
```

#### 4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *electrical_load_ptr* | A pointer to the ElectricalLoad component of the Model. |
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |
| *renewable_ptr_vec_ptr* | A pointer to the Renewable pointer vector of the Model. |
| *storage_ptr_vec_ptr* | A pointer to the Storage pointer vector of the Model. |

curtailment

```
{
    //  1. get dt_hrs, net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = this->net_load_vec_kW[timestep];

    //  2. partition Storage assets into depleted and non-depleted
    std::list<Storage*> depleted_storage_ptr_list;
    std::list<Storage*> nondepleted_storage_ptr_list;

    Storage* storage_ptr;
    for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
        storage_ptr = storage_ptr_vec_ptr->at(i);

        if (storage_ptr->is_depleted) {
            depleted_storage_ptr_list.push_back(storage_ptr);
        }

        else {
            nondepleted_storage_ptr_list.push_back(storage_ptr);
        }
    }

    //  3. discharge non-depleted storage assets
    net_load_kW = this->__handleStorageDischarging(
        timestep,
        dt_hrs,
        net_load_kW,
        nondepleted_storage_ptr_list
    );

    //  4. request optimal production from all Combustion assets
    net_load_kW = this->__handleCombustionDispatch(
        timestep,
```

```
        dt_hrs,
        net_load_kW,
        combustion_ptr_vec_ptr,
        false   // is_cycle_charging
    );

    //  5. attempt to charge depleted Storage assets using any and all available
    //      charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        depleted_storage_ptr_list,
        combustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    //  6. record any missed load
    if (net_load_kW > 1e-6) {
        this->missed_load_vec_kW[timestep] = net_load_kW;
    }

    return;
}   /* __applyLoadFollowingControl_DISCHARGING() */
```

#### 4.3.3.5 __computeNetLoad()

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

Helper method to compute and populate the net load vector.

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

**Parameters**

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

```
{
    //  1. init
    this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
    this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);

    //  2. populate net load vector
    double dt_hrs = 0;
    double load_kW = 0;
    double net_load_kW = 0;
    double production_kW = 0;

    Renewable* renewable_ptr;

    for (int i = 0; i < electrical_load_ptr->n_points; i++) {
        dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
        load_kW = electrical_load_ptr->load_vec_kW[i];
        net_load_kW = load_kW;

        for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {
            renewable_ptr = renewable_ptr_vec_ptr->at(j);

            production_kW = this->__getRenewableProduction(
                i,
                dt_hrs,
                renewable_ptr,
                resources_ptr
            );
```

```
            load_kW = renewable_ptr->commit(
                i,
                dt_hrs,
                production_kW,
                load_kW
            );

            net_load_kW -= production_kW;
        }

        this->net_load_vec_kW[i] = net_load_kW;
    }

    return;
}   /* __computeNetLoad() */
```

#### 4.3.3.6 __constructCombustionMap()

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

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

**Parameters**

| | |
|---|---|
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |

```
{
    //  1. get state table dimensions
    int n_cols = combustion_ptr_vec_ptr->size();
    int n_rows = pow(2, n_cols);

    //  2. init state table (all possible on/off combinations)
    std::vector<std::vector<bool» state_table;
    state_table.resize(n_rows, {});

    int x = 0;
    for (int i = 0; i < n_rows; i++) {
        state_table[i].resize(n_cols, false);

        x = i;
        for (int j = 0; j < n_cols; j++) {
            if (x % 2 == 0) {
                state_table[i][j] = true;
            }
            x /= 2;
        }
    }

    //  3. construct combustion map (handle duplicates by keeping rows with minimum
    //     trues)
    double total_capacity_kW = 0;
    int truth_count = 0;
    int current_truth_count = 0;

    for (int i = 0; i < n_rows; i++) {
        total_capacity_kW = 0;
        truth_count = 0;
        current_truth_count = 0;

        for (int j = 0; j < n_cols; j++) {
            if (state_table[i][j]) {
                total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
                truth_count++;
            }
        }

        if (this->combustion_map.count(total_capacity_kW) > 0) {
            for (int j = 0; j < n_cols; j++) {
                if (this->combustion_map[total_capacity_kW][j]) {
                    current_truth_count++;
                }
            }

```

```
            if (truth_count < current_truth_count) {
                this->combustion_map.erase(total_capacity_kW);
            }
        }

        this->combustion_map.insert(
            std::pair<double, std::vector<bool>> (
                total_capacity_kW,
                state_table[i]
            )
        );
    }

    //  4. test print
    /*
    std::cout << std::endl;

    std::cout << "\t\t";
    for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
        std::cout << combustion_ptr_vec_ptr->at(i)->capacity_kW << "\t";
    }
    std::cout << std::endl;

    std::map<double, std::vector<bool>>::iterator iter;
    for (
        iter = this->combustion_map.begin();
        iter != this->combustion_map.end();
        iter++
    ) {
        std::cout << iter->first << ":\t{\t";

        for (size_t i = 0; i < iter->second.size(); i++) {
            std::cout << iter->second[i] << "\t";
        }
        std::cout << "}" << std::endl;
    }
    */

    return;
}   /* __constructCombustionTable() */
```

#### 4.3.3.7 __getRenewableProduction()

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

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

**Parameters**

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

**Returns**

The production [kW] of the Renewable asset.

```
{
    double production_kW = 0;

    switch (renewable_ptr->type) {
        case (RenewableType :: SOLAR): {
            production_kW = renewable_ptr->computeProductionkW(
```

```
                timestep,
                dt_hrs,
                resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
            );

            break;
        }

        case (RenewableType :: TIDAL): {
            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
            );

            break;
        }

        case (RenewableType :: WAVE): {
            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0],
                resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1]
            );

            break;
        }

        case (RenewableType :: WIND): {
            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Controller::__getRenewableProduction():  ";
            error_str += "renewable type ";
            error_str += std::to_string(renewable_ptr->type);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* __getRenewableProduction() */
```

#### 4.3.3.8 __handleCombustionDispatch()

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

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

bool is_cycle_charging )

**Parameters**

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

**Returns**

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

```
{
    //  1. get minimal Combustion dispatch
    double target_production_kW = 1.2 * net_load_kW;
    double total_capacity_kW = 0;

    std::map<double, std::vector<bool»::iterator iter = this->combustion_map.begin();
    while (iter != std::prev(this->combustion_map.end(), 1)) {
        if (target_production_kW <= total_capacity_kW) {
            break;
        }

        iter++;
        total_capacity_kW = iter->first;
    }

    //  2. share load proportionally (by rated capacity) over active diesels
    Combustion* combustion_ptr;
    double production_kW = 0;
    double request_kW = 0;
    double _net_load_kW = net_load_kW;

    for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
        combustion_ptr = combustion_ptr_vec_ptr->at(i);

        if (total_capacity_kW > 0) {
            request_kW =
                int(this->combustion_map[total_capacity_kW][i]) *
                net_load_kW *
                (combustion_ptr->capacity_kW / total_capacity_kW);
        }

        else {
            request_kW = 0;
        }

        if (is_cycle_charging and request_kW > 0) {
            if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
                request_kW = 0.85 * combustion_ptr->capacity_kW;
            }
        }

        production_kW = combustion_ptr->requestProductionkW(
            timestep,
            dt_hrs,
            request_kW
        );

        _net_load_kW = combustion_ptr->commit(
            timestep,
            dt_hrs,
            production_kW,
            _net_load_kW
        );
    }

    return _net_load_kW;
}   /* __handleCombustionDispatch() */
```

### 4.3.3.9 __handleStorageCharging() [1/2]

```
void Controller::__handleStorageCharging (
        int timestep,
```

```
double dt_hrs,
std::list< Storage * > storage_ptr_list,
std::vector< Combustion * > * combustion_ptr_vec_ptr,
std::vector< Renewable * > * renewable_ptr_vec_ptr )  [private]
```

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

**Parameters**

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

```
{
    std::list<Storage*>::iterator iter;
    for (
        iter = storage_ptr_list.begin();
        iter != storage_ptr_list.end();
        iter++
    ){
        //...
    }

    return;
}   /* __handleStorageCharging() */
```

#### 4.3.3.10 __handleStorageCharging() [2/2]

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

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

**Parameters**

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

```
{
    for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
        //...
    }

    return;
}   /* __handleStorageCharging() */
```

#### 4.3.3.11 __handleStorageDischarging()

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

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

**Parameters**

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

**Returns**

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

```
{
    std::list<Storage*>::iterator iter;
    for (
        iter = storage_ptr_list.begin();
        iter != storage_ptr_list.end();
        iter++
    ){
        //...
    }

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

#### 4.3.3.12 applyDispatchControl()

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

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

**Parameters**

| | |
|---|---|
| *electrical_load_ptr* | A pointer to the ElectricalLoad component of the Model. |
| *combustion_ptr_vec_ptr* | A pointer to the Combustion pointer vector of the Model. |
| *renewable_ptr_vec_ptr* | A pointer to the Renewable pointer vector of the Model. |
| *storage_ptr_vec_ptr* | A pointer to the Storage pointer vector of the Model. |

```
{
    for (int i = 0; i < electrical_load_ptr->n_points; i++) {
        switch (this->control_mode) {
            case (ControlMode :: LOAD_FOLLOWING): {
                if (this->net_load_vec_kW[i] <= 0) {
                    this->__applyLoadFollowingControl_CHARGING(
                        i,
                        electrical_load_ptr,
```

```
                        combustion_ptr_vec_ptr,
                        renewable_ptr_vec_ptr,
                        storage_ptr_vec_ptr
                    );
                }

                else {
                    this->__applyLoadFollowingControl_DISCHARGING(
                        i,
                        electrical_load_ptr,
                        combustion_ptr_vec_ptr,
                        renewable_ptr_vec_ptr,
                        storage_ptr_vec_ptr
                    );
                }

                break;
            }

            case (ControlMode :: CYCLE_CHARGING): {
                if (this->net_load_vec_kW[i] <= 0) {
                    this->__applyCycleChargingControl_CHARGING(
                        i,
                        electrical_load_ptr,
                        combustion_ptr_vec_ptr,
                        renewable_ptr_vec_ptr,
                        storage_ptr_vec_ptr
                    );
                }

                else {
                    this->__applyCycleChargingControl_DISCHARGING(
                        i,
                        electrical_load_ptr,
                        combustion_ptr_vec_ptr,
                        renewable_ptr_vec_ptr,
                        storage_ptr_vec_ptr
                    );
                }

                break;
            }

            default: {
                std::string error_str = "ERROR:  Controller :: applyDispatchControl():  ";
                error_str += "control mode ";
                error_str += std::to_string(this->control_mode);
                error_str += " not recognized";

                #ifdef _WIN32
                    std::cout « error_str « std::endl;
                #endif

                throw std::runtime_error(error_str);

                break;
            }
        }
    }

    return;
}   /* applyDispatchControl() */
```

#### 4.3.3.13 clear()

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

Method to clear all attributes of the Controller object.

```
{
    this->net_load_vec_kW.clear();
    this->missed_load_vec_kW.clear();
    this->combustion_map.clear();

    return;
}   /* clear() */
```

#### 4.3.3.14 init()

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

Method to initialize the Controller component of the Model.

**Parameters**

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

```
{
    //  1. compute net load
    this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);

    //  2. construct Combustion table
    this->__constructCombustionMap(combustion_ptr_vec_ptr);

    return;
}   /* init() */
```

#### 4.3.3.15 setControlMode()

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

**Parameters**

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

```
{
    this->control_mode = control_mode;

    switch(control_mode) {
        case (ControlMode :: LOAD_FOLLOWING): {
            this->control_string = "LOAD_FOLLOWING";

            break;
        }

        case (ControlMode :: CYCLE_CHARGING): {
            this->control_string = "CYCLE_CHARGING";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Controller :: setControlMode():  ";
                error_str += "control mode ";
                error_str += std::to_string(control_mode);
                error_str += " not recognized";

                #ifdef _WIN32
                    std::cout << error_str << std::endl;
                #endif

                throw std::runtime_error(error_str);

            break;
```

```
        }
    }

    return;
}   /* setControlMode() */
```

## 4.3.4 Member Data Documentation

### 4.3.4.1 combustion_map

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

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

### 4.3.4.2 control_mode

`ControlMode Controller::control_mode`

The ControlMode that is active in the Model.

### 4.3.4.3 control_string

`std::string Controller::control_string`

A string describing the active ControlMode.

### 4.3.4.4 missed_load_vec_kW

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

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

### 4.3.4.5 net_load_vec_kW

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

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

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

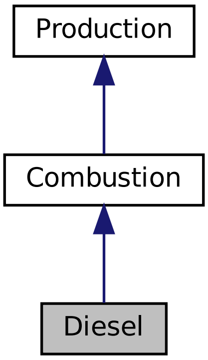
- header/Controller.h
- source/Controller.cpp

## 4.4 Diesel Class Reference

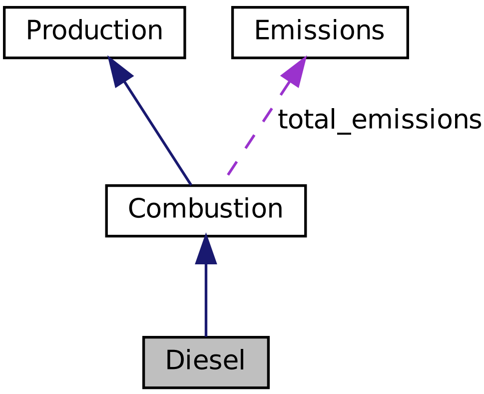
A derived class of the Combustion branch of Production which models production using a diesel generator.

```
#include <Diesel.h>
```

Inheritance diagram for Diesel:




Collaboration diagram for Diesel:




### Public Member Functions

- Diesel (void)

    *Constructor (dummy) for the Diesel class.*

- Diesel (int, double, DieselInputs)

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

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

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

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

    *Destructor for the Diesel class.*

## Public Attributes

- double minimum_load_ratio

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

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

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

## Private Member Functions

- void __checkInputs (DieselInputs)

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

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

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

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

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

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

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

    *Helper method to write time series results for Diesel.*

### 4.4.1 Detailed Description

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

## 4.4.2 Constructor & Destructor Documentation

### 4.4.2.1 Diesel() [1/2]

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

Constructor (dummy) for the Diesel class.

```
{
    return;
}   /* Diesel() */
```

### 4.4.2.2 Diesel() [2/2]

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

Constructor (intended) for the Diesel class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *diesel_inputs* | A structure of Diesel constructor inputs. |

```
  :
Combustion(
    n_points,
    n_years,
    diesel_inputs.combustion_inputs
)
{
    //  1. check inputs
    this->__checkInputs(diesel_inputs);

    //  2. set attributes
    this->type = CombustionType :: DIESEL;
    this->type_str = "DIESEL";

    this->replace_running_hrs = diesel_inputs.replace_running_hrs;

    this->fuel_cost_L = diesel_inputs.fuel_cost_L;

    this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
    this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
    this->time_since_last_start_hrs = 0;

    this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
    this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
    this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
    this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
    this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
    this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;

    if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
        this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
    }

    if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
```

```
        this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
    }

    if (diesel_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Diesel object constructed at " « this « std::endl;
    }

    return;
}   /* Diesel() */
```

#### 4.4.2.3 ∼Diesel()

Diesel::∼Diesel (
void )

Destructor for the Diesel class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Diesel object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Diesel() */
```

### 4.4.3 Member Function Documentation

#### 4.4.3.1 __checkInputs()

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

Helper method to check inputs to the Diesel constructor.

**Parameters**

| | |
|---|---|
| *diesel_inputs* | A structure of Diesel constructor inputs. |

```
{
    //  1. check fuel_cost_L
    if (diesel_inputs.fuel_cost_L < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::fuel_cost_L must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

```

```
        throw std::invalid_argument(error_str);
    }

    //  2. check CO2_emissions_intensity_kgL
    if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check CO_emissions_intensity_kgL
        if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check NOx_emissions_intensity_kgL
    if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  5. check SOx_emissions_intensity_kgL
    if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  6. check CH4_emissions_intensity_kgL
    if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  7. check PM_emissions_intensity_kgL
    if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  8. check minimum_load_ratio
    if (diesel_inputs.minimum_load_ratio < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::minimum_load_ratio must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

```

```
    //  9. check minimum_runtime_hrs
    if (diesel_inputs.minimum_runtime_hrs < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  10. check replace_running_hrs
    if (diesel_inputs.replace_running_hrs <= 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::replace_running_hrs must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

#### 4.4.3.2 __getGenericCapitalCost()

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

Helper method to generate a generic diesel generator capital cost.

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

**Returns**

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

```
{
    double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.425) + 800;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

#### 4.4.3.3 __getGenericFuelIntercept()

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

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

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

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

**Returns**

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

```
{
    double linear_fuel_intercept_LkWh = 0.0940 * pow(this->capacity_kW, -0.2735);

    return linear_fuel_intercept_LkWh;
}   /* __getGenericFuelIntercept() */
```

#### 4.4.3.4 __getGenericFuelSlope()

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

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

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

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

**Returns**

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

```
{
    double linear_fuel_slope_LkWh = 0.4234 * pow(this->capacity_kW, -0.1012);

    return linear_fuel_slope_LkWh;
}   /* __getGenericFuelSlope() */
```

#### 4.4.3.5 __getGenericOpMaintCost()

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

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

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

**Returns**

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

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

#### 4.4.3.6 __handleStartStop()

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

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

**Parameters**

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

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

    if (this->is_running) {
        // handle stopping
        if (
            production_kW <= 0 and
            this->time_since_last_start_hrs >= this->minimum_runtime_hrs
        ) {
            this->is_running = false;
        }
    }

    else {
        // handle starting
        if (production_kW > 0) {
            this->is_running = true;
            this->n_starts++;
            this->time_since_last_start_hrs = 0;
        }
    }

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

### 4.4.3.7 __writeSummary()

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

Helper method to write summary results for Diesel.

**Parameters**

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

Reimplemented from Combustion.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write to summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW DIESEL Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « "kW  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
```

```
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Combustion attributes
    ofs « "## Combustion Attributes\n";
    ofs « "\n";

    ofs « "Fuel Cost: " « this->fuel_cost_L « " per L  \n";
    ofs « "Nominal Fuel Escalation Rate (annual): "
        « this->nominal_fuel_escalation_annual « "  \n";
    ofs « "Real Fuel Escalation Rate (annual): "
        « this->real_fuel_escalation_annual « "  \n";
    ofs « "\n";

    ofs « "Linear Fuel Slope: " « this->linear_fuel_slope_LkWh « " L/kWh  \n";
    ofs « "Linear Fuel Intercept Coefficient: " « this->linear_fuel_intercept_LkWh
        « " L/kWh  \n";
    ofs « "\n";

    ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
        « this->CO2_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Carbon Monoxide (CO) Emissions Intensity: "
        « this->CO_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
        « this->NOx_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
        « this->SOx_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Methane (CH4) Emissions Intensity: "
        « this->CH4_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Particulate Matter (PM) Emissions Intensity: "
        « this->PM_emissions_intensity_kgL « " kg/L  \n";

    ofs « "\n--------\n\n";

    //  2.3. Diesel attributes
    ofs « "## Diesel Attributes\n";
    ofs « "\n";

    ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « "  \n";
    ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs  \n";

    ofs « "\n--------\n\n";

    //  2.4. Diesel Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Starts: " « this->n_starts « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
        « "(Annual Average: " « this->total_fuel_consumed_L / this->n_years
        « " L/yr)  \n";
    ofs « "\n";

    ofs « "Total Carbon Dioxide (CO2) Emissions: " «
        this->total_emissions.CO2_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.CO2_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Carbon Monoxide (CO) Emissions: " «
        this->total_emissions.CO_kg « " kg "
```

```
        « "(Annual Average: " «  this->total_emissions.CO_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
        this->total_emissions.NOx_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.NOx_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Sulfur Oxides (SOx) Emissions: " «
        this->total_emissions.SOx_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.SOx_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.CH4_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Particulate Matter (PM) Emissions: " «
        this->total_emissions.PM_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.PM_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __writeSummary() */
```

### 4.4.3.8 __writeTimeSeries()

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

Helper method to write time series results for Diesel.

**Parameters**

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

Reimplemented from Combustion.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Is Running (N = 0 / Y = 1),";
    ofs « "Fuel Consumption [L],";
    ofs « "Fuel Cost (actual),";
    ofs « "Carbon Dioxide (CO2) Emissions [kg],";
    ofs « "Carbon Monoxide (CO) Emissions [kg],";
    ofs « "Nitrogen Oxides (NOx) Emissions [kg],";
    ofs « "Sulfur Oxides (SOx) Emissions [kg],";
    ofs « "Methane (CH4) Emissions [kg],";
    ofs « "Particulate Matter (PM) Emissions [kg],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

```

```
    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->is_running_vec[i] « ",";
        ofs « this->fuel_consumption_vec_L[i] « ",";
        ofs « this->fuel_cost_vec[i] « ",";
        ofs « this->CO2_emissions_vec_kg[i] « ",";
        ofs « this->CO_emissions_vec_kg[i] « ",";
        ofs « this->NOx_emissions_vec_kg[i] « ",";
        ofs « this->SOx_emissions_vec_kg[i] « ",";
        ofs « this->CH4_emissions_vec_kg[i] « ",";
        ofs « this->PM_emissions_vec_kg[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

#### 4.4.3.9 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Combustion.

```
{
    //  1. handle start/stop, enforce minimum runtime constraint
    this->__handleStartStop(timestep, dt_hrs, production_kW);

    //  2. invoke base class method
    load_kW = Combustion :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );

    if (this->is_running) {
        //  3. log time since last start
        this->time_since_last_start_hrs += dt_hrs;

        //  4. correct operation and maintenance costs (should be non-zero if idling)
        if (production_kW <= 0) {
```

```
            double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;

            double operation_maintenance_cost =
                this->operation_maintenance_cost_kWh * produced_kWh;
            this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
        }
    }

    return load_kW;
}   /* commit() */
```

#### 4.4.3.10 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Combustion.

```
{
    //  1. reset attributes
    this->time_since_last_start_hrs = 0;

    //  2. invoke base class method
    Combustion :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

#### 4.4.3.11 requestProductionkW()

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

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

**Parameters**

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

**Returns**

The production [kW] delivered by the diesel generator.

Reimplemented from Combustion.

```
{
    //  1. return on request of zero
    if (request_kW <= 0) {
        return 0;
    }

    double deliver_kW = request_kW;

    //  2. enforce capacity constraint
    if (deliver_kW > this->capacity_kW) {
        deliver_kW = this->capacity_kW;
    }

    //  3. enforce minimum load ratio
    if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
        deliver_kW = this->minimum_load_ratio * this->capacity_kW;
    }

    return deliver_kW;
}   /* requestProductionkW() */
```

## 4.4.4 Member Data Documentation

### 4.4.4.1 minimum_load_ratio

`double Diesel::minimum_load_ratio`

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

### 4.4.4.2 minimum_runtime_hrs

`double Diesel::minimum_runtime_hrs`

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

### 4.4.4.3 time_since_last_start_hrs

`double Diesel::time_since_last_start_hrs`

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

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

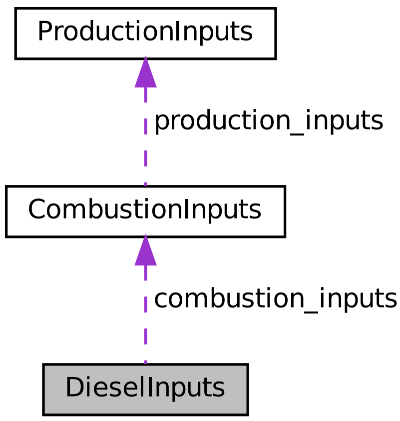
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

## 4.5 DieselInputs Struct Reference

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

```
#include <Diesel.h>
```

Collaboration diagram for DieselInputs:




### Public Attributes

- CombustionInputs combustion_inputs

  *An encapsulated CombustionInputs instance.*
- double replace_running_hrs = 30000

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

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

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

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

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

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

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

- double linear_fuel_intercept_LkWh = -1

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

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

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

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

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

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

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

## 4.5.1 Detailed Description

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

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

## 4.5.2 Member Data Documentation

### 4.5.2.1 capital_cost

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

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

### 4.5.2.2 CH4_emissions_intensity_kgL

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

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

#### 4.5.2.3 CO2_emissions_intensity_kgL

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

Carbon dioxide ($CO_2$) 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 &lt;ElectricalLoad.h&gt;
```

### Public Member Functions

- ElectricalLoad (void)

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

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

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

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

  *Destructor for the ElectricalLoad class.*

### Public Attributes

- int n_points

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

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

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

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

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

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

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

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

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

## 4.6.1 Detailed Description

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

## 4.6.2 Constructor & Destructor Documentation

### 4.6.2.1 ElectricalLoad() [1/2]

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

Constructor (dummy) for the ElectricalLoad class.

```
{
    return;
}   /* ElectricalLoad() */
```

### 4.6.2.2 ElectricalLoad() [2/2]

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

Constructor (intended) for the ElectricalLoad class.

**Parameters**

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

```
{
    this->readLoadData(path_2_electrical_load_time_series);

    return;
}   /* ElectricalLoad() */
```

### 4.6.2.3 ∼ElectricalLoad()

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

Destructor for the ElectricalLoad class.

```
{
    this->clear();
    return;
}   /* ~ElectricalLoad() */
```

## 4.6.3 Member Function Documentation

#### 4.6.3.1 clear()

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

Method to clear all attributes of the ElectricalLoad object.

```
{
    this->n_points = 0;
    this->n_years = 0;
    this->min_load_kW = 0;
    this->mean_load_kW = 0;
    this->max_load_kW = 0;

    this->path_2_electrical_load_time_series.clear();
    this->time_vec_hrs.clear();
    this->dt_vec_hrs.clear();
    this->load_vec_kW.clear();

    return;
}   /* clear() */
```

#### 4.6.3.2 readLoadData()

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

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

**Parameters**

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

```
{
    //  1. clear
    this->clear();

    //  2. init CSV reader, record path
    io::CSVReader<2> CSV(path_2_electrical_load_time_series);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Electrical Load [kW]"
    );

    this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;

    //  3. read in time and load data, increment n_points, track min and max load
    double time_hrs = 0;
    double load_kW = 0;
    double load_sum_kW = 0;

    this->n_points = 0;

    this->min_load_kW = std::numeric_limits<double>::infinity();
    this->max_load_kW = -1 * std::numeric_limits<double>::infinity();

    while (CSV.read_row(time_hrs, load_kW)) {
        this->time_vec_hrs.push_back(time_hrs);
        this->load_vec_kW.push_back(load_kW);

        load_sum_kW += load_kW;

        this->n_points++;

        if (this->min_load_kW > load_kW) {
            this->min_load_kW = load_kW;
        }

```

```
        if (this->max_load_kW < load_kW) {
            this->max_load_kW = load_kW;
        }
    }

    //  4. compute mean load
    this->mean_load_kW = load_sum_kW / this->n_points;

    //  5. set number of years (assuming 8,760 hours per year)
    this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;

    //  6. populate dt_vec_hrs
    this->dt_vec_hrs.resize(n_points, 0);

    for (int i = 0; i < n_points; i++) {
        if (i == n_points - 1) {
            this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
        }

        else {
            double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];

            this->dt_vec_hrs[i] = dt_hrs;
        }
    }

    return;
}   /* readLoadData() */
```

## 4.6.4 Member Data Documentation

### 4.6.4.1 dt_vec_hrs

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

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

### 4.6.4.2 load_vec_kW

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

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

### 4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

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

#### 4.6.4.4 mean_load_kW

`double ElectricalLoad::mean_load_kW`

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

#### 4.6.4.5 min_load_kW

`double ElectricalLoad::min_load_kW`

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

#### 4.6.4.6 n_points

`int ElectricalLoad::n_points`

The number of points in the modelling time series.

#### 4.6.4.7 n_years

`double ElectricalLoad::n_years`

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

#### 4.6.4.8 path_2_electrical_load_time_series

`std::string ElectricalLoad::path_2_electrical_load_time_series`

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

#### 4.6.4.9 time_vec_hrs

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

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

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

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

# 4.7 Emissions Struct Reference

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

```
#include <Combustion.h>
```

## Public Attributes

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

### 4.7.1 Detailed Description

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

### 4.7.2 Member Data Documentation

#### 4.7.2.1 CH4_kg

```
double Emissions::CH4_kg = 0
```

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

#### 4.7.2.2 CO2_kg

```
double Emissions::CO2_kg = 0
```

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

#### 4.7.2.3 CO_kg

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

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

#### 4.7.2.4 NOx_kg

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

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

#### 4.7.2.5 PM_kg

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

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

#### 4.7.2.6 SOx_kg

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

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

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

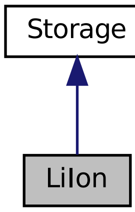
- header/Production/Combustion/Combustion.h

## 4.8 LiIon Class Reference

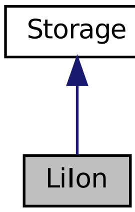
A derived class of Storage which models energy storage by way of lithium-ion batteries.

`#include <LiIon.h>`

Inheritance diagram for LiIon:




Collaboration diagram for LiIon:




### Public Member Functions

- LiIon (void)

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

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

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

  *Method to get the discharge power currently available from the asset.*
- double getAcceptablekW (double)

  *Method to get the charge power currently acceptable by the asset.*
- void commitCharge (int, double, double)

  *Method which takes in the charging power for the current timestep and records.*
- double commitDischarge (int, double, double, double)
- ∼LiIon (void)

  *Destructor for the LiIon class.*

## Public Attributes

- double dynamic_energy_capacity_kWh

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

  *The state of health of the asset.*
- double replace_SOH

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

  *The initial state of charge of the asset.*
- double min_SOC

  *The minimum state of charge of the asset. Will toggle is_depleted when reached.*
- double hysteresis_SOC

  *The state of charge the asset must achieve to toggle is_depleted.*
- double max_SOC

  *The maximum state of charge of the asset.*
- double charging_efficiency

  *The charging efficiency of the asset.*
- double discharging_efficiency

  *The discharging efficiency of the asset.*
- std::vector< double > SOH_vec

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

## Private Member Functions

- void __checkInputs (LiIonInputs)

  *Helper method to check inputs to the LiIon constructor.*
- double __getGenericCapitalCost (void)

  *Helper method to generate a generic lithium ion battery energy storage system capital cost.*
- double __getGenericOpMaintCost (void)

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

  *Helper method to toggle the is_depleted attribute of LiIon.*
- void __writeSummary (std::string)

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

  *Helper method to write time series results for LiIon.*

### 4.8.1 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() [1/2]

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

Constructor (dummy) for the LiIon class.

```
{
    return;
}   /* LiIon() */
```

#### 4.8.2.2 LiIon() [2/2]

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

Constructor (intended) for the LiIon class.

**Parameters**

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

```
  :
Storage(
    n_points,
    n_years,
    liion_inputs.storage_inputs
)
{
    //  1. check inputs
    this->__checkInputs(liion_inputs);

    //  2. set attributes
    this->type = StorageType :: LIION;
    this->type_str = "LiIon";

    this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
    this->SOH = 1;
    this->replace_SOH = liion_inputs.replace_SOH;

    this->init_SOC = liion_inputs.init_SOC;
    this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;

    this->min_SOC = liion_inputs.min_SOC;
    this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
    this->max_SOC = liion_inputs.max_SOC;

    this->charging_efficiency = liion_inputs.charging_efficiency;
    this->discharging_efficiency = liion_inputs.discharging_efficiency;

    if (liion_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (liion_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    this->SOH_vec.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
```

```
        std::cout « "LiIon object constructed at " « this « std::endl;
    }

    return;
}   /* LiIon() */
```

#### 4.8.2.3 ∼LiIon()

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

Destructor for the LiIon class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "LiIon object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~LiIon() */
```

### 4.8.3 Member Function Documentation

#### 4.8.3.1 __checkInputs()

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

Helper method to check inputs to the LiIon constructor.

**Parameters**

| | |
|---|---|
| *liion_inputs* | A structure of LiIon constructor inputs. |

```
{
    //  1. check replace_SOH
    if (liion_inputs.replace_SOH < 0 or liion_inputs.replace_SOH > 1) {
        std::string error_str = "ERROR:  LiIon():  replace_SOH must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check init_SOC
    if (liion_inputs.init_SOC < 0 or liion_inputs.init_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  init_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check min_SOC
    if (liion_inputs.min_SOC < 0 or liion_inputs.min_SOC > 1) {
```

```
        std::string error_str = "ERROR:  LiIon():  min_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check hysteresis_SOC
    if (liion_inputs.hysteresis_SOC < 0 or liion_inputs.hysteresis_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  hysteresis_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  5. check max_SOC
    if (liion_inputs.max_SOC < 0 or liion_inputs.max_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  max_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  6. check charging_efficiency
    if (liion_inputs.charging_efficiency <= 0 or liion_inputs.charging_efficiency > 1) {
        std::string error_str = "ERROR:  LiIon():  charging_efficiency must be in the ";
        error_str += "half-open interval (0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  7. check discharging_efficiency
    if (
        liion_inputs.discharging_efficiency <= 0 or
        liion_inputs.discharging_efficiency > 1
    ) {
        std::string error_str = "ERROR:  LiIon():  discharging_efficiency must be in the ";
        error_str += "half-open interval (0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

#### 4.8.3.2 __getGenericCapitalCost()

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

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

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

**Returns**

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

```
{
    double capital_cost_per_kWh = 250 * pow(this->energy_capacity_kWh, -0.15) + 650;

    return capital_cost_per_kWh * this->energy_capacity_kWh;
}   /* __getGenericCapitalCost() */
```

#### 4.8.3.3 __getGenericOpMaintCost()

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

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

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

**Returns**

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

```
{
    return 0.01;
}   /* __getGenericOpMaintCost() */
```

#### 4.8.3.4 __toggleDepleted()

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

Helper method to toggle the is_depleted attribute of LiIon.

```
{
    if (this->is_depleted) {
        double hysteresis_charge_kWh = this->hysteresis_SOC * this->energy_capacity_kWh;

        if (hysteresis_charge_kWh > this->dynamic_energy_capacity_kWh) {
            hysteresis_charge_kWh = this->dynamic_energy_capacity_kWh;
        }

        if (this->charge_kWh >= hysteresis_charge_kWh) {
            this->is_depleted = false;
        }
    }

    else {
        double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;

        if (this->charge_kWh <= min_charge_kWh) {
            this->is_depleted = true;
        }
    }

    return;
}   /* __toggleDepleted() */
```

#### 4.8.3.5 __writeSummary()

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

Helper method to write summary results for LiIon.

**Parameters**

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

Reimplemented from Storage.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->power_capacity_kW)));
    ofs « " kW ";
    ofs « std::to_string(int(ceil(this->energy_capacity_kWh)));
    ofs « " kWh LIION Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Storage attributes
    ofs « "## Storage Attributes\n";
    ofs « "\n";
    ofs « "Power Capacity: " « this->power_capacity_kW « "kW  \n";
    ofs « "Energy Capacity: " « this->energy_capacity_kWh « "kWh  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh charged/discharged  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";

    ofs « "\n--------\n\n";

    //  2.2. LiIon attributes
    ofs « "## LiIon Attributes\n";
    ofs « "\n";

    ofs « "Charging Efficiency: " « this->charging_efficiency « "  \n";
    ofs « "Discharging Efficiency: " « this->discharging_efficiency « "  \n";
    ofs « "\n";

    ofs « "Initial State of Charge: " « this->init_SOC « "  \n";
    ofs « "Minimum State of Charge: " « this->min_SOC « "  \n";
    ofs « "Hyteresis State of Charge: " « this->hysteresis_SOC « "  \n";
    ofs « "Maximum State of Charge: " « this->max_SOC « "  \n";
    ofs « "\n";

    ofs « "Replacement State of Health: " « this->replace_SOH « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. LiIon Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Discharge: " « this->total_discharge_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";
    ofs.close();
    return;
}   /* __writeSummary() */
```

#### 4.8.3.6 __writeTimeSeries()

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

Helper method to write time series results for LiIon.

**Parameters**

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

Reimplemented from Storage.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Charging Power [kW],";
    ofs « "Discharging Power [kW],";
    ofs « "Charge (at end of timestep) [kWh],";
    ofs « "State of Health (at end of timestep) [ ],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « this->charging_power_vec_kW[i] « ",";
        ofs « this->discharging_power_vec_kW[i] « ",";
        ofs « this->charge_vec_kWh[i] « ",";
        ofs « this->SOH_vec[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

#### 4.8.3.7 commitCharge()

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

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

**Parameters**

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

Reimplemented from Storage.

```
{
    // 1. record charging power
    this->charging_power_vec_kW[timestep] = charging_kW;

    // 2. update charge and record
    this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
    this->charge_vec_kWh[timestep] = this->charge_kWh;

    // 3. toggle depleted flag (if applicable)
    this->__toggleDepleted();

    // 4. model degradation
    //...

    // 5. trigger replacement (if applicable)
    if (this->SOH <= this->replace_SOH) {
        this->handleReplacement(timestep);
    }

    this->power_kW= 0;
    return;
}   /* commitCharge() */
```

#### 4.8.3.8 commitDischarge()

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

Reimplemented from Storage.

```
{
    // 1. record discharging power, update total
    this->discharging_power_vec_kW[timestep] = discharging_kW;
    this->total_discharge_kWh += discharging_kW * dt_hrs;

    // 2. update charge and record
    this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
    this->charge_vec_kWh[timestep] = this->charge_kWh;

    // 3. update load
    load_kW -= discharging_kW;

    // 4. toggle depleted flag (if applicable)
    this->__toggleDepleted();

    // 5. model degradation
    //...

    // 6. trigger replacement (if applicable)
    if (this->SOH <= this->replace_SOH) {
        this->handleReplacement(timestep);
    }

    this->power_kW = 0;
    return load_kW;
}   /* commitDischarge() */
```

#### 4.8.3.9 getAcceptablekW()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Storage.

```
{
    //  1. get max charge
    double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;

    if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
        max_charge_kWh = this->dynamic_energy_capacity_kWh;
    }

    //  2. compute acceptable power
    //     (accounting for the power currently being charged/discharged by the asset)
    double acceptable_kW =
        (max_charge_kWh - this->charge_kWh) /
        (this->charging_efficiency * dt_hrs);

    acceptable_kW -= this->power_kW;

    if (acceptable_kW <= 0) {
        return 0;
    }

    //  3. apply power constraint
    if (acceptable_kW > this->power_capacity_kW) {
        acceptable_kW = this->power_capacity_kW;
    }

    return acceptable_kW;
}   /* getAcceptablekW( */
```

### 4.8.3.10 getAvailablekW()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Storage.

```
{
    //  1. get min charge
    double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;

    //  2. compute available power
    //     (accounting for the power currently being charged/discharged by the asset)
    double available_kW =
        ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
        dt_hrs;
```

```
    available_kW -= this->power_kW;

    if (available_kW <= 0) {
        return 0;
    }

    //  3. apply power constraint
    if (available_kW > this->power_capacity_kW) {
        available_kW = this->power_capacity_kW;
    }

    return available_kW;
}   /* getAvailablekW() */
```

#### 4.8.3.11 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Storage.

```
{
    //  1. reset attributes
    this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
    this->SOH = 1;

    // 2. invoke base class method
    Storage::handleReplacement(timestep);

    //  3. correct attributes
    this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
    this->is_depleted = false;

    return;
}   /* __handleReplacement() */
```

### 4.8.4 Member Data Documentation

#### 4.8.4.1 charging_efficiency

```
double LiIon::charging_efficiency
```

The charging efficiency of the asset.

#### 4.8.4.2 discharging_efficiency

```
double LiIon::discharging_efficiency
```

The discharging efficiency of the asset.

#### 4.8.4.3 dynamic_energy_capacity_kWh

```
double LiIon::dynamic_energy_capacity_kWh
```

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

#### 4.8.4.4 hysteresis_SOC

```
double LiIon::hysteresis_SOC
```

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

#### 4.8.4.5 init_SOC

```
double LiIon::init_SOC
```

The initial state of charge of the asset.

#### 4.8.4.6 max_SOC

```
double LiIon::max_SOC
```

The maximum state of charge of the asset.

#### 4.8.4.7 min_SOC

```
double LiIon::min_SOC
```

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

#### 4.8.4.8 replace_SOH

```
double LiIon::replace_SOH
```

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

#### 4.8.4.9 SOH

```
double LiIon::SOH
```

The state of health of the asset.

#### 4.8.4.10 SOH_vec

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

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

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

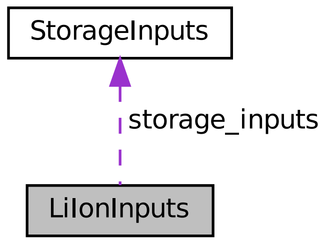
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

## 4.9 LiIonInputs Struct Reference

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

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

Collaboration diagram for LiIonInputs:



## Public Attributes

- StorageInputs storage_inputs

  *An encapsulated StorageInputs instance.*
- double capital_cost = -1

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

  *The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].*
- double init_SOC = 0.5

  *The initial state of charge of the asset.*
- double min_SOC = 0.15

  *The minimum state of charge of the asset. Will toggle is_depleted when reached.*
- double hysteresis_SOC = 0.5

  *The state of charge the asset must achieve to toggle is_depleted.*
- double max_SOC = 0.9

  *The maximum state of charge of the asset.*
- double charging_efficiency = 0.9

  *The charging efficiency of the asset.*
- double discharging_efficiency = 0.9

  *The discharging efficiency of the asset.*
- double replace_SOH = 0.8

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

### 4.9.1 Detailed Description

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

### 4.9.2 Member Data Documentation

#### 4.9.2.1 capital_cost

```
double LiIonInputs::capital_cost = -1
```

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

#### 4.9.2.2 charging_efficiency

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

The charging efficiency of the asset.

#### 4.9.2.3 discharging_efficiency

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

The discharging efficiency of the asset.

#### 4.9.2.4 hysteresis_SOC

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

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

#### 4.9.2.5 init_SOC

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

The initial state of charge of the asset.

#### 4.9.2.6 max_SOC

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

The maximum state of charge of the asset.

#### 4.9.2.7 min_SOC

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

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

#### 4.9.2.8 operation_maintenance_cost_kWh

```
double LiIonInputs::operation_maintenance_cost_kWh = -1
```

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

#### 4.9.2.9 replace_SOH

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

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

#### 4.9.2.10 storage_inputs

```
StorageInputs LiIonInputs::storage_inputs
```

An encapsulated StorageInputs instance.

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

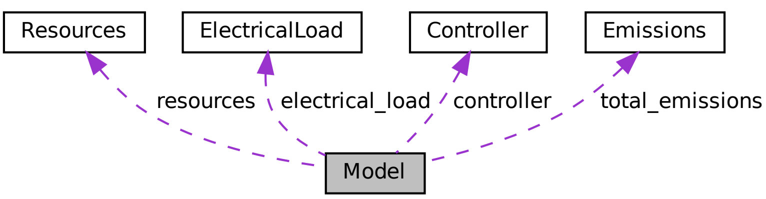
- header/Storage/LiIon.h

## 4.10 Model Class Reference

A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

```
#include <Model.h>
```

Collaboration diagram for Model:



## Public Member Functions

- Model (void)

  *Constructor (dummy) for the Model class.*
- Model (ModelInputs)

  *Constructor (intended) for the Model class.*
- void addDiesel (DieselInputs)

  *Method to add a Diesel asset to the Model.*
- void addResource (RenewableType, std::string, int)

  *A method to add a renewable resource time series to the Model.*
- void addSolar (SolarInputs)

  *Method to add a Solar asset to the Model.*
- void addTidal (TidalInputs)

  *Method to add a Tidal asset to the Model.*
- void addWave (WaveInputs)

  *Method to add a Wave asset to the Model.*
- void addWind (WindInputs)

  *Method to add a Wind asset to the Model.*
- void addLiIon (LiIonInputs)

  *Method to add a LiIon asset to the Model.*
- void run (void)

  *A method to run the Model.*
- void reset (void)

  *Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors and resets select Model attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.*
- void clear (void)

  *Method to clear all attributes of the Model object.*
- void writeResults (std::string, int=-1)

  *Method which writes Model results to an output directory. Also calls out to writeResults() for each contained asset.*
- ∼Model (void)

  *Destructor for the Model class.*

## Public Attributes

- double total_fuel_consumed_L

  *The total fuel consumed [L] over a model run.*
- Emissions total_emissions

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

  *The net present cost of the Model (undefined currency).*
- double total_dispatch_discharge_kWh

  *The total energy dispatched/discharged [kWh] over the Model run.*
- double levellized_cost_of_energy_kWh

  *The levellized cost of energy, per unit energy dispatched/discharged, of the Model [1/kWh] (undefined currency).*
- Controller controller

  *Controller component of Model.*
- ElectricalLoad electrical_load

  *ElectricalLoad component of Model.*
- Resources resources

  *Resources component of Model.*

- std::vector< Combustion * > combustion_ptr_vec

  *A vector of pointers to the various Combustion assets in the Model.*
- std::vector< Renewable * > renewable_ptr_vec

  *A vector of pointers to the various Renewable assets in the Model.*
- std::vector< Storage * > storage_ptr_vec

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

## Private Member Functions

- void __checkInputs (ModelInputs)

  *Helper method (private) to check inputs to the Model constructor.*
- void __computeFuelAndEmissions (void)

  *Helper method to compute the total fuel consumption and emissions over the Model run.*
- void __computeNetPresentCost (void)

  *Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs.*
- void __computeLevellizedCostOfEnergy (void)

  *Helper method to compute the overall levellized cost of energy, for the Model run, from the asset-wise levellized costs of energy.*
- void __computeEconomics (void)

  *Helper method to compute key economic metrics for the Model run.*
- void __writeSummary (std::string)

  *Helper method to write summary results for Model.*
- void __writeTimeSeries (std::string, int=-1)

  *Helper method to write time series results for Model.*

## 4.10.1 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.

## 4.10.2 Constructor & Destructor Documentation

### 4.10.2.1 Model() [1/2]

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

Constructor (dummy) for the Model class.

```
497 {
498     return;
499 }   /* Model() */
```

### 4.10.2.2 Model() [2/2]

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

Constructor (intended) for the Model class.

**Parameters**

| | |
|---|---|
| *model_inputs* | A structure of Model constructor inputs. |

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

    //  2. read in electrical load data
    this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);

    //  3. set control mode
    this->controller.setControlMode(model_inputs.control_mode);

    //  4. set public attributes
    this->total_fuel_consumed_L = 0;
    this->net_present_cost = 0;
    this->total_dispatch_discharge_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    return;
}   /* Model() */
```

#### 4.10.2.3 ∼Model()

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

Destructor for the Model class.

```
{
    this->clear();
    return;
}   /* ~Model() */
```

### 4.10.3 Member Function Documentation

#### 4.10.3.1 __checkInputs()

```
void Model::__checkInputs (
            ModelInputs ) [private]
```

Helper method (private) to check inputs to the Model constructor.

**Parameters**

| | |
|---|---|
| *model_inputs* | A structure of Model constructor inputs. |

```
{
    //...

    return;
}   /* __checkInputs() */
```

#### 4.10.3.2 __computeEconomics()

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

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

```
{
    this->__computeNetPresentCost();
    this->__computeLevellizedCostOfEnergy();

    return;
}   /* __computeEconomics() */
```

#### 4.10.3.3 __computeFuelAndEmissions()

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

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

```
{
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->computeFuelAndEmissions();

        this->total_fuel_consumed_L +=
            this->combustion_ptr_vec[i]->total_fuel_consumed_L;

        this->total_emissions.CO2_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CO2_kg;

        this->total_emissions.CO_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CO_kg;

        this->total_emissions.NOx_kg +=
            this->combustion_ptr_vec[i]->total_emissions.NOx_kg;

        this->total_emissions.SOx_kg +=
            this->combustion_ptr_vec[i]->total_emissions.SOx_kg;

        this->total_emissions.CH4_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CH4_kg;

        this->total_emissions.PM_kg +=
            this->combustion_ptr_vec[i]->total_emissions.PM_kg;
    }

    return;
}   /* __computeFuelAndEmissions() */
```

#### 4.10.3.4 __computeLevellizedCostOfEnergy()

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

Helper method to compute the overall levellized cost of energy, for the Model run, from the asset-wise levellized costs of energy.

```
{
    //  1. account for Combustion economics in levellized cost of energy
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->combustion_ptr_vec[i]->total_dispatch_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

```

```
    //  2. account for Renewable economics in levellized cost of energy
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->renewable_ptr_vec[i]->total_dispatch_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

    //  3. account for Storage economics in levellized cost of energy
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        /*
        this->levellized_cost_of_energy_kWh +=
            (
                this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->storage_ptr_vec[i]->total_discharge_kWh
            ) / this->total_dispatch_discharge_kWh;
        */
    }

    return;
}   /* __computeLevellizedCostOfEnergy() */
```

#### 4.10.3.5 __computeNetPresentCost()

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

Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs.

```
{
    //  1. account for Combustion economics in net present cost
    //     increment total dispatch
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->combustion_ptr_vec[i]->total_dispatch_kWh;
    }

    //  2. account for Renewable economics in net present cost,
    //     increment total dispatch
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->renewable_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->renewable_ptr_vec[i]->total_dispatch_kWh;
    }

    //  3. account for Storage economics in net present cost
    //     increment total dispatch
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        this->storage_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->storage_ptr_vec[i]->total_discharge_kWh;
    }

    return;
}   /* __computeNetPresentCost() */
```

### 4.10.3.6 __writeSummary()

```
void Model::__writeSummary (
            std::string write_path ) [private]
```

Helper method to write summary results for Model.

**Parameters**

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

```
{
    //  1. create subdirectory
    write_path += "Model/";
    std::filesystem::create_directory(write_path);

    //  2. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  3. write summary results (markdown)
    ofs « "# Model Summary Results\n";
    ofs « "\n--------\n\n";

    //  3.1. ElectricalLoad
    ofs « "## Electrical Load\n";
    ofs « "\n";
    ofs « "Path: " «
        this->electrical_load.path_2_electrical_load_time_series « "  \n";
    ofs « "Data Points: " « this->electrical_load.n_points « "  \n";
    ofs « "Years: " « this->electrical_load.n_years « "  \n";
    ofs « "Min: " « this->electrical_load.min_load_kW « " kW  \n";
    ofs « "Mean: " « this->electrical_load.mean_load_kW « " kW  \n";
    ofs « "Max: " « this->electrical_load.max_load_kW « " kW  \n";
    ofs « "\n--------\n\n";

    //  3.2. Controller
    ofs « "## Controller\n";
    ofs « "\n";
    ofs « "Control Mode: " « this->controller.control_string « "  \n";
    ofs « "\n--------\n\n";

    //  3.3. Resources (1D)
    ofs « "## 1D Renewable Resources\n";
    ofs « "\n";

    std::map<int, std::string>::iterator string_map_1D_iter =
        this->resources.string_map_1D.begin();
    std::map<int, std::string>::iterator path_map_1D_iter =
        this->resources.path_map_1D.begin();

    while (
        string_map_1D_iter != this->resources.string_map_1D.end() and
        path_map_1D_iter != this->resources.path_map_1D.end()
    ) {
        ofs « "Resource Key: " « string_map_1D_iter->first « "  \n";
        ofs « "Type: " « string_map_1D_iter->second « "  \n";
        ofs « "Path: " « path_map_1D_iter->second « "  \n";
        ofs « "\n";

        string_map_1D_iter++;
        path_map_1D_iter++;
    }

    ofs « "\n--------\n\n";

    //  3.4. Resources (2D)
    ofs « "## 2D Renewable Resources\n";
    ofs « "\n";

    std::map<int, std::string>::iterator string_map_2D_iter =
        this->resources.string_map_2D.begin();
    std::map<int, std::string>::iterator path_map_2D_iter =
        this->resources.path_map_2D.begin();

    while (
        string_map_2D_iter != this->resources.string_map_2D.end() and
        path_map_2D_iter != this->resources.path_map_2D.end()
```

```
    ) {
        ofs « "Resource Key: " « string_map_2D_iter->first « "  \n";
        ofs « "Type: " « string_map_2D_iter->second « "  \n";
        ofs « "Path: " « path_map_2D_iter->second « "  \n";
        ofs « "\n";

        string_map_2D_iter++;
        path_map_2D_iter++;
    }

    ofs « "\n--------\n\n";

    //  3.5. Combustion
    ofs « "## Combustion Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->combustion_ptr_vec[i]->type_str « "  \n";
        ofs « "Capacity: " « this->combustion_ptr_vec[i]->capacity_kW « " kW  \n";
        ofs « "\n";
    }

    ofs « "\n--------\n\n";

    //  3.6. Renewable
    ofs « "## Renewable Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->renewable_ptr_vec[i]->type_str « "  \n";
        ofs « "Capacity: " « this->renewable_ptr_vec[i]->capacity_kW « " kW  \n";
        ofs « "\n";
    }

    ofs « "\n--------\n\n";

    //  3.7. Storage
    ofs « "## Storage Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        //...
    }

    ofs « "\n--------\n\n";

    //  3.8. Model Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch + Discharge: " « this->total_dispatch_discharge_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched/discharged  \n";
    ofs « "\n";

    ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
        « "(Annual Average: " «
            this->total_fuel_consumed_L / this->electrical_load.n_years
        « " L/yr)  \n";
    ofs « "\n";

    ofs « "Total Carbon Dioxide (CO2) Emissions: " «
        this->total_emissions.CO2_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CO2_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Carbon Monoxide (CO) Emissions: " «
        this->total_emissions.CO_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CO_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
        this->total_emissions.NOx_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.NOx_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Sulfur Oxides (SOx) Emissions: " «
```

```
        this->total_emissions.SOx_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.SOx_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CH4_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Particulate Matter (PM) Emissions: " «
        this->total_emissions.PM_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.PM_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __writeSummary() */
```

### 4.10.3.7 __writeTimeSeries()

```
void Model::__writeTimeSeries (
                std::string write_path,
                int max_lines = -1 )  [private]
```

Helper method to write time series results for Model.

**Parameters**

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

```
{
    //  1. create filestream
    write_path += "Model/time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results header (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Electrical Load [kW],";
    ofs « "Net Load [kW],";
    ofs « "Missed Load [kW],";

    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        ofs « this->renewable_ptr_vec[i]->capacity_kW « " kW "
            « this->renewable_ptr_vec[i]->type_str « " Dispatch [kW],";
    }

    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        //...
    }

    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        ofs « this->combustion_ptr_vec[i]->capacity_kW « " kW "
            « this->combustion_ptr_vec[i]->type_str « " Dispatch [kW],";
    }

    ofs « "\n";

    //  3. write time series results values (comma separated value)
    for (int i = 0; i < max_lines; i++) {
        //  3.1. load values
        ofs « this->electrical_load.time_vec_hrs[i] « ",";
        ofs « this->electrical_load.load_vec_kW[i] « ",";
        ofs « this->controller.net_load_vec_kW[i] « ",";
        ofs « this->controller.missed_load_vec_kW[i] « ",";

        //  3.2. asset-wise dispatch/discharge
```

```
        for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
            ofs « this->renewable_ptr_vec[j]->dispatch_vec_kW[i] « ",";
        }

        for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
            //...
        }

        for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
            ofs « this->combustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
        }

        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

#### 4.10.3.8 addDiesel()

```
void Model::addDiesel (
            DieselInputs diesel_inputs )
```

Method to add a Diesel asset to the Model.

**Parameters**

| | |
|---|---|
| *diesel_inputs* | A structure of Diesel constructor inputs. |

```
{
    Combustion* diesel_ptr = new Diesel(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        diesel_inputs
    );

    this->combustion_ptr_vec.push_back(diesel_ptr);

    return;
}   /* addDiesel() */
```

#### 4.10.3.9 addLiIon()

```
void Model::addLiIon (
            LiIonInputs liion_inputs )
```

Method to add a LiIon asset to the Model.

**Parameters**

| | |
|---|---|
| *liion_inputs* | A structure of LiIon constructor inputs. |

```
{
    Storage* liion_ptr = new LiIon(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        liion_inputs
    );

    this->storage_ptr_vec.push_back(liion_ptr);

```

```
    return;
}   /* addLiIon() */
```

#### 4.10.3.10 addResource()

```
void Model::addResource (
            RenewableType renewable_type,
            std::string path_2_resource_data,
            int resource_key )
```

A method to add a renewable resource time series to the Model.

**Parameters**

| | |
|---|---|
| *renewable_type* | The type of renewable resource being added to the Model. |
| *path_2_resource_data* | A string defining the path (either relative or absolute) to the given resource time series. |
| *resource_key* | A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource. |

```
{
    resources.addResource(
        renewable_type,
        path_2_resource_data,
        resource_key,
        &(this->electrical_load)
    );

    return;
}   /* addResource() */
```

#### 4.10.3.11 addSolar()

```
void Model::addSolar (
            SolarInputs solar_inputs )
```

Method to add a Solar asset to the Model.

**Parameters**

| | |
|---|---|
| *solar_inputs* | A structure of Solar constructor inputs. |

```
{
    Renewable* solar_ptr = new Solar(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        solar_inputs
    );

    this->renewable_ptr_vec.push_back(solar_ptr);

    return;
}   /* addSolar() */
```

#### 4.10.3.12 addTidal()

```
void Model::addTidal (
            TidalInputs tidal_inputs )
```

Method to add a Tidal asset to the Model.

**Parameters**

| | |
|---|---|
| *tidal_inputs* | A structure of Tidal constructor inputs. |

```
{
    Renewable* tidal_ptr = new Tidal(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        tidal_inputs
    );

    this->renewable_ptr_vec.push_back(tidal_ptr);

    return;
}   /* addTidal() */
```

#### 4.10.3.13 addWave()

```
void Model::addWave (
            WaveInputs wave_inputs )
```

Method to add a Wave asset to the Model.

**Parameters**

| | |
|---|---|
| *wave_inputs* | A structure of Wave constructor inputs. |

```
{
    Renewable* wave_ptr = new Wave(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        wave_inputs
    );

    this->renewable_ptr_vec.push_back(wave_ptr);

    return;
}   /* addWave() */
```

#### 4.10.3.14 addWind()

```
void Model::addWind (
            WindInputs wind_inputs )
```

Method to add a Wind asset to the Model.

**Parameters**

| | |
|---|---|
| *wind_inputs* | A structure of Wind constructor inputs. |

```
{
    Renewable* wind_ptr = new Wind(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        wind_inputs
    );

    this->renewable_ptr_vec.push_back(wind_ptr);

    return;
}   /* addWind() */
```

#### 4.10.3.15 clear()

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

Method to clear all attributes of the Model object.

```
{
    //  1. reset
    this->reset();

    //  2. clear components
    controller.clear();
    electrical_load.clear();
    resources.clear();

    return;
}   /* clear() */
```

#### 4.10.3.16 reset()

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

Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors and resets select Model attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

```
{
    //  1. clear combustion_ptr_vec
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        delete this->combustion_ptr_vec[i];
    }
    this->combustion_ptr_vec.clear();

    //  2. clear renewable_ptr_vec
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        delete this->renewable_ptr_vec[i];
    }
    this->renewable_ptr_vec.clear();

    //  3. clear storage_ptr_vec
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        delete this->storage_ptr_vec[i];
    }
    this->storage_ptr_vec.clear();

    //  4. reset attributes
    this->total_fuel_consumed_L = 0;

    this->total_emissions.CO2_kg = 0;
    this->total_emissions.CO_kg = 0;
    this->total_emissions.NOx_kg = 0;
    this->total_emissions.SOx_kg = 0;
    this->total_emissions.CH4_kg = 0;
    this->total_emissions.PM_kg = 0;

    this->net_present_cost = 0;
    this->total_dispatch_discharge_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    return;
}   /* reset() */
```

#### 4.10.3.17 run()

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

A method to run the Model.

```
{
    // 1. init Controller
    this->controller.init(
        &(this->electrical_load),
        &(this->renewable_ptr_vec),
        &(this->resources),
        &(this->combustion_ptr_vec)
    );

    //  2. apply dispatch control
    this->controller.applyDispatchControl(
        &(this->electrical_load),
        &(this->combustion_ptr_vec),
        &(this->renewable_ptr_vec),
        &(this->storage_ptr_vec)
    );

    //  3. compute total fuel consumption and emissions
    this->__computeFuelAndEmissions();

    //  4. compute key economic metrics
    this->__computeEconomics();

    return;
}   /* run() */
```

#### 4.10.3.18 writeResults()

```
void Model::writeResults (
             std::string write_path,
             int max_lines = -1 )
```

Method which writes Model results to an output directory. Also calls out to writeResults() for each contained asset.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *max_lines* | The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written. |

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->electrical_load.n_points;
    }

    //  2. check for pre-existing, warn (and remove), then create
    if (write_path.back() != '/') {
        write_path += '/';
    }

    if (std::filesystem::is_directory(write_path)) {
        std::string warning_str = "WARNING:  Model::writeResults():  ";
        warning_str += write_path;
        warning_str += " already exists, contents will be overwritten!";

        std::cout « warning_str « std::endl;

        std::filesystem::remove_all(write_path);
    }

    std::filesystem::create_directory(write_path);
```

```
    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->electrical_load.n_points) {
        max_lines = this->electrical_load.n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(write_path, max_lines);
    }

    //  5. call out to Combustion :: writeResults()
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            i,
            max_lines
        );
    }

    //  6. call out to Renewable :: writeResults()
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->renewable_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            &(this->resources.resource_map_1D),
            &(this->resources.resource_map_2D),
            i,
            max_lines
        );
    }

    //  7. call out to Storage :: writeResults()
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        this->storage_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            i,
            max_lines
        );
    }

    return;
}   /* writeResults() */
```

### 4.10.4 Member Data Documentation

#### 4.10.4.1 combustion_ptr_vec

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

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

#### 4.10.4.2 controller

Controller Model::controller

Controller component of Model.

#### 4.10.4.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

ElectricalLoad component of Model.

#### 4.10.4.4 levellized_cost_of_energy_kWh

```
double Model::levellized_cost_of_energy_kWh
```

The levellized cost of energy, per unit energy dispatched/discharged, of the Model [1/kWh] (undefined currency).

#### 4.10.4.5 net_present_cost

```
double Model::net_present_cost
```

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

#### 4.10.4.6 renewable_ptr_vec

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

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

#### 4.10.4.7 resources

```
Resources Model::resources
```

Resources component of Model.

#### 4.10.4.8 storage_ptr_vec

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

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

#### 4.10.4.9 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

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

#### 4.10.4.10 total_emissions

```
Emissions Model::total_emissions
```

An Emissions structure for holding total emissions [kg].

#### 4.10.4.11 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

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

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

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

## 4.11 ModelInputs Struct Reference

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).

```
#include <Model.h>
```

### Public Attributes

- std::string path_2_electrical_load_time_series = ""

  *A string defining the path (either relative or absolute) to the given electrical load time series.*
- ControlMode control_mode = ControlMode :: LOAD_FOLLOWING

  *The control mode to be applied by the Controller object.*

### 4.11.1 Detailed Description

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).

### 4.11.2 Member Data Documentation

#### 4.11.2.1 control_mode

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

The control mode to be applied by the Controller object.

#### 4.11.2.2 path_2_electrical_load_time_series

`std::string ModelInputs::path_2_electrical_load_time_series = ""`

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

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

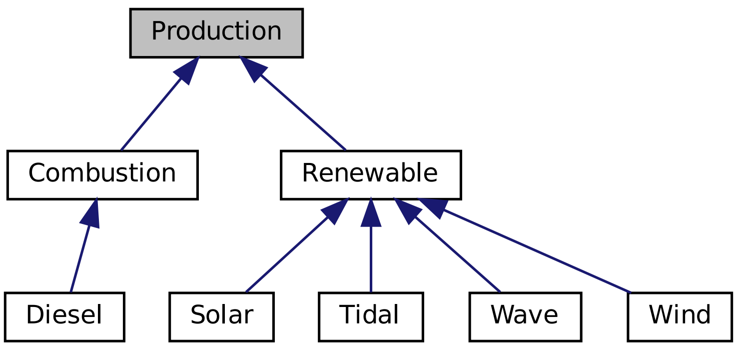
- header/Model.h

## 4.12 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, double, ProductionInputs)

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

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

    *Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.*
- virtual void computeEconomics (std::vector$<$ double $>$ $*$)

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

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

    *Destructor for the Production class.*

## Public Attributes

- bool print_flag

    *A flag which indicates whether or not object construct/destruction should be verbose.*
- bool is_running

    *A boolean which indicates whether or not the asset is running.*
- bool is_sunk

    *A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).*
- int n_points

    *The number of points in the modelling time series.*
- int n_starts

    *The number of times the asset has been started.*
- int n_replacements

    *The number of times the asset has been replaced.*
- double n_years

    *The number of years being modelled.*
- double running_hours

    *The number of hours for which the assset has been operating.*
- double replace_running_hrs

    *The number of running hours after which the asset must be replaced.*
- double capacity_kW

    *The rated production capacity [kW] of the asset.*
- double nominal_inflation_annual

    *The nominal, annual inflation rate to use in computing model economics.*
- double nominal_discount_annual

    *The nominal, annual discount rate to use in computing model economics.*
- double real_discount_annual

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

    *The capital cost of the asset (undefined currency).*

- double operation_maintenance_cost_kWh

  *The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.*
- double net_present_cost

  *The net present cost of this asset.*
- double total_dispatch_kWh

  *The total energy dispatched [kWh] over the Model run.*
- double levellized_cost_of_energy_kWh

  *The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatch.*
- std::string type_str

  *A string describing the type of the asset.*
- std::vector< bool > is_running_vec

  *A boolean vector for tracking if the asset is running at a particular point in time.*
- std::vector< double > production_vec_kW

  *A vector of production [kW] at each point in the modelling time series.*
- std::vector< double > dispatch_vec_kW

  *A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.*
- std::vector< double > storage_vec_kW

  *A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.*
- std::vector< double > curtailment_vec_kW

  *A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.*
- std::vector< double > capital_cost_vec

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

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

## Private Member Functions

- void __checkInputs (int, double, ProductionInputs)

  *Helper method to check inputs to the Production constructor.*

### 4.12.1 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.

### 4.12.2 Constructor & Destructor Documentation

#### 4.12.2.1 Production() [1/2]

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

Constructor (dummy) for the Production class.

```
{
    return;
}   /* Production() */
```

#### 4.12.2.2 Production() [2/2]

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

Constructor (intended) for the Production class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *production_inputs* | A structure of Production constructor inputs. |

```
{
    //  1. check inputs
    this->__checkInputs(n_points, n_years, production_inputs);

    //  2. set attributes
    this->print_flag = production_inputs.print_flag;
    this->is_running = false;
    this->is_sunk = production_inputs.is_sunk;

    this->n_points = n_points;
    this->n_starts = 0;
    this->n_replacements = 0;

    this->n_years = n_years;

    this->running_hours = 0;
    this->replace_running_hrs = production_inputs.replace_running_hrs;

    this->capacity_kW = production_inputs.capacity_kW;

    this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
    this->nominal_discount_annual = production_inputs.nominal_discount_annual;

    this->real_discount_annual = this->computeRealDiscountAnnual(
        production_inputs.nominal_inflation_annual,
        production_inputs.nominal_discount_annual
    );

    this->capital_cost = 0;
    this->operation_maintenance_cost_kWh = 0;
    this->net_present_cost = 0;
    this->total_dispatch_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    this->is_running_vec.resize(this->n_points, 0);

    this->production_vec_kW.resize(this->n_points, 0);
    this->dispatch_vec_kW.resize(this->n_points, 0);
    this->storage_vec_kW.resize(this->n_points, 0);
    this->curtailment_vec_kW.resize(this->n_points, 0);

    this->capital_cost_vec.resize(this->n_points, 0);
    this->operation_maintenance_cost_vec.resize(this->n_points, 0);

```

```
    // 3. construction print
    if (this->print_flag) {
        std::cout « "Production object constructed at " « this « std::endl;
    }

    return;
}   /* Production() */
```

#### 4.12.2.3 ∼Production()

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

Destructor for the Production class.

```
{
    // 1. destruction print
    if (this->print_flag) {
        std::cout « "Production object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Production() */
```

### 4.12.3 Member Function Documentation

#### 4.12.3.1 __checkInputs()

```
void Production::__checkInputs (
            int n_points,
            double n_years,
            ProductionInputs production_inputs ) [private]
```

Helper method to check inputs to the Production constructor.

**Parameters**

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

```
{
    // 1. check n_points
    if (n_points <= 0) {
        std::string error_str = "ERROR: Production(): n_points must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    // 2. check n_years
    if (n_years <= 0) {
        std::string error_str = "ERROR: Production(): n_years must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

```

```
            throw std::invalid_argument(error_str);
        }

        //  3. check capacity_kW
        if (production_inputs.capacity_kW <= 0) {
            std::string error_str = "ERROR:  Production():  ";
            error_str += "ProductionInputs::capacity_kW must be > 0";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::invalid_argument(error_str);
        }

        //  4. check replace_running_hrs
        if (production_inputs.replace_running_hrs <= 0) {
            std::string error_str = "ERROR:  Production():  ";
            error_str += "ProductionInputs::replace_running_hrs must be > 0";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::invalid_argument(error_str);
        }

        return;
}   /* __checkInputs() */
```

#### 4.12.3.2 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented in Wind, Wave, Tidal, Solar, Renewable, Diesel, and Combustion.

```
{
    //  1. record production
    this->production_vec_kW[timestep] = production_kW;

    //  2. compute and record dispatch and curtailment
    double dispatch_kW = 0;
    double curtailment_kW = 0;

    if (production_kW > load_kW) {
        dispatch_kW = load_kW;
        curtailment_kW = production_kW - dispatch_kW;
    }
```

```
    else {
        dispatch_kW = production_kW;
    }

    this->dispatch_vec_kW[timestep] = dispatch_kW;
    this->total_dispatch_kWh += dispatch_kW * dt_hrs;
    this->curtailment_vec_kW[timestep] = curtailment_kW;

    //  3. update load
    load_kW -= dispatch_kW;

    //  4. update and log running attributes
    if (this->is_running) {
        //  4.1. log running state, running hours
        this->is_running_vec[timestep] = this->is_running;
        this->running_hours += dt_hrs;

        //  4.2. incur operation and maintenance costs
        double produced_kWh = production_kW * dt_hrs;

        double operation_maintenance_cost =
            this->operation_maintenance_cost_kWh * produced_kWh;
        this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
    }

    //  5. trigger replacement, if applicable
    if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
        this->handleReplacement(timestep);
    }

    return load_kW;
}   /* commit() */
```

#### 4.12.3.3 computeEconomics()

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

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

Ref: HOMER [2023b]
Ref: HOMER [2023g]
Ref: HOMER [2023i]
Ref: HOMER [2023a]

**Parameters**

| | |
|---|---|
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |

1. compute levellized cost of energy (per unit dispatched)

Reimplemented in Renewable, and Combustion.

```
{
    //  1. compute net present cost
    double t_hrs = 0;
    double real_discount_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_discount_scalar = 1.0 / pow(
            1 + this->real_discount_annual,
            t_hrs / 8760
        );
```

```
        this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];

        this->net_present_cost +=
            real_discount_scalar * this->operation_maintenance_cost_vec[i];
    }

    //     assuming 8,760 hours per year
    double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;

    double capital_recovery_factor =
        (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
        (pow(1 + this->real_discount_annual, n_years) - 1);

    double total_annualized_cost = capital_recovery_factor *
        this->net_present_cost;

    this->levellized_cost_of_energy_kWh =
        (n_years * total_annualized_cost) /
        this->total_dispatch_kWh;

    return;
}   /* computeEconomics() */
```

#### 4.12.3.4 computeRealDiscountAnnual()

```
double Production::computeRealDiscountAnnual (
              double nominal_inflation_annual,
              double nominal_discount_annual )
```

Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

Ref: HOMER [2023h]
Ref: HOMER [2023b]

**Parameters**

| | |
|---|---|
| *nominal_inflation_annual* | The nominal, annual inflation rate to use in computing model economics. |
| *nominal_discount_annual* | The nominal, annual discount rate to use in computing model economics. |

**Returns**

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

```
{
    double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
    real_discount_annual /= 1 + nominal_inflation_annual;

    return real_discount_annual;
}   /* __computeRealDiscountAnnual() */
```

#### 4.12.3.5 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented in Wind, Wave, Tidal, Solar, Renewable, Diesel, and Combustion.

```
{
    // 1. reset attributes
    this->is_running = false;

    // 2. log replacement
    this->n_replacements++;

    // 3. incur capital cost in timestep
    this->capital_cost_vec[timestep] = this->capital_cost;

    return;
}   /* __handleReplacement() */
```

## 4.12.4 Member Data Documentation

### 4.12.4.1 capacity_kW

`double Production::capacity_kW`

The rated production capacity [kW] of the asset.

### 4.12.4.2 capital_cost

`double Production::capital_cost`

The capital cost of the asset (undefined currency).

### 4.12.4.3 capital_cost_vec

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

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

### 4.12.4.4 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.

#### 4.12.4.5 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.

#### 4.12.4.6 is_running

```
bool Production::is_running
```

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

#### 4.12.4.7 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.

#### 4.12.4.8 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).

#### 4.12.4.9 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 dispatch.

#### 4.12.4.10 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

#### 4.12.4.11 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

#### 4.12.4.12 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

#### 4.12.4.13 n_years

```
double Production::n_years
```

The number of years being modelled.

#### 4.12.4.14 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

#### 4.12.4.15 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

#### 4.12.4.16 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

#### 4.12.4.17 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.

#### 4.12.4.18 operation_maintenance_cost_vec

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

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

#### 4.12.4.19 print_flag

`bool Production::print_flag`

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

#### 4.12.4.20 production_vec_kW

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

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

#### 4.12.4.21 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.

#### 4.12.4.22 replace_running_hrs

`double Production::replace_running_hrs`

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

#### 4.12.4.23 running_hours

```
double Production::running_hours
```

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

#### 4.12.4.24 storage_vec_kW

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

A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.

#### 4.12.4.25 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

#### 4.12.4.26 type_str

```
std::string Production::type_str
```

A string describing the type of the asset.

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

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

## 4.13 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.*

## 4.13.1 Detailed Description

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

## 4.13.2 Member Data Documentation

### 4.13.2.1 capacity_kW

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

The rated production capacity [kW] of the asset.

### 4.13.2.2 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).

### 4.13.2.3 nominal_discount_annual

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

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

#### 4.13.2.4 nominal_inflation_annual

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

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

#### 4.13.2.5 print_flag

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

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

#### 4.13.2.6 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:

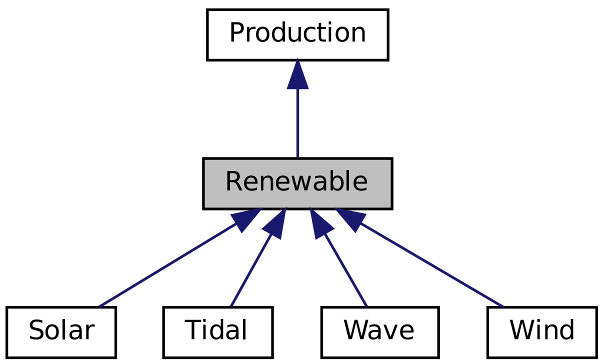
- header/Production/Production.h

## 4.14 Renewable Class Reference

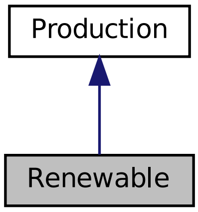
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#include <Renewable.h>
```

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:




## Public Member Functions

- Renewable (void)

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

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

  *Method to handle asset replacement and capital cost incursion, if applicable.*
- void computeEconomics (std::vector< double > ∗)

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

  *Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.*
- void writeResults (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std::map< int, std::vector< std::vector< double >>> ∗, int, int=-1)

  *Method which writes Renewable results to an output directory.*
- virtual ∼Renewable (void)

  *Destructor for the Renewable class.*

## Public Attributes

- RenewableType type

  *The type (RenewableType) of the asset.*
- int resource_key

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

## Private Member Functions

- void __checkInputs (RenewableInputs)

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

  *Helper method to handle the starting/stopping of the renewable asset.*
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std::map< int, std::vector< std::vector< double >>> ∗, int=-1)

### 4.14.1 Detailed Description

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

### 4.14.2 Constructor & Destructor Documentation

#### 4.14.2.1 Renewable() [1/2]

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

Constructor (dummy) for the Renewable class.

```
{
    //...

    return;
}   /* Renewable() */
```

#### 4.14.2.2 Renewable() [2/2]

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

Constructor (intended) for the Renewable class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *renewable_inputs* | A structure of Renewable constructor inputs. |

```
  :
Production(
    n_points,
    n_years,
    renewable_inputs.production_inputs
)
{
    //  1. check inputs
    this->__checkInputs(renewable_inputs);

    //  2. set attributes
    //...

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Renewable object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

#### 4.14.2.3 ∼Renewable()

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

Destructor for the Renewable class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Renewable object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Renewable() */
```

### 4.14.3 Member Function Documentation

#### 4.14.3.1 __checkInputs()

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

Helper method to check inputs to the Renewable constructor.

```
{
    //...

    return;
}   /* __checkInputs() */
```

#### 4.14.3.2 __handleStartStop()

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

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

```
{
    if (this->is_running) {
        // handle stopping
        if (production_kW <= 0) {
            this->is_running = false;
        }
    }

    else {
        // handle starting
        if (production_kW > 0) {
            this->is_running = true;
            this->n_starts++;
        }
    }

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

#### 4.14.3.3 __writeSummary()

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

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{return;}
```

#### 4.14.3.4 __writeTimeSeries()

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

Reimplemented in Wind, Wave, Tidal, and Solar.

```
            {return;}
```

#### 4.14.3.5 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{
    //  1. handle start/stop
    this->__handleStartStop(timestep, dt_hrs, production_kW);

```

```
    // 2. invoke base class method
    load_kW = Production :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

### 4.14.3.6 computeEconomics()

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

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

**Parameters**

| | |
|---|---|
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |

Reimplemented from Production.

```
{
    // 1. invoke base class method
    Production :: computeEconomics(time_vec_hrs_ptr);

    return;
}   /* computeEconomics() */
```

### 4.14.3.7 computeProductionkW() [1/2]

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

Reimplemented in Wind, Tidal, and Solar.

```
{return 0;}
```

### 4.14.3.8 computeProductionkW() [2/2]

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

Reimplemented in Wave.

```
{return 0;}
```

#### 4.14.3.9 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Production :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

#### 4.14.3.10 writeResults()

```
void Renewable::writeResults (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int renewable_index,
            int max_lines = -1 )
```

Method which writes Renewable results to an output directory.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |
| *resource_map_1D_ptr* | A pointer to the 1D map of Resources. |
| *resource_map_2D_ptr* | A pointer to the 2D map of Resources. |
| *renewable_index* | An integer which corresponds to the index of the Renewable asset in the Model. |
| *max_lines* | The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written. |

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Production/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
```

```
    }

    write_path += "Renewable/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->capacity_kW)));
    write_path += "kW_idx";
    write_path += std::to_string(renewable_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(
            write_path,
            time_vec_hrs_ptr,
            resource_map_1D_ptr,
            resource_map_2D_ptr,
            max_lines
        );
    }

    return;
}   /* writeResults() */
```

## 4.14.4 Member Data Documentation

### 4.14.4.1 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.

### 4.14.4.2 type

`RenewableType Renewable::type`

The type (RenewableType) of the asset.

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

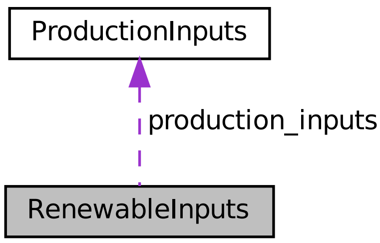
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

## 4.15 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.*

### 4.15.1 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.

### 4.15.2 Member Data Documentation

#### 4.15.2.1 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

## 4.16 Resources Class Reference

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

`#include <Resources.h>`

### Public Member Functions

- Resources (void)

  *Constructor for the Resources class.*
- void addResource (RenewableType, std::string, int, ElectricalLoad ∗)

  *A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).*
- void clear (void)

  *Method to clear all attributes of the Resources object.*
- ∼Resources (void)

  *Destructor for the Resources class.*

### Public Attributes

- std::map< int, std::vector< double > > resource_map_1D

  *A map <int, vector> of given 1D renewable resource time series.*
- std::map< int, std::string > string_map_1D

  *A map <int, string> of descriptors for the type of the given 1D renewable resource time series.*
- std::map< int, std::string > path_map_1D

  *A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.*
- std::map< int, std::vector< std::vector< double > > > resource_map_2D

  *A map <int, vector> of given 2D renewable resource time series.*
- std::map< int, std::string > string_map_2D

  *A map <int, string> of descriptors for the type of the given 2D renewable resource time series.*
- std::map< int, std::string > path_map_2D

  *A map <int, string> of the paths (either relative or absolute) to given 2D renewable resource time series.*

### Private Member Functions

- void __checkResourceKey1D (int, RenewableType)

  *Helper method to check if given resource key (1D) is already in use.*
- void __checkResourceKey2D (int, RenewableType)

  *Helper method to check if given resource key (2D) is already in use.*
- void __checkTimePoint (double, double, std::string, ElectricalLoad ∗)

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

  *Helper method to throw data length error.*
- void __readSolarResource (std::string, int, ElectricalLoad ∗)

  *Helper method to handle reading a solar resource time series into Resources.*
- void __readTidalResource (std::string, int, ElectricalLoad ∗)

  *Helper method to handle reading a tidal resource time series into Resources.*
- void __readWaveResource (std::string, int, ElectricalLoad ∗)

  *Helper method to handle reading a wave resource time series into Resources.*
- void __readWindResource (std::string, int, ElectricalLoad ∗)

  *Helper method to handle reading a wind resource time series into Resources.*

## 4.16.1 Detailed Description

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

## 4.16.2 Constructor & Destructor Documentation

### 4.16.2.1 Resources()

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

Constructor for the Resources class.

```
{
    return;
}   /* Resources() */
```

### 4.16.2.2 ∼Resources()

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

Destructor for the Resources class.

```
{
    this->clear();
    return;
}   /* ~Resources() */
```

## 4.16.3 Member Function Documentation

### 4.16.3.1 __checkResourceKey1D()

```
void Resources::__checkResourceKey1D (
        int resource_key,
        RenewableType renewable_type )  [private]
```

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

**Parameters**

| | |
|---|---|
| *resource_key* | The key associated with the given renewable resource. |

```
{
    if (this->resource_map_1D.count(resource_key) > 0) {
        std::string error_str = "ERROR:  Resources::addResource(";
```

```
        switch (renewable_type) {
            case (RenewableType :: SOLAR): {
                error_str += "SOLAR):  ";

                break;
            }

            case (RenewableType :: TIDAL): {
                error_str += "TIDAL):  ";

                break;
            }

            case (RenewableType :: WIND): {
                error_str += "WIND):  ";

                break;
            }

            default: {
                error_str += "UNDEFINED_TYPE):  ";

                break;
            }
        }

        error_str += "resource key (1D) ";
        error_str += std::to_string(resource_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkResourceKey1D() */
```

#### 4.16.3.2 __checkResourceKey2D()

```
void Resources::__checkResourceKey2D (
        int resource_key,
        RenewableType renewable_type )  [private]
```

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

**Parameters**

| | |
|---|---|
| *resource_key* | The key associated with the given renewable resource. |

```
{
    if (this->resource_map_2D.count(resource_key) > 0) {
        std::string error_str = "ERROR:  Resources::addResource(";

        switch (renewable_type) {
            case (RenewableType :: WAVE): {
                error_str += "WAVE):  ";

                break;
            }

            default: {
                error_str += "UNDEFINED_TYPE):  ";

                break;
            }
        }

        error_str += "resource key (2D) ";
        error_str += std::to_string(resource_key);
        error_str += " is already in use";
```

```
        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkResourceKey2D() */
```

#### 4.16.3.3 __checkTimePoint()

```
void Resources::__checkTimePoint (
            double time_received_hrs,
            double time_expected_hrs,
            std::string path_2_resource_data,
            ElectricalLoad * electrical_load_ptr ) [private]
```

Helper method to check received time point against expected time point.

**Parameters**

| | |
|---|---|
| *time_received_hrs* | The point in time received from the given data. |
| *time_expected_hrs* | The point in time expected (this comes from the electrical load time series). |
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    if (time_received_hrs != time_expected_hrs) {
        std::string error_str = "ERROR:  Resources::addResource():  ";
        error_str += "the given resource time series at ";
        error_str += path_2_resource_data;
        error_str += " does not align with the ";
        error_str += "previously given electrical load time series at ";
        error_str += electrical_load_ptr->path_2_electrical_load_time_series;

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::runtime_error(error_str);
    }

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

#### 4.16.3.4 __readSolarResource()

```
void Resources::__readSolarResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr ) [private]
```

Helper method to handle reading a solar resource time series into Resources.

**Parameters**

| | |
|---|---|
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *resource_key* | The key associated with the given renewable resource. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    // 1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Solar GHI [kW/m2]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));

    // 2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    // 3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double solar_resource_kWm2 = 0;

    while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;

        n_points++;
    }

    // 4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readSolarResource() */
```

#### 4.16.3.5 __readTidalResource()

```
void Resources::__readTidalResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to handle reading a tidal resource time series into Resources.

**Parameters**

| | |
|---|---|
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *resource_key* | The key associated with the given renewable resource. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    // 1. init CSV reader, record path and type
```

```
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Tidal Speed (hub depth) [m/s]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double tidal_resource_ms = 0;

    while (CSV.read_row(time_hrs, tidal_resource_ms)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readTidalResource() */
```

#### 4.16.3.6 __readWaveResource()

```
void Resources::__readWaveResource (
        std::string path_2_resource_data,
        int resource_key,
        ElectricalLoad * electrical_load_ptr )   [private]
```

Helper method to handle reading a wave resource time series into Resources.

**Parameters**

| | |
|---|---|
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *resource_key* | The key associated with the given renewable resource. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<3> CSV(path_2_resource_data);

```

```
    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Significant Wave Height [m]",
        "Energy Period [s]"
    );

    this->path_map_2D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));

    //  2. init map element
    this->resource_map_2D.insert(
        std::pair<int, std::vector<std::vector<double»>(resource_key, {})
    );
    this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double significant_wave_height_m = 0;
    double energy_period_s = 0;

    while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
        this->resource_map_2D[resource_key][n_points][1] = energy_period_s;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readWaveResource() */
```

#### 4.16.3.7 __readWindResource()

```
void Resources::__readWindResource (
            std::string path_2_resource_data,
            int resource_key,
            ElectricalLoad * electrical_load_ptr )   [private]
```

Helper method to handle reading a wind resource time series into Resources.

**Parameters**

| | |
|---|---|
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *resource_key* | The key associated with the given renewable resource. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);
```

```

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Wind Speed (hub height) [m/s]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double wind_resource_ms = 0;

    while (CSV.read_row(time_hrs, wind_resource_ms)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = wind_resource_ms;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readWindResource() */
```

#### 4.16.3.8 __throwLengthError()

```
void Resources::__throwLengthError (
            std::string path_2_resource_data,
            ElectricalLoad * electrical_load_ptr )  [private]
```

Helper method to throw data length error.

**Parameters**

| | |
|---|---|
| *path_2_resource_data* | The path (either relative or absolute) to the given resource time series. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    std::string error_str = "ERROR:  Resources::addResource():  ";
    error_str += "the given resource time series at ";
    error_str += path_2_resource_data;
    error_str += " is not the same length as the previously given electrical";
    error_str += " load time series at ";
    error_str += electrical_load_ptr->path_2_electrical_load_time_series;

```

```
    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);

    return;
}   /* __throwLengthError() */
```

### 4.16.3.9 addResource()

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

A method to add a renewable resource time series to Resources. Checks if given resource key is already in use. The associated helper methods also check against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

**Parameters**

| | |
|---|---|
| *renewable_type* | The type of renewable resource being added to Resources. |
| *path_2_resource_data* | A string defining the path (either relative or absolute) to the given resource time series. |
| *resource_key* | A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource. |
| *electrical_load_ptr* | A pointer to the Model's ElectricalLoad object. |

```
{
    switch (renewable_type) {
        case (RenewableType :: SOLAR): {
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readSolarResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: TIDAL): {
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readTidalResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: WAVE): {
            this->__checkResourceKey2D(resource_key, renewable_type);

            this->__readWaveResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: WIND): {
```

```
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readWindResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Resources :: addResource(:  ";
            error_str += "renewable type ";
            error_str += std::to_string(renewable_type);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return;
}   /* addResource() */
```

#### 4.16.3.10 clear()

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

Method to clear all attributes of the Resources object.

```
{
    this->resource_map_1D.clear();
    this->string_map_1D.clear();
    this->path_map_1D.clear();

    this->resource_map_2D.clear();
    this->string_map_2D.clear();
    this->path_map_2D.clear();

    return;
}   /* clear() */
```

### 4.16.4 Member Data Documentation

#### 4.16.4.1 path_map_1D

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

A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.

#### 4.16.4.2 path_map_2D

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

A map <int, string> of the paths (either relative or absolute) to given 2D renewable resource time series.

#### 4.16.4.3 resource_map_1D

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

A map <int, vector> of given 1D renewable resource time series.

#### 4.16.4.4 resource_map_2D

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

A map <int, vector> of given 2D renewable resource time series.

#### 4.16.4.5 string_map_1D

```
std::map<int, std::string> Resources::string_map_1D
```

A map <int, string> of descriptors for the type of the given 1D renewable resource time series.

#### 4.16.4.6 string_map_2D

```
std::map<int, std::string> Resources::string_map_2D
```

A map <int, string> of descriptors for the type of the given 2D renewable resource time series.

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

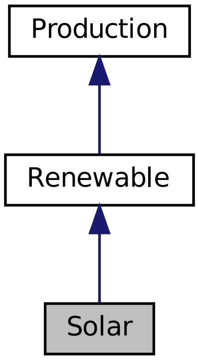
- header/Resources.h
- source/Resources.cpp

## 4.17 Solar Class Reference

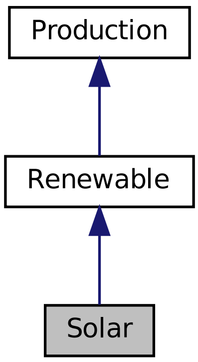
A derived class of the Renewable branch of Production which models solar production.

`#include <Solar.h>`

Inheritance diagram for Solar:




Collaboration diagram for Solar:




### Public Member Functions

- Solar (void)

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

*Constructor (intended) for the Solar class.*

- void handleReplacement (int)

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

  *Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.*
- double commit (int, double, double, double)

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

  *Destructor for the Solar class.*

## Public Attributes

- double derating

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

## Private Member Functions

- void __checkInputs (SolarInputs)

  *Helper method to check inputs to the Solar constructor.*
- double __getGenericCapitalCost (void)

  *Helper method to generate a generic solar PV array capital cost.*
- double __getGenericOpMaintCost (void)

  *Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.*
- void __writeSummary (std::string)

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

  *Helper method to write time series results for Solar.*

### 4.17.1 Detailed Description

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

### 4.17.2 Constructor & Destructor Documentation

#### 4.17.2.1 Solar() [1/2]

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

Constructor (dummy) for the Solar class.

```
{
    //...

    return;
}   /* Solar() */
```

#### 4.17.2.2 Solar() [2/2]

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

Constructor (intended) for the Solar class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *solar_inputs* | A structure of Solar constructor inputs. |

```
  :
Renewable(
    n_points,
    n_years,
    solar_inputs.renewable_inputs
)
{
    //  1. check inputs
    this->__checkInputs(solar_inputs);

    //  2. set attributes
    this->type = RenewableType :: SOLAR;
    this->type_str = "SOLAR";

    this->resource_key = solar_inputs.resource_key;

    this->derating = solar_inputs.derating;

    if (solar_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (solar_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Solar object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

#### 4.17.2.3 ∼Solar()

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

Destructor for the Solar class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Solar object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Solar() */
```

## 4.17.3 Member Function Documentation

### 4.17.3.1 __checkInputs()

```
void Solar::__checkInputs (
            SolarInputs solar_inputs )  [private]
```

Helper method to check inputs to the Solar constructor.

```
{
    //  1. check derating
    if (
        solar_inputs.derating < 0 or
        solar_inputs.derating > 1
    ) {
        std::string error_str = "ERROR:  Solar():  ";
        error_str += "SolarInputs::derating must be in the closed interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

### 4.17.3.2 __getGenericCapitalCost()

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

Helper method to generate a generic solar PV array capital cost.

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

**Returns**

A generic capital cost for the solar PV array [CAD].

```
{
    double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

### 4.17.3.3 __getGenericOpMaintCost()

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

Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.

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

**Returns**

A generic operation and maintenance cost, per unit energy produced, for the solar PV array [CAD/kWh].

```
{
    return 0.01;
}   /* __getGenericOpMaintCost() */
```

#### 4.17.3.4 __writeSummary()

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

Helper method to write summary results for Solar.

**Parameters**

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW SOLAR Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « "kW  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Solar attributes
    ofs « "## Solar Attributes\n";
    ofs « "\n";

    ofs « "Derating Factor: " « this->derating « "  \n";

    ofs « "\n--------\n\n";

    //  2.4. Solar Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";
```

```
    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __writeSummary() */
```

### 4.17.3.5 __writeTimeSeries()

```
void Solar::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Solar.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |
| *resource_map_1D_ptr* | A pointer to the 1D map of Resources. |
| *resource_map_2D_ptr* | A pointer to the 2D map of Resources. |
| *max_lines* | The maximum number of lines of output to write. |

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Solar Resource [kW/m2],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

#### 4.17.3.6 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

#### 4.17.3.7 computeProductionkW()

```
double Solar::computeProductionkW (
            int timestep,
            double dt_hrs,
            double solar_resource_kWm2 )  [virtual]
```

Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.

Ref: HOMER [2023f]

**Parameters**

| | |
|---|---|
| *timestep* | The timestep (i.e., time series index) for the request. |
| *dt_hrs* | The interval of time [hrs] associated with the timestep. |
| *solar_resource_kWm2* | Solar resource (i.e. irradiance) [kW/m2]. |

**Returns**

The production [kW] of the solar PV array.

Reimplemented from Renewable.

```
{
    // check if no resource
    if (solar_resource_kWm2 <= 0) {
        return 0;
    }

    // compute production
    double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;

    // cap production at capacity
    if (production_kW > this->capacity_kW) {
        production_kW = this->capacity_kW;
    }

    return production_kW;
}   /* computeProductionkW() */
```

#### 4.17.3.8 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

### 4.17.4 Member Data Documentation

#### 4.17.4.1 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:

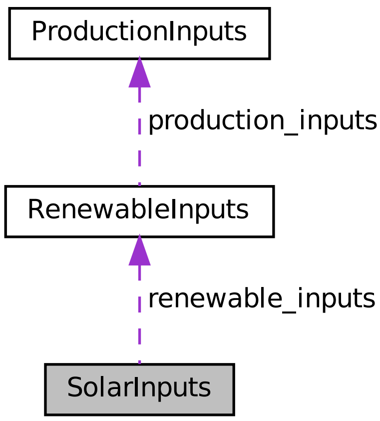
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

## 4.18 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.).*

### 4.18.1 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.

## 4.18.2 Member Data Documentation

### 4.18.2.1 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].

### 4.18.2.2 derating

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

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

### 4.18.2.3 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].

### 4.18.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

### 4.18.2.5 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:

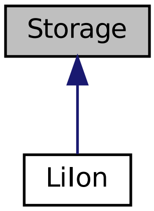
- header/Production/Renewable/Solar.h

## 4.19 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 (dummy) for the Storage class.*
- Storage (int, double, StorageInputs)

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

    *Method to handle asset replacement and capital cost incursion, if applicable.*
- void computeEconomics (std::vector< double > ∗)

    *Helper method to compute key economic metrics for the Model run.*
- virtual double getAvailablekW (double)
- virtual double getAcceptablekW (double)
- virtual void commitCharge (int, double, double)
- virtual double commitDischarge (int, double, double, double)
- void writeResults (std::string, std::vector< double > ∗, int, int=-1)

    *Method which writes Storage results to an output directory.*
- virtual ∼Storage (void)

    *Destructor for the Storage class.*

### Public Attributes

- StorageType type

    *The type (StorageType) of the asset.*
- bool print_flag

    *A flag which indicates whether or not object construct/destruction should be verbose.*
- bool is_depleted

    *A boolean which indicates whether or not the asset is currently considered depleted.*
- bool is_sunk

*A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).*

- int n_points

  *The number of points in the modelling time series.*
- int n_replacements

  *The number of times the asset has been replaced.*
- double n_years

  *The number of years being modelled.*
- double power_capacity_kW

  *The rated power capacity [kW] of the asset.*
- double energy_capacity_kWh

  *The rated energy capacity [kWh] of the asset.*
- double charge_kWh

  *The energy [kWh] stored in the asset.*
- double power_kW

  *The power [kW] currently being charged/discharged by the asset.*
- double nominal_inflation_annual

  *The nominal, annual inflation rate to use in computing model economics.*
- double nominal_discount_annual

  *The nominal, annual discount rate to use in computing model economics.*
- double real_discount_annual

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

  *The capital cost of the asset (undefined currency).*
- double operation_maintenance_cost_kWh

  *The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged.*
- double net_present_cost

  *The net present cost of this asset.*
- double total_discharge_kWh

  *The total energy discharged [kWh] over the Model run.*
- double levellized_cost_of_energy_kWh

  *The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only discharge.*
- std::string type_str

  *A string describing the type of the asset.*
- std::vector< double > charge_vec_kWh

  *A vector of the charge state [kWh] at each point in the modelling time series.*
- std::vector< double > charging_power_vec_kW

  *A vector of the charging power [kW] at each point in the modelling time series.*
- std::vector< double > discharging_power_vec_kW

  *A vector of the discharging power [kW] at each point in the modelling time series.*
- std::vector< double > capital_cost_vec

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

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

## Private Member Functions

- void __checkInputs (int, double, StorageInputs)

  *Helper method to check inputs to the Storage constructor.*
- double __computeRealDiscountAnnual (double, double)

  *Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.*
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

## 4.19.1 Detailed Description

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

## 4.19.2 Constructor & Destructor Documentation

### 4.19.2.1 Storage() [1/2]

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

Constructor (dummy) for the Storage class.

```
{
    return;
}   /* Storage() */
```

### 4.19.2.2 Storage() [2/2]

```
Storage::Storage (
            int n_points,
            double n_years,
            StorageInputs storage_inputs )
```

Constructor (intended) for the Storage class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *storage_inputs* | A structure of Storage constructor inputs. |

```
{
    //  1. check inputs
    this->__checkInputs(n_points, n_years, storage_inputs);

    //  2. set attributes
    this->print_flag = storage_inputs.print_flag;
    this->is_depleted = false;
```

```
    this->is_sunk = storage_inputs.is_sunk;

    this->n_points = n_points;
    this->n_replacements = 0;

    this->n_years = n_years;

    this->power_capacity_kW = storage_inputs.power_capacity_kW;
    this->energy_capacity_kWh = storage_inputs.energy_capacity_kWh;

    this->charge_kWh = 0;
    this->power_kW = 0;

    this->nominal_inflation_annual = storage_inputs.nominal_inflation_annual;
    this->nominal_discount_annual = storage_inputs.nominal_discount_annual;

    this->real_discount_annual = this->__computeRealDiscountAnnual(
        storage_inputs.nominal_inflation_annual,
        storage_inputs.nominal_discount_annual
    );

    this->capital_cost = 0;
    this->operation_maintenance_cost_kWh = 0;
    this->net_present_cost = 0;
    this->total_discharge_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    this->charge_vec_kWh.resize(this->n_points, 0);
    this->charging_power_vec_kW.resize(this->n_points, 0);
    this->discharging_power_vec_kW.resize(this->n_points, 0);

    this->capital_cost_vec.resize(this->n_points, 0);
    this->operation_maintenance_cost_vec.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Storage object constructed at " « this « std::endl;
    }

    return;
}   /* Storage() */
```

#### 4.19.2.3 ∼Storage()

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

Destructor for the Storage class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Storage object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Storage() */
```

### 4.19.3 Member Function Documentation

#### 4.19.3.1 __checkInputs()

```
void Storage::__checkInputs (
              int n_points,
              double n_years,
              StorageInputs storage_inputs )  [private]
```

Helper method to check inputs to the Storage constructor.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *storage_inputs* | A structure of Storage constructor inputs. |

```
{
    //  1. check n_points
    if (n_points <= 0) {
        std::string error_str = "ERROR:  Storage():  n_points must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check n_years
    if (n_years <= 0) {
        std::string error_str = "ERROR:  Storage():  n_years must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check power_capacity_kW
    if (storage_inputs.power_capacity_kW <= 0) {
        std::string error_str = "ERROR:  Storage():  ";
        error_str += "StorageInputs::power_capacity_kW must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check energy_capacity_kWh
    if (storage_inputs.energy_capacity_kWh <= 0) {
        std::string error_str = "ERROR:  Storage():  ";
        error_str += "StorageInputs::energy_capacity_kWh must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

### 4.19.3.2 __computeRealDiscountAnnual()

```
double Storage::__computeRealDiscountAnnual (
            double nominal_inflation_annual,
            double nominal_discount_annual )   [private]
```

Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

Ref: HOMER [2023h]
Ref: HOMER [2023b]

**Parameters**

| | |
|---|---|
| *nominal_inflation_annual* | The nominal, annual inflation rate to use in computing model economics. |
| *nominal_discount_annual* | The nominal, annual discount rate to use in computing model economics. |

**Returns**

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

```
{
    double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
    real_discount_annual /= 1 + nominal_inflation_annual;

    return real_discount_annual;
}   /* __computeRealDiscountAnnual() */
```

#### 4.19.3.3 __writeSummary()

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

Reimplemented in LiIon.

```
{return;}
```

#### 4.19.3.4 __writeTimeSeries()

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

Reimplemented in LiIon.

```
{return;}
```

#### 4.19.3.5 commitCharge()

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

Reimplemented in LiIon.

```
{return;}
```

#### 4.19.3.6 commitDischarge()

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

Reimplemented in LiIon.

```
{return 0;}
```

#### 4.19.3.7 computeEconomics()

```
void Storage::computeEconomics (
            std::vector< double > * time_vec_hrs_ptr )
```

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

Ref: HOMER [2023b]
Ref: HOMER [2023g]
Ref: HOMER [2023i]
Ref: HOMER [2023a]

**Parameters**

| | |
|---|---|
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |

1. compute levellized cost of energy (per unit discharged)

```
{
    //  1. compute net present cost
    double t_hrs = 0;
    double real_discount_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_discount_scalar = 1.0 / pow(
            1 + this->real_discount_annual,
            t_hrs / 8760
        );

        this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];

        this->net_present_cost +=
            real_discount_scalar * this->operation_maintenance_cost_vec[i];
    }

    //     assuming 8,760 hours per year
    double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;

    double capital_recovery_factor =
        (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
        (pow(1 + this->real_discount_annual, n_years) - 1);

    double total_annualized_cost = capital_recovery_factor *
        this->net_present_cost;

    this->levellized_cost_of_energy_kWh =
        (n_years * total_annualized_cost) /
        this->total_discharge_kWh;

    return;
}   /* computeEconomics() */
```

#### 4.19.3.8 getAcceptablekW()

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

Reimplemented in LiIon.

```
{return 0;}
```

#### 4.19.3.9 getAvailablekW()

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

Reimplemented in LiIon.

```
{return 0;}
```

#### 4.19.3.10 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented in LiIon.

```
{
    //  1. reset attributes
    this->charge_kWh = 0;
    this->power_kW = 0;

    //  2. log replacement
    this->n_replacements++;

    //  3. incur capital cost in timestep
    this->capital_cost_vec[timestep] = this->capital_cost;

    return;
}   /* __handleReplacement() */
```

#### 4.19.3.11 writeResults()

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

Method which writes Storage results to an output directory.

**Parameters**

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

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Storage/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->power_capacity_kW)));
    write_path += "kW_";
    write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
    write_path += "kWh_idx";
    write_path += std::to_string(storage_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(
            write_path,
            time_vec_hrs_ptr,
            max_lines
        );
    }

    return;
}   /* writeResults() */
```

## 4.19.4 Member Data Documentation

### 4.19.4.1 capital_cost

`double Storage::capital_cost`

The capital cost of the asset (undefined currency).

### 4.19.4.2 capital_cost_vec

`std::vector<double> Storage::capital_cost_vec`

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

#### 4.19.4.3 charge_kWh

```
double Storage::charge_kWh
```

The energy [kWh] stored in the asset.

#### 4.19.4.4 charge_vec_kWh

```
std::vector<double> Storage::charge_vec_kWh
```

A vector of the charge state [kWh] at each point in the modelling time series.

#### 4.19.4.5 charging_power_vec_kW

```
std::vector<double> Storage::charging_power_vec_kW
```

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

#### 4.19.4.6 discharging_power_vec_kW

```
std::vector<double> Storage::discharging_power_vec_kW
```

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

#### 4.19.4.7 energy_capacity_kWh

```
double Storage::energy_capacity_kWh
```

The rated energy capacity [kWh] of the asset.

#### 4.19.4.8 is_depleted

```
bool Storage::is_depleted
```

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

#### 4.19.4.9 is_sunk

```
bool Storage::is_sunk
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

#### 4.19.4.10 levellized_cost_of_energy_kWh

```
double Storage::levellized_cost_of_energy_kWh
```

The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only discharge.

#### 4.19.4.11 n_points

```
int Storage::n_points
```

The number of points in the modelling time series.

#### 4.19.4.12 n_replacements

```
int Storage::n_replacements
```

The number of times the asset has been replaced.

#### 4.19.4.13 n_years

```
double Storage::n_years
```

The number of years being modelled.

#### 4.19.4.14 net_present_cost

```
double Storage::net_present_cost
```

The net present cost of this asset.

#### 4.19.4.15 nominal_discount_annual

```
double Storage::nominal_discount_annual
```

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

#### 4.19.4.16 nominal_inflation_annual

```
double Storage::nominal_inflation_annual
```

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

#### 4.19.4.17 operation_maintenance_cost_kWh

```
double Storage::operation_maintenance_cost_kWh
```

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged.

#### 4.19.4.18 operation_maintenance_cost_vec

```
std::vector<double> Storage::operation_maintenance_cost_vec
```

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

#### 4.19.4.19 power_capacity_kW

```
double Storage::power_capacity_kW
```

The rated power capacity [kW] of the asset.

#### 4.19.4.20 power_kW

```
double Storage::power_kW
```

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

#### 4.19.4.21 print_flag

```
bool Storage::print_flag
```

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

#### 4.19.4.22 real_discount_annual

```
double Storage::real_discount_annual
```

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

#### 4.19.4.23 total_discharge_kWh

```
double Storage::total_discharge_kWh
```

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

#### 4.19.4.24 type

```
StorageType Storage::type
```

The type (StorageType) of the asset.

#### 4.19.4.25 type_str

```
std::string Storage::type_str
```

A string describing the type of the asset.

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

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

## 4.20 StorageInputs Struct Reference

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

```
#include <Storage.h>
```

## Public Attributes

- bool print_flag = false

  *A flag which indicates whether or not object construct/destruction should be verbose.*
- bool is_sunk = false

  *A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).*
- double power_capacity_kW = 100

  *The rated power capacity [kW] of the asset.*
- double energy_capacity_kWh = 1000

  *The rated energy capacity [kWh] of the asset.*
- double nominal_inflation_annual = 0.02

  *The nominal, annual inflation rate to use in computing model economics.*
- double nominal_discount_annual = 0.04

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

## 4.20.1 Detailed Description

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

## 4.20.2 Member Data Documentation

### 4.20.2.1 energy_capacity_kWh

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

The rated energy capacity [kWh] of the asset.

### 4.20.2.2 is_sunk

```
bool StorageInputs::is_sunk = false
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

### 4.20.2.3 nominal_discount_annual

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

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

#### 4.20.2.4 nominal_inflation_annual

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

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

#### 4.20.2.5 power_capacity_kW

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

The rated power capacity [kW] of the asset.

#### 4.20.2.6 print_flag

```
bool StorageInputs::print_flag = false
```

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

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

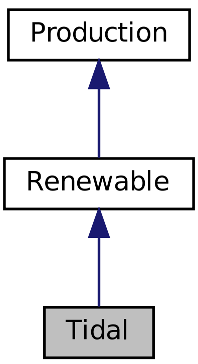
- header/Storage/Storage.h

## 4.21 Tidal Class Reference

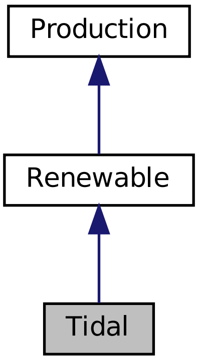
A derived class of the Renewable branch of Production which models tidal production.

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

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:




## Public Member Functions

- Tidal (void)

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

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

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

  *Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.*
- double commit (int, double, double, double)

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

  *Destructor for the Tidal class.*

## Public Attributes

- double design_speed_ms

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

  *The tidal power production model to be applied.*
- std::string power_model_string

  *A string describing the active power production model.*

## Private Member Functions

- void __checkInputs (TidalInputs)

    *Helper method to check inputs to the Tidal constructor.*
- double __getGenericCapitalCost (void)

    *Helper method to generate a generic tidal turbine capital cost.*
- double __getGenericOpMaintCost (void)

    *Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.*
- double __computeCubicProductionkW (int, double, double)

    *Helper method to compute tidal turbine production under a cubic production model.*
- double __computeExponentialProductionkW (int, double, double)

    *Helper method to compute tidal turbine production under an exponential production model.*
- double __computeLookupProductionkW (int, double, double)

    *Helper method to compute tidal turbine production by way of looking up using given power curve data.*
- void __writeSummary (std::string)

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

    *Helper method to write time series results for Tidal.*

### 4.21.1 Detailed Description

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

### 4.21.2 Constructor & Destructor Documentation

#### 4.21.2.1 Tidal() [1/2]

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

Constructor (dummy) for the Tidal class.

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

#### 4.21.2.2 Tidal() [2/2]

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

Constructor (intended) for the Tidal class.

#### Parameters

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *tidal_inputs* | A structure of Tidal constructor inputs. |

```
  :
Renewable(
    n_points,
    n_years,
    tidal_inputs.renewable_inputs
)
{
    //  1. check inputs
    this->__checkInputs(tidal_inputs);

    //  2. set attributes
    this->type = RenewableType :: TIDAL;
    this->type_str = "TIDAL";

    this->resource_key = tidal_inputs.resource_key;

    this->design_speed_ms = tidal_inputs.design_speed_ms;

    this->power_model = tidal_inputs.power_model;

    switch (this->power_model) {
        case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
            this->power_model_string = "CUBIC";

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
            this->power_model_string = "EXPONENTIAL";

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Tidal():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    if (tidal_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Tidal object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

#### 4.21.2.3 ∼Tidal()

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

Destructor for the Tidal class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Tidal object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Tidal() */
```

### 4.21.3 Member Function Documentation

#### 4.21.3.1 __checkInputs()

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

Helper method to check inputs to the Tidal constructor.

```
{
    //  1. check design_speed_ms
    if (tidal_inputs.design_speed_ms <= 0) {
        std::string error_str = "ERROR:  Tidal():  ";
        error_str += "TidalInputs::design_speed_ms must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

#### 4.21.3.2 __computeCubicProductionkW()

```
double Tidal::__computeCubicProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production under a cubic production model.

Ref: Buckham et al. [2023]

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *tidal_resource_ms* | The available tidal stream resource [m/s]. |

**Returns**

The production [kW] of the tidal turbine, under a cubic model.

```
{
    double production = 0;

    if (
        tidal_resource_ms < 0.15 * this->design_speed_ms or
        tidal_resource_ms > 1.25 * this->design_speed_ms
    ){
        production = 0;
    }

    else if (
        0.15 * this->design_speed_ms <= tidal_resource_ms and
        tidal_resource_ms <= this->design_speed_ms
    ) {
        production =
            (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
    }

    else {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __computeCubicProductionkW() */
```

### 4.21.3.3 __computeExponentialProductionkW()

```
double Tidal::__computeExponentialProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production under an exponential production model.

Ref: Truelove et al. [2019]

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *tidal_resource_ms* | The available tidal stream resource [m/s]. |

**Returns**

The production [kW] of the tidal turbine, under an exponential model.

```
{
    double production = 0;

    double turbine_speed =
        (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;

    if (turbine_speed < -0.71 or turbine_speed > 0.65) {
        production = 0;
    }

    else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
        production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
    }

    else {
        production = 1;
    }
```

```
    return production * this->capacity_kW;
}   /* __computeExponentialProductionkW() */
```

#### 4.21.3.4 __computeLookupProductionkW()

```
double Tidal::__computeLookupProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms )  [private]
```

Helper method to compute tidal turbine production by way of looking up using given power curve data.

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *tidal_resource_ms* | The available tidal stream resource [m/s]. |

**Returns**

The interpolated production [kW] of the tidal tubrine.

```
{
    // *** WORK IN PROGRESS *** //

    return 0;
}   /* __computeLookupProductionkW() */
```

#### 4.21.3.5 __getGenericCapitalCost()

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

Helper method to generate a generic tidal turbine capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: MacDougall [2019]

**Returns**

A generic capital cost for the tidal turbine [CAD].

```
{
    double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

#### 4.21.3.6 __getGenericOpMaintCost()

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

Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: MacDougall [2019]

**Returns**

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

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

#### 4.21.3.7 __writeSummary()

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

Helper method to write summary results for Tidal.

**Parameters**

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW TIDAL Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « "kW  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
```

```
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Tidal attributes
    ofs « "## Tidal Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";

    ofs « "\n--------\n\n";

    //  2.4. Tidal Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __writeSummary() */
```

#### 4.21.3.8 __writeTimeSeries()

```
void Tidal::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Tidal.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |
| *resource_map_1D_ptr* | A pointer to the 1D map of Resources. |
| *resource_map_2D_ptr* | A pointer to the 2D map of Resources. |
| *max_lines* | The maximum number of lines of output to write. |

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Tidal Resource [m/s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    return;
}   /* __writeTimeSeries() */
```

#### 4.21.3.9 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


```

```
    //...

    return load_kW;
}   /* commit() */
```

#### 4.21.3.10 computeProductionkW()

```
double Tidal::computeProductionkW (
            int timestep,
            double dt_hrs,
            double tidal_resource_ms ) [virtual]
```

Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.

**Parameters**

| | |
|---|---|
| *timestep* | The timestep (i.e., time series index) for the request. |
| *dt_hrs* | The interval of time [hrs] associated with the timestep. |
| *tidal_resource_ms* | Tidal resource (i.e. tidal stream speed) [m/s]. |

**Returns**

The production [kW] of the tidal turbine.

Reimplemented from Renewable.

```
{
    // check if no resource
    if (tidal_resource_ms <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
            production_kW = this->__computeCubicProductionkW(
                timestep,
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }


        case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
            production_kW = this->__computeExponentialProductionkW(
                timestep,
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }
```

```
        default: {
            std::string error_str = "ERROR:  Tidal::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

#### 4.21.3.11 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

### 4.21.4 Member Data Documentation

#### 4.21.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

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

#### 4.21.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

#### 4.21.4.3 power_model_string

```
std::string Tidal::power_model_string
```

A string describing the active power production model.

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

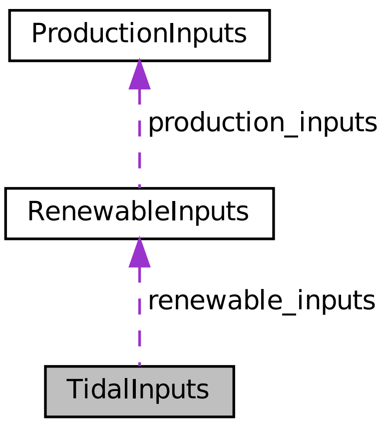
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

## 4.22 TidalInputs Struct Reference

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

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

Collaboration diagram for TidalInputs:




### Public Attributes

- RenewableInputs renewable_inputs
  *An encapsulated RenewableInputs instance.*
- int resource_key = 0
  *A key used to index into the Resources object, to associate this asset with the appropriate resource time series.*
- double capital_cost = -1
  *The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].*
- double operation_maintenance_cost_kWh = -1

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

- double design_speed_ms = 3

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

- TidalPowerProductionModel power_model = TidalPowerProductionModel :: TIDAL_POWER_CUBIC

  *The tidal power production model to be applied.*

### 4.22.1 Detailed Description

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

### 4.22.2 Member Data Documentation

#### 4.22.2.1 capital_cost

```
double TidalInputs::capital_cost = -1
```

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

#### 4.22.2.2 design_speed_ms

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

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

#### 4.22.2.3 operation_maintenance_cost_kWh

```
double TidalInputs::operation_maintenance_cost_kWh = -1
```

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

#### 4.22.2.4 power_model

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

The tidal power production model to be applied.

#### 4.22.2.5 renewable_inputs

`RenewableInputs TidalInputs::renewable_inputs`

An encapsulated RenewableInputs instance.

#### 4.22.2.6 resource_key

`int TidalInputs::resource_key = 0`

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

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

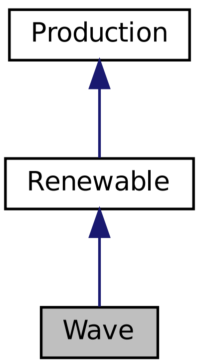
- header/Production/Renewable/Tidal.h

## 4.23 Wave Class Reference

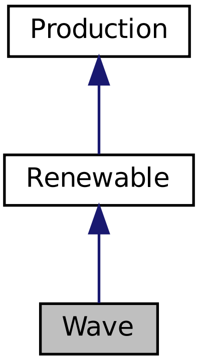
A derived class of the Renewable branch of Production which models wave production.

`#include <Wave.h>`

Inheritance diagram for Wave:



Collaboration diagram for Wave:




## Public Member Functions

- Wave (void)

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

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

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

    *Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.*
- double commit (int, double, double, double)

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

    *Destructor for the Wave class.*

## Public Attributes

- double design_significant_wave_height_m

    *The significant wave height [m] at which the wave energy converter achieves its rated capacity.*
- double design_energy_period_s

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

    *The wave power production model to be applied.*
- std::string power_model_string

    *A string describing the active power production model.*

## Private Member Functions

- void __checkInputs (WaveInputs)
  *Helper method to check inputs to the Wave constructor.*
- double __getGenericCapitalCost (void)
  *Helper method to generate a generic wave energy converter capital cost.*
- double __getGenericOpMaintCost (void)
  *Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.*
- double __computeGaussianProductionkW (int, double, double, double)
  *Helper method to compute wave energy converter production under a Gaussian production model.*
- double __computeParaboloidProductionkW (int, double, double, double)
  *Helper method to compute wave energy converter production under a paraboloid production model.*
- double __computeLookupProductionkW (int, double, double, double)
  *Helper method to compute wave energy converter production by way of looking up using given performance matrix.*
- void __writeSummary (std::string)
  *Helper method to write summary results for Wave.*
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
  *Helper method to write time series results for Wave.*

## 4.23.1 Detailed Description

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

## 4.23.2 Constructor & Destructor Documentation

### 4.23.2.1 Wave() [1/2]

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

Constructor (dummy) for the Wave class.

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

### 4.23.2.2 Wave() [2/2]

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

Constructor (intended) for the Wave class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *wave_inputs* | A structure of Wave constructor inputs. |

```
  :
Renewable(
    n_points,
    n_years,
    wave_inputs.renewable_inputs
)
{
    //  1. check inputs
    this->__checkInputs(wave_inputs);

    //  2. set attributes
    this->type = RenewableType :: WAVE;
    this->type_str = "WAVE";

    this->resource_key = wave_inputs.resource_key;

    this->design_significant_wave_height_m =
        wave_inputs.design_significant_wave_height_m;
    this->design_energy_period_s = wave_inputs.design_energy_period_s;

    this->power_model = wave_inputs.power_model;

    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            this->power_model_string = "GAUSSIAN";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
            this->power_model_string = "PARABOLOID";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wave():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    if (wave_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (wave_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Wave object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

#### 4.23.2.3 ∼Wave()

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

Destructor for the Wave class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Wave object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Wave() */
```

### 4.23.3 Member Function Documentation

#### 4.23.3.1 __checkInputs()

```
void Wave::__checkInputs (
            WaveInputs wave_inputs ) [private]
```

Helper method to check inputs to the Wave constructor.

**Parameters**

| | |
|---|---|
| *wave_inputs* | A structure of Wave constructor inputs. |

```
{
    //  1. check design_significant_wave_height_m
    if (wave_inputs.design_significant_wave_height_m <= 0) {
        std::string error_str = "ERROR:  Wave():  ";
        error_str += "WaveInputs::design_significant_wave_height_m must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check design_energy_period_s
    if (wave_inputs.design_energy_period_s <= 0) {
        std::string error_str = "ERROR:  Wave():  ";
        error_str += "WaveInputs::design_energy_period_s must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

#### 4.23.3.2 __computeGaussianProductionkW()

```
double Wave::__computeGaussianProductionkW (
            int timestep,
```

```
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [private]
```

Helper method to compute wave energy converter production under a Gaussian production model.

Ref: Truelove et al. [2019]

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *significant_wave_height←_m* | The significant wave height [m] in the vicinity of the wave energy converter. |
| *energy_period_s* | The energy period [s] in the vicinity of the wave energy converter |

**Returns**

The production [kW] of the wave energy converter, under an exponential model.

```
{
    double H_s_nondim =
        (significant_wave_height_m - this->design_significant_wave_height_m) /
        this->design_significant_wave_height_m;

    double T_e_nondim =
        (energy_period_s - this->design_energy_period_s) /
        this->design_energy_period_s;

    double production = exp(
        -2.25119 * pow(T_e_nondim, 2) +
        3.44570 * T_e_nondim * H_s_nondim -
        4.01508 * pow(H_s_nondim, 2)
    );

    return production * this->capacity_kW;
}   /* __computeGaussianProductionkW() */
```

### 4.23.3.3 __computeLookupProductionkW()

```
double Wave::__computeLookupProductionkW (
            int timestep,
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [private]
```

Helper method to compute wave energy converter production by way of looking up using given performance matrix.

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *significant_wave_height←_m* | The significant wave height [m] in the vicinity of the wave energy converter. |
| *energy_period_s* | The energy period [s] in the vicinity of the wave energy converter |

**Returns**

The interpolated production [kW] of the wave energy converter.

```
{
    // *** WORK IN PROGRESS *** //

    return 0;
}   /* __computeLookupProductionkW() */
```

#### 4.23.3.4 __computeParaboloidProductionkW()

```
double Wave::__computeParaboloidProductionkW (
            int timestep,
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [private]
```

Helper method to compute wave energy converter production under a paraboloid production model.

Ref: Robertson et al. [2021]

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *significant_wave_height↩_m* | The significant wave height [m] in the vicinity of the wave energy converter. |
| *energy_period_s* | The energy period [s] in the vicinity of the wave energy converter |

**Returns**

The production [kW] of the wave energy converter, under a paraboloid model.

```
{
    // first, check for idealized wave breaking (deep water)
    if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
        return 0;
    }

    // otherwise, apply generic quadratic performance model
    // (with outputs bounded to [0, 1])
    double production =
        0.289 * significant_wave_height_m -
        0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
        0.0169 * energy_period_s;

    if (production < 0) {
        production = 0;
    }

    else if (production > 1) {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __computeParaboloidProductionkW() */
```

#### 4.23.3.5 __getGenericCapitalCost()

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

Helper method to generate a generic wave energy converter capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: MacDougall [2019]

**Returns**

A generic capital cost for the wave energy converter [CAD].

```
{
    double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

#### 4.23.3.6 __getGenericOpMaintCost()

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

Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: MacDougall [2019]

**Returns**

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

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

#### 4.23.3.7 __writeSummary()

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

Helper method to write summary results for Wave.

**Parameters**

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW WAVE Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « "kW  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (2D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Wave attributes
    ofs « "## Wave Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            ofs « "Design Significant Wave Height: "
                « this->design_significant_wave_height_m « " m  \n";

            ofs « "Design Energy Period: " « this->design_energy_period_s « " s  \n";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            //...

            break;
        }

        default: {
            // write nothing!

            break;
        }
    }

    ofs « "\n--------\n\n";

    //  2.4. Wave Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";
```

```
    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __writeSummary() */
```

#### 4.23.3.8 __writeTimeSeries()

```
void Wave::__writeTimeSeries (
              std::string write_path,
              std::vector< double > * time_vec_hrs_ptr,
              std::map< int, std::vector< double >> * resource_map_1D_ptr,
              std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
              int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Wave.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |
| *resource_map_1D_ptr* | A pointer to the 1D map of Resources. |
| *resource_map_2D_ptr* | A pointer to the 2D map of Resources. |
| *max_lines* | The maximum number of lines of output to write. |

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Significant Wave Height [m],";
    ofs « "Energy Period [s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « resource_map_2D_ptr->at(this->resource_key)[i][0] « ",";
        ofs « resource_map_2D_ptr->at(this->resource_key)[i][1] « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
```

```
        ofs « "\n";
    }

    return;
}   /* __writeTimeSeries() */
```

#### 4.23.3.9 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

#### 4.23.3.10 computeProductionkW()

```
double Wave::computeProductionkW (
            int timestep,
            double dt_hrs,
            double significant_wave_height_m,
            double energy_period_s )  [virtual]
```

Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.

#### Parameters

| | |
|---|---|
| *timestep* | The timestep (i.e., time series index) for the request. |
| *dt_hrs* | The interval of time [hrs] associated with the timestep. |
| *signficiant_wave_height←↩ _m* | The significant wave height (wave statistic) [m]. |
| *energy_period_s* | The energy period (wave statistic) [s]. |

#### Returns

The production [kW] of the wave turbine.

Reimplemented from Renewable.

```
{
    // check if no resource
    if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
            production_kW = this->__computeParaboloidProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            production_kW = this->__computeGaussianProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wave::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

#### 4.23.3.11 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Renewable.

```
{
    // 1. reset attributes
    //...

    // 2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

### 4.23.4 Member Data Documentation

#### 4.23.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

#### 4.23.4.2 design_significant_wave_height_m

```
double Wave::design_significant_wave_height_m
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

#### 4.23.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

#### 4.23.4.4 power_model_string

```
std::string Wave::power_model_string
```

A string describing the active power production model.

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

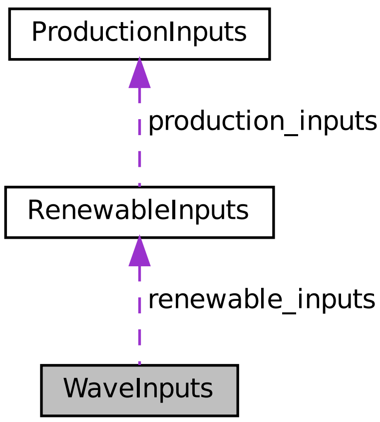
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

## 4.24 WaveInputs Struct Reference

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

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

Collaboration diagram for WaveInputs:




### Public Attributes

- RenewableInputs renewable_inputs

  *An encapsulated RenewableInputs instance.*
- int resource_key = 0

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

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

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

- double design_significant_wave_height_m = 3

  *The significant wave height [m] at which the wave energy converter achieves its rated capacity.*
- double design_energy_period_s = 10

  *The energy period [s] at which the wave energy converter achieves its rated capacity.*
- WavePowerProductionModel power_model = WavePowerProductionModel :: WAVE_POWER_PARABOLOID

  *The wave power production model to be applied.*

## 4.24.1 Detailed Description

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

## 4.24.2 Member Data Documentation

### 4.24.2.1 capital_cost

```
double WaveInputs::capital_cost = -1
```

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

### 4.24.2.2 design_energy_period_s

```
double WaveInputs::design_energy_period_s = 10
```

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

### 4.24.2.3 design_significant_wave_height_m

```
double WaveInputs::design_significant_wave_height_m = 3
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

#### 4.24.2.4 operation_maintenance_cost_kWh

```
double WaveInputs::operation_maintenance_cost_kWh = -1
```

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

#### 4.24.2.5 power_model

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

The wave power production model to be applied.

#### 4.24.2.6 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

#### 4.24.2.7 resource_key

```
int WaveInputs::resource_key = 0
```

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

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

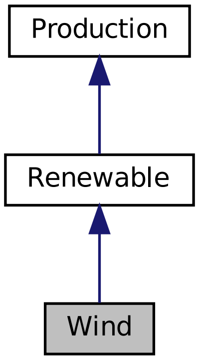
- header/Production/Renewable/Wave.h

## 4.25 Wind Class Reference

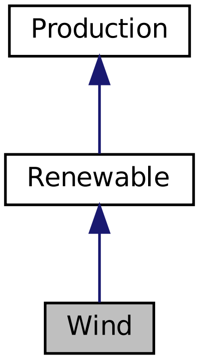
A derived class of the Renewable branch of Production which models wind production.

`#include <Wind.h>`

Inheritance diagram for Wind:




Collaboration diagram for Wind:




### Public Member Functions

- Wind (void)

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

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

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

  *Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.*
- double commit (int, double, double, double)

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

  *Destructor for the Wind class.*

## Public Attributes

- double design_speed_ms

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

  *The wind power production model to be applied.*
- std::string power_model_string

  *A string describing the active power production model.*

## Private Member Functions

- void __checkInputs (WindInputs)

  *Helper method to check inputs to the Wind constructor.*
- double __getGenericCapitalCost (void)

  *Helper method to generate a generic wind turbine capital cost.*
- double __getGenericOpMaintCost (void)

  *Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.*
- double __computeExponentialProductionkW (int, double, double)

  *Helper method to compute wind turbine production under an exponential production model.*
- double __computeLookupProductionkW (int, double, double)

  *Helper method to compute wind turbine production by way of looking up using given power curve data.*
- void __writeSummary (std::string)

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

  *Helper method to write time series results for Wind.*

### 4.25.1 Detailed Description

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

### 4.25.2 Constructor & Destructor Documentation

#### 4.25.2.1 Wind() [1/2]

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

Constructor (dummy) for the Wind class.

```
{
    return;
}   /* Wind() */
```

#### 4.25.2.2 Wind() [2/2]

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

Constructor (intended) for the Wind class.

**Parameters**

| | |
|---|---|
| *n_points* | The number of points in the modelling time series. |
| *n_years* | The number of years being modelled. |
| *wind_inputs* | A structure of Wind constructor inputs. |

```
  :
Renewable(
    n_points,
    n_years,
    wind_inputs.renewable_inputs
)
{
    //  1. check inputs
    this->__checkInputs(wind_inputs);

    //  2. set attributes
    this->type = RenewableType :: WIND;
    this->type_str = "WIND";

    this->resource_key = wind_inputs.resource_key;

    this->design_speed_ms = wind_inputs.design_speed_ms;

    this->power_model = wind_inputs.power_model;

    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            this->power_model_string = "EXPONENTIAL";

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wind():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);
```

```

            break;
        }
    }

    if (wind_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }

    if (wind_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Wind object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

#### 4.25.2.3 ∼Wind()

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

Destructor for the Wind class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Wind object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~Wind() */
```

### 4.25.3 Member Function Documentation

#### 4.25.3.1 __checkInputs()

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

Helper method to check inputs to the Wind constructor.

**Parameters**

| | |
|---|---|
| *wind_inputs* | A structure of Wind constructor inputs. |

```
{
    //  1. check design_speed_ms
    if (wind_inputs.design_speed_ms <= 0) {
        std::string error_str = "ERROR:  Wind():  ";
        error_str += "WindInputs::design_speed_ms must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
```

```
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

#### 4.25.3.2 __computeExponentialProductionkW()

```
double Wind::__computeExponentialProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms )  [private]
```

Helper method to compute wind turbine production under an exponential production model.

Ref: Truelove et al. [2019]

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *wind_resource_ms* | The available wind resource [m/s]. |

**Returns**

The production [kW] of the wind turbine, under an exponential model.

```
{
    double production = 0;

    double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
        this->design_speed_ms;

    if (turbine_speed < -0.76 or turbine_speed > 0.68) {
        production = 0;
    }

    else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
        production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
    }

    else {
        production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
    }

    return production * this->capacity_kW;
}   /* __computeExponentialProductionkW() */
```

#### 4.25.3.3 __computeLookupProductionkW()

```
double Wind::__computeLookupProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms )  [private]
```

Helper method to compute wind turbine production by way of looking up using given power curve data.

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |
| *dt_hrs* | The interval of time [hrs] associated with the action. |
| *wind_resource_ms* | The available wind resource [m/s]. |

**Returns**

The interpolated production [kW] of the wind turbine.

```
{
    // *** WORK IN PROGRESS *** //

    return 0;
}   /* __computeLookupProductionkW() */
```

### 4.25.3.4 __getGenericCapitalCost()

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

Helper method to generate a generic wind turbine capital cost.

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

**Returns**

A generic capital cost for the wind turbine [CAD].

```
{
    double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

### 4.25.3.5 __getGenericOpMaintCost()

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

Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

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

**Returns**

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

```
{
    double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

### 4.25.3.6 __writeSummary()

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

Helper method to write summary results for Wind.

**Parameters**

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW WIND Summary Results\n";
    ofs « "\n--------\n\n";


    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « "kW  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Wind attributes
    ofs « "## Wind Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            //...

            break;
        }

        default: {
            // write nothing!

            break;
        }
    }

    ofs « "\n--------\n\n";

    //  2.4. Wind Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
```

```
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __writeSummary() */
```

#### 4.25.3.7 __writeTimeSeries()

```
void Wind::__writeTimeSeries (
            std::string write_path,
            std::vector< double > * time_vec_hrs_ptr,
            std::map< int, std::vector< double >> * resource_map_1D_ptr,
            std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
            int max_lines = -1 )  [private], [virtual]
```

Helper method to write time series results for Wind.

**Parameters**

| | |
|---|---|
| *write_path* | A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite. |
| *time_vec_hrs_ptr* | A pointer to the time_vec_hrs attribute of the ElectricalLoad. |
| *resource_map_1D_ptr* | A pointer to the 1D map of Resources. |
| *resource_map_2D_ptr* | A pointer to the 2D map of Resources. |
| *max_lines* | The maximum number of lines of output to write. |

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Wind Resource [m/s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    return;
```

```
}    /* __writeTimeSeries() */
```

#### 4.25.3.8 commit()

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

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

**Parameters**

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

**Returns**

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

#### 4.25.3.9 computeProductionkW()

```
double Wind::computeProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms ) [virtual]
```

Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.

**Parameters**

| | |
|---|---|
| *timestep* | The timestep (i.e., time series index) for the request. |
| *dt_hrs* | The interval of time [hrs] associated with the timestep. |
| *wind_resource_ms* | Wind resource (i.e. wind speed) [m/s]. |

**Returns**

The production [kW] of the wind turbine.

Reimplemented from Renewable.

```
{
    // check if no resource
    if (wind_resource_ms <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            production_kW = this->__computeExponentialProductionkW(
                timestep,
                dt_hrs,
                wind_resource_ms
            );

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                wind_resource_ms
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wind::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

#### 4.25.3.10 handleReplacement()

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

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

**Parameters**

| | |
|---|---|
| *timestep* | The current time step of the Model run. |

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
```

```
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

## 4.25.4 Member Data Documentation

### 4.25.4.1 design_speed_ms

`double Wind::design_speed_ms`

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

### 4.25.4.2 power_model

`WindPowerProductionModel Wind::power_model`

The wind power production model to be applied.

### 4.25.4.3 power_model_string

`std::string Wind::power_model_string`

A string describing the active power production model.

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

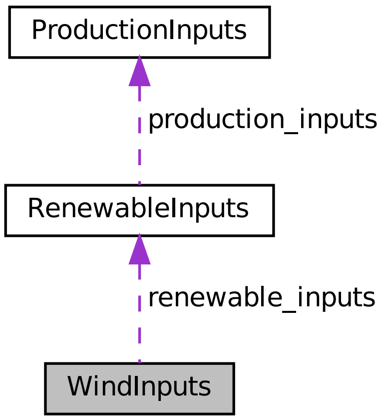
- header/Production/Renewable/Wind.h
- source/Production/Renewable/Wind.cpp

## 4.26 WindInputs Struct Reference

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

```
#include <Wind.h>
```

Collaboration diagram for WindInputs:




### Public Attributes

- RenewableInputs renewable_inputs

    *An encapsulated RenewableInputs instance.*
- int resource_key = 0

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

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

    *The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].*
- double design_speed_ms = 8

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

    *The wind power production model to be applied.*

### 4.26.1 Detailed Description

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

## 4.26.2 Member Data Documentation

### 4.26.2.1 capital_cost

```
double WindInputs::capital_cost = -1
```

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

### 4.26.2.2 design_speed_ms

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

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

### 4.26.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

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

### 4.26.2.4 power_model

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

The wind power production model to be applied.

### 4.26.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

### 4.26.2.6 resource_key

```
int WindInputs::resource_key = 0
```

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

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

- header/Production/Renewable/Wind.h

# Chapter 5

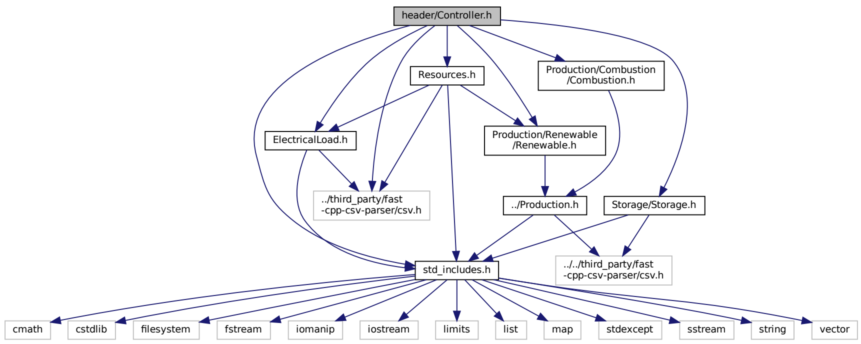
# File Documentation

## 5.1 header/Controller.h File Reference

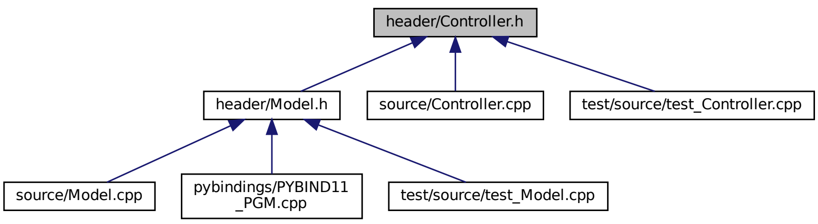
Header file the Controller class.

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

Include dependency graph for Controller.h:




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



## Classes

- class Controller

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

## Enumerations

- enum ControlMode { LOAD_FOLLOWING , CYCLE_CHARGING , N_CONTROL_MODES }

    *An enumeration of the types of control modes supported by PGMcpp.*

### 5.1.1 Detailed Description

Header file the Controller class.

### 5.1.2 Enumeration Type Documentation

#### 5.1.2.1 ControlMode

```
enum ControlMode
```

An enumeration of the types of control modes supported by PGMcpp.

**Enumerator**

| | |
|---|---|
| LOAD_FOLLOWING | Load following control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets. |
| CYCLE_CHARGING | Cycle charging control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets. |
| N_CONTROL_MODES | A simple hack to get the number of elements in ControlMode. |

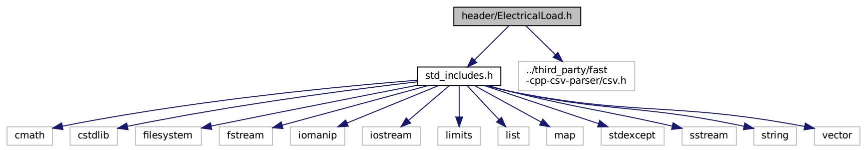
```
                  {
    LOAD_FOLLOWING,
    CYCLE_CHARGING,
    N_CONTROL_MODES
};
```

## 5.2 header/ElectricalLoad.h File Reference

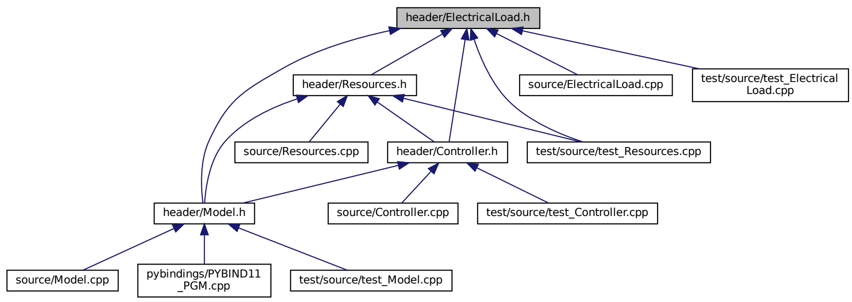
Header file the ElectricalLoad class.

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

Include dependency graph for ElectricalLoad.h:




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




## Classes

- class ElectricalLoad

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

### 5.2.1 Detailed Description

Header file the ElectricalLoad class.

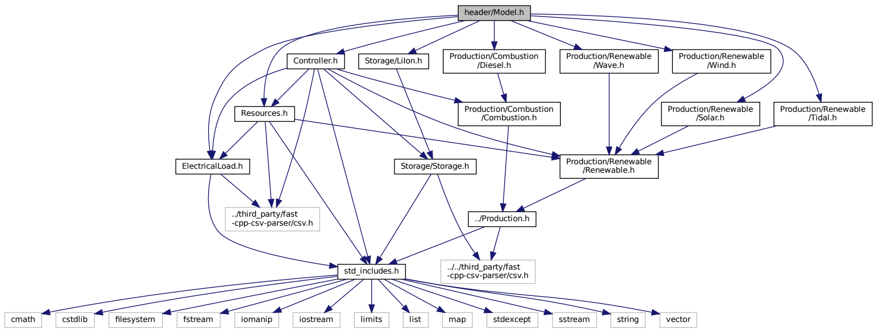
## 5.3 header/Model.h File Reference

Header file the Model class.

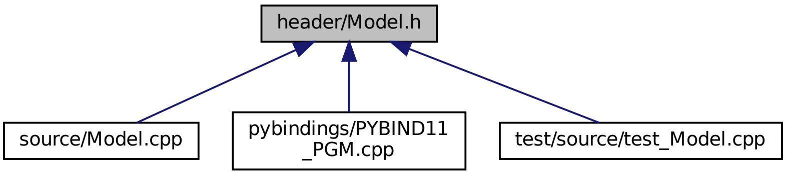
```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
#include "Production/Renewable/Wave.h"
#include "Production/Renewable/Wind.h"
```

```
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:




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




## Classes

- struct ModelInputs

    *A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided).*

- class Model

    *A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.*

### 5.3.1 Detailed Description

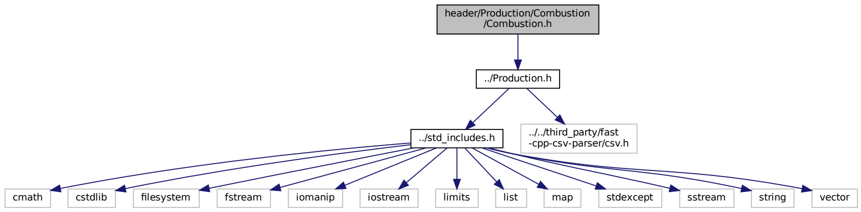
Header file the Model class.

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

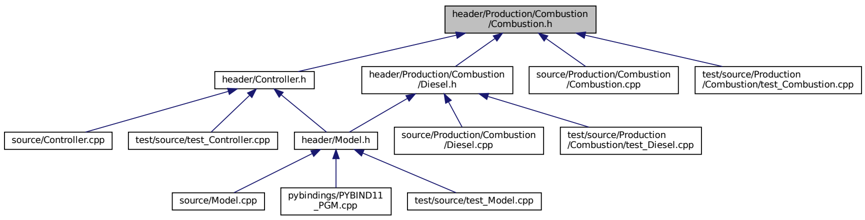
Header file the Combustion class.

```
#include "../Production.h"
```

Include dependency graph for Combustion.h:




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




## Classes

- struct CombustionInputs

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

  *A structure which bundles the emitted masses of various emissions chemistries.*
- class Combustion

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

## Enumerations

- enum CombustionType { DIESEL , N_COMBUSTION_TYPES }

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

## 5.4.1 Detailed Description

Header file the Combustion class.

## 5.4.2 Enumeration Type Documentation

### 5.4.2.1 CombustionType

```
enum CombustionType
```

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

**Enumerator**

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

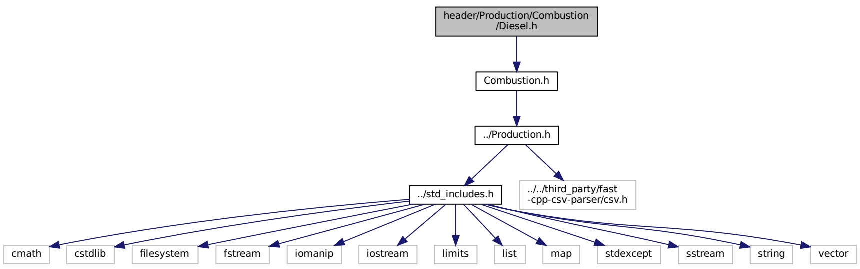
```
            {
    DIESEL,
    N_COMBUSTION_TYPES
};
```

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

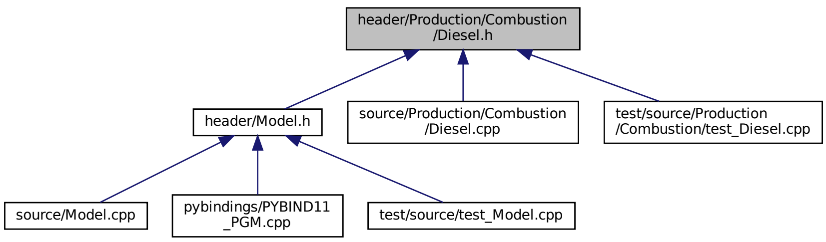
Header file the Diesel class.

`#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

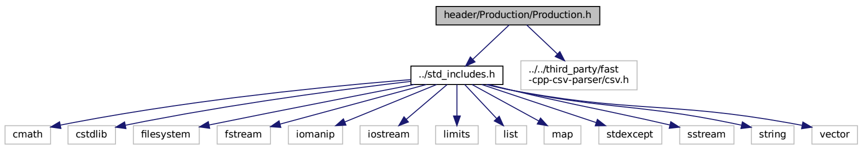
Header file the Diesel class.

## 5.6 header/Production/Production.h File Reference

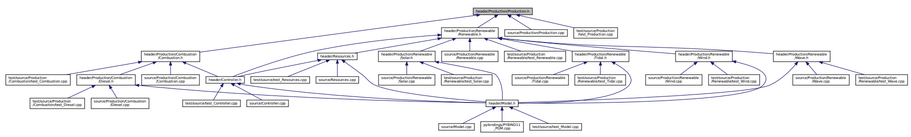
Header file the Production class.

```
#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

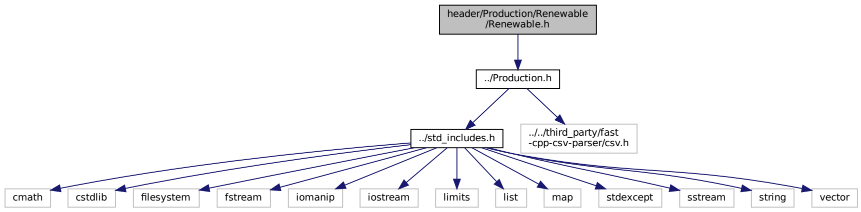
Header file the Production class.

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

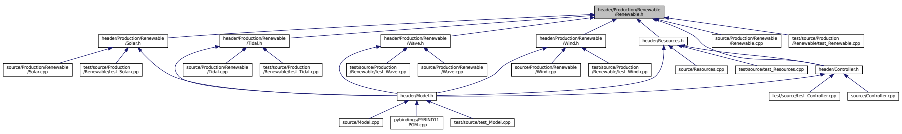
Header file the Renewable class.

`#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**

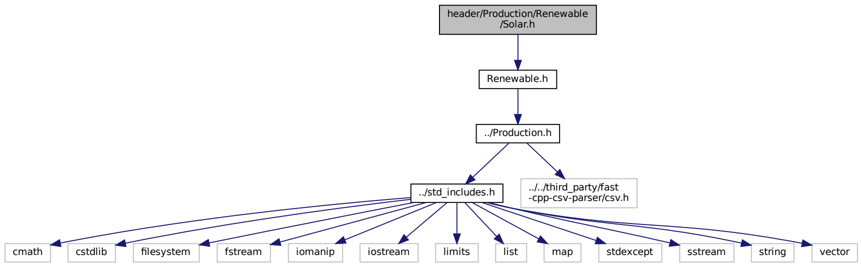
| | |
|---|---|
| SOLAR | A solar photovoltaic (PV) array. |
| TIDAL | A tidal stream turbine (or tidal energy converter, TEC) |
| WAVE | A wave energy converter (WEC) |
| WIND | A wind turbine. |
| N_RENEWABLE_TYPES | A simple hack to get the number of elements in RenewableType. |

```
               {
    SOLAR,
    TIDAL,
    WAVE,
    WIND,
    N_RENEWABLE_TYPES
};
```

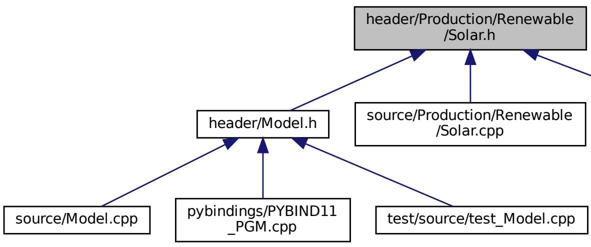
## 5.8 header/Production/Renewable/Solar.h File Reference

Header file the Solar class.

`#include "Renewable.h"`
Include dependency graph for Solar.h:




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




## Classes

- struct SolarInputs

  *A structure which bundles the necessary inputs for the Solar constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.*
- class Solar

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

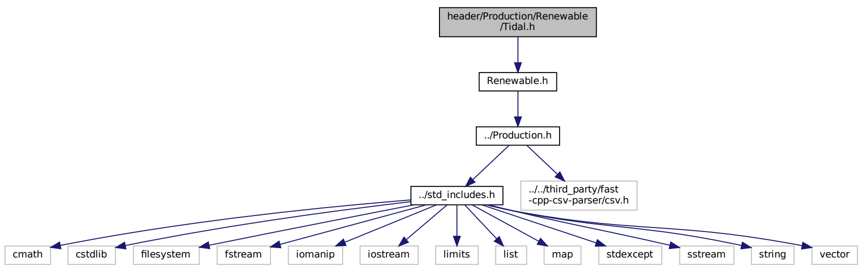
### 5.8.1 Detailed Description

Header file the Solar class.

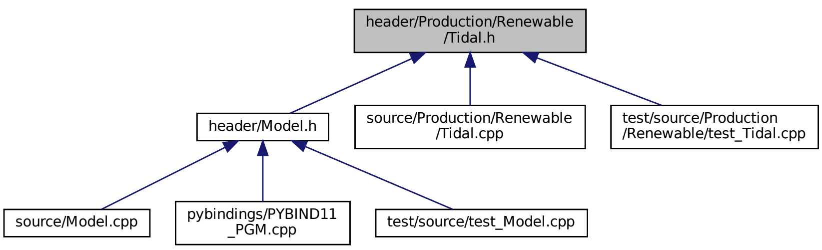
## 5.9 header/Production/Renewable/Tidal.h File Reference

Header file the Tidal class.

```
#include "Renewable.h"
```
Include dependency graph for Tidal.h:




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




### Classes

- struct TidalInputs

  *A structure which bundles the necessary inputs for the Tidal constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.*
- class Tidal

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

### Enumerations

- enum TidalPowerProductionModel { TIDAL_POWER_CUBIC , TIDAL_POWER_EXPONENTIAL , TIDAL_POWER_LOOKUP , N_TIDAL_POWER_PRODUCTION_MODELS }

### 5.9.1 Detailed Description

Header file the Tidal class.

### 5.9.2 Enumeration Type Documentation

#### 5.9.2.1 TidalPowerProductionModel

```
enum TidalPowerProductionModel
```

**Enumerator**

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

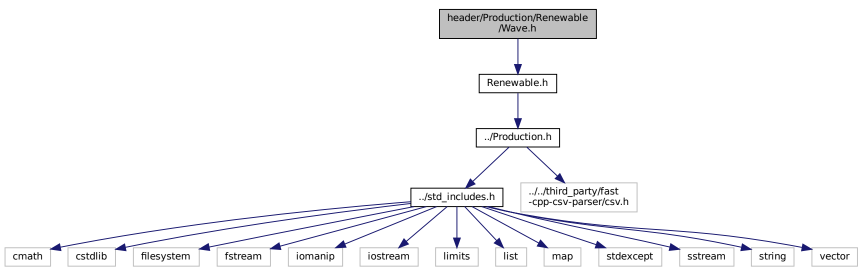
```
                          {
    TIDAL_POWER_CUBIC,
    TIDAL_POWER_EXPONENTIAL,
    TIDAL_POWER_LOOKUP,
    N_TIDAL_POWER_PRODUCTION_MODELS
};
```

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

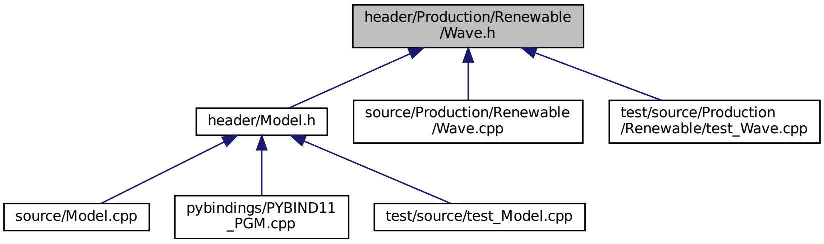
Header file the Wave class.

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

Include dependency graph for Wave.h:



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




## Classes

- struct WaveInputs

  *A structure which bundles the necessary inputs for the Wave constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.*
- class Wave

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

## Enumerations

- enum WavePowerProductionModel { WAVE_POWER_GAUSSIAN , WAVE_POWER_PARABOLOID , WAVE_POWER_LOOKUP , N_WAVE_POWER_PRODUCTION_MODELS }

## 5.10.1 Detailed Description

Header file the Wave class.

## 5.10.2 Enumeration Type Documentation

### 5.10.2.1 WavePowerProductionModel

```
enum WavePowerProductionModel
```

**Enumerator**

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

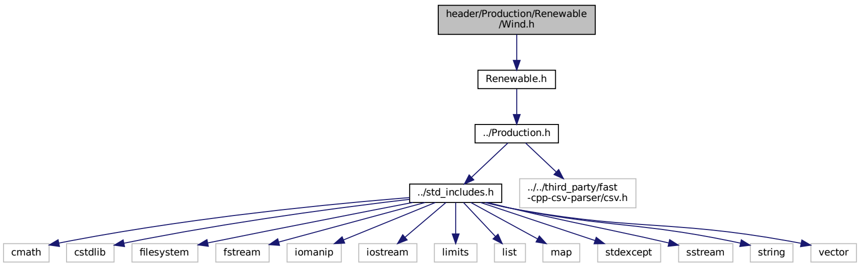
```
                        {
    WAVE_POWER_GAUSSIAN,
    WAVE_POWER_PARABOLOID,
    WAVE_POWER_LOOKUP,
    N_WAVE_POWER_PRODUCTION_MODELS
};
```

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

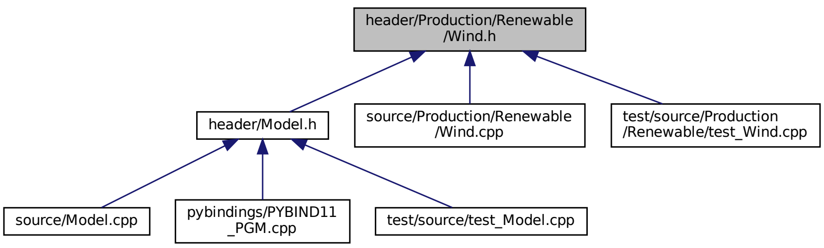
Header file the Wind class.

`#include "Renewable.h"`

Include dependency graph for Wind.h:




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




## Classes

- struct WindInputs

    *A structure which bundles the necessary inputs for the Wind constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs.*

- class Wind

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

## Enumerations

- enum WindPowerProductionModel { WIND_POWER_EXPONENTIAL , WIND_POWER_LOOKUP , N_WIND_POWER_PRODUCTION_MODELS }

### 5.11.1 Detailed Description

Header file the Wind class.

### 5.11.2 Enumeration Type Documentation

#### 5.11.2.1 WindPowerProductionModel

`enum WindPowerProductionModel`

**Enumerator**

| | |
|---|---|
| WIND_POWER_EXPONENTIAL | An exponential power production model. |
| WIND_POWER_LOOKUP | Lookup from a given set of power curve data. |
| N_WIND_POWER_PRODUCTION_MODELS | A simple hack to get the number of elements in WindPowerProductionModel. |

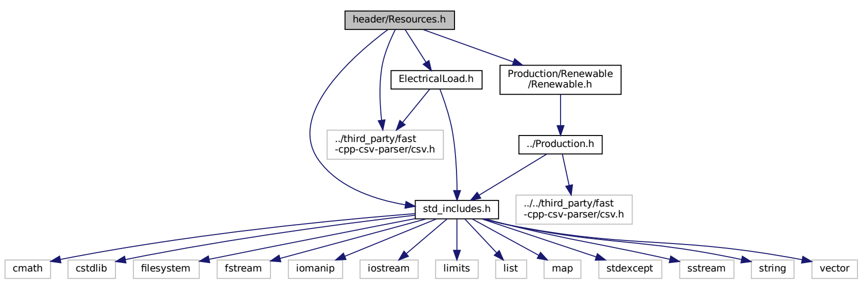
```
                         {
    WIND_POWER_EXPONENTIAL,
    WIND_POWER_LOOKUP,
    N_WIND_POWER_PRODUCTION_MODELS
};
```

## 5.12 header/Resources.h File Reference

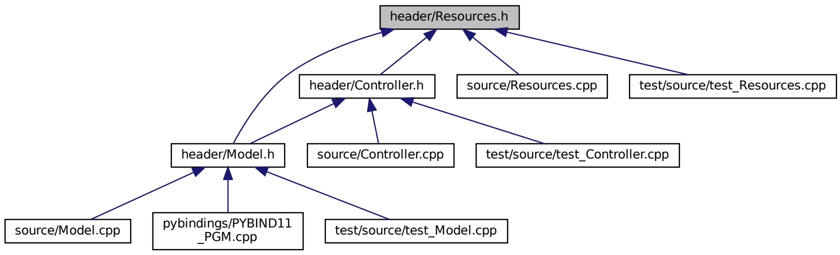
Header file the Resources class.

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

Include dependency graph for Resources.h:



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




## Classes

- class Resources

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

### 5.12.1 Detailed Description

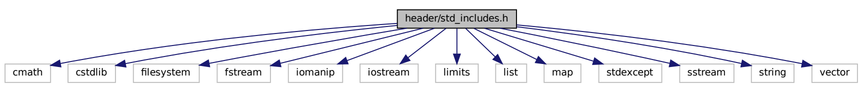
Header file the Resources class.

## 5.13 header/std_includes.h File Reference

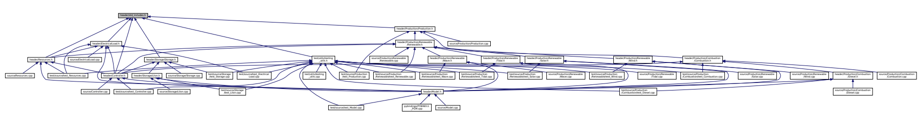
Header file which simply batches together the usual, standard includes.

```
#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <stdexcept>
#include <sstream>
#include <string>
#include <vector>
```

Include dependency graph for std_includes.h:




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



### 5.13.1 Detailed Description

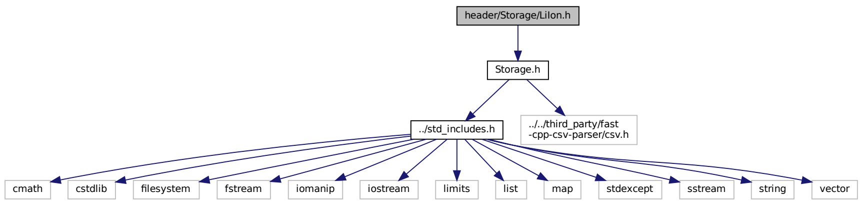
Header file which simply batches together the usual, standard includes.

## 5.14 header/Storage/LiIon.h File Reference

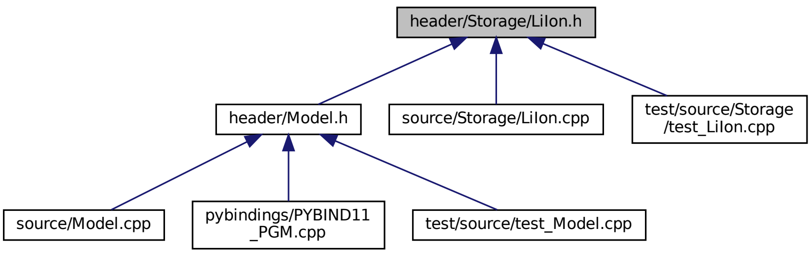
Header file the LiIon class.

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

Include dependency graph for LiIon.h:




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




### Classes

- struct LiIonInputs

  *A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.*

- class LiIon

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

### 5.14.1 Detailed Description

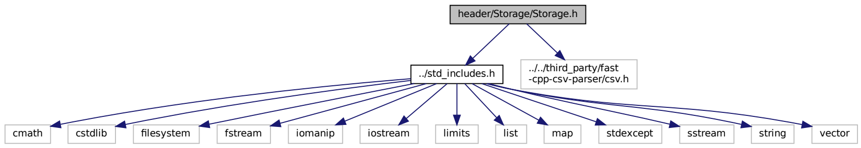
Header file the LiIon class.

## 5.15 header/Storage/Storage.h File Reference

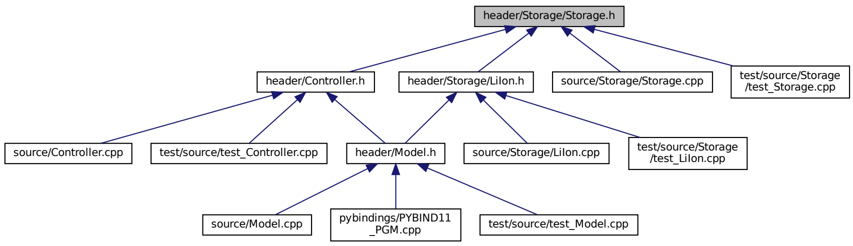
Header file the Storage class.

```
#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

- struct StorageInputs

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

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

### Enumerations

- enum StorageType { LIION , N_STORAGE_TYPES }

  *An enumeration of the types of Storage asset supported by PGMcpp.*

### 5.15.1 Detailed Description

Header file the Storage class.

### 5.15.2 Enumeration Type Documentation

#### 5.15.2.1 StorageType

```
enum StorageType
```

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

**Enumerator**

| | |
|---|---|
| LIION | A system of lithium ion batteries. |
| N_STORAGE_TYPES | A simple hack to get the number of elements in StorageType. |

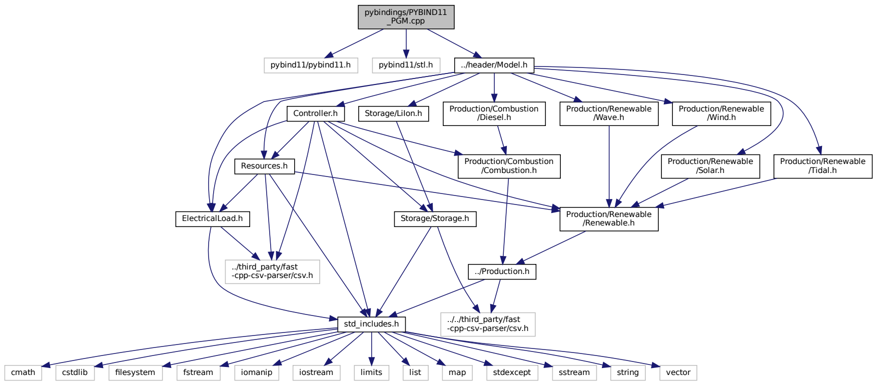
```
           {
    LIION,
    N_STORAGE_TYPES
};
```

## 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()

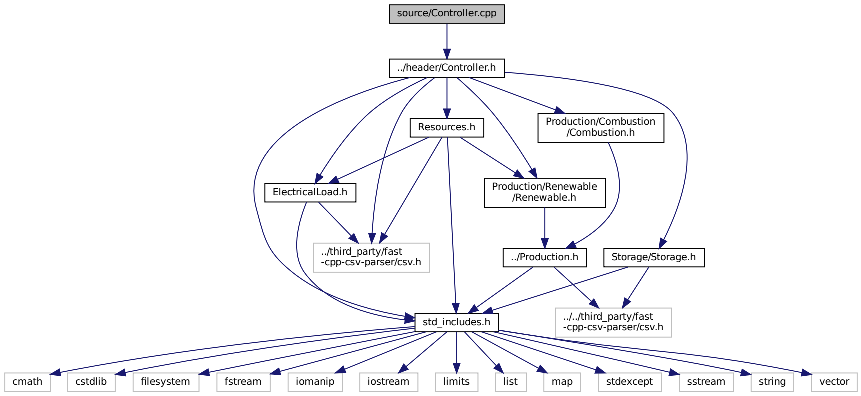
```
PYBIND11_MODULE (
        PGMcpp ,
        m )
{

// ================ Controller ================ //
/*
pybind11::class_<Controller>(m, "Controller")
    .def(pybind11::init());
*/
// ================ END Controller ================ //



// ================ ElectricalLoad ================ //
/*
pybind11::class_<ElectricalLoad>(m, "ElectricalLoad")
    .def_readwrite("n_points", &ElectricalLoad::n_points)
    .def_readwrite("max_load_kW", &ElectricalLoad::max_load_kW)
    .def_readwrite("mean_load_kW", &ElectricalLoad::mean_load_kW)
    .def_readwrite("min_load_kW", &ElectricalLoad::min_load_kW)
    .def_readwrite("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs)
    .def_readwrite("load_vec_kW", &ElectricalLoad::load_vec_kW)
    .def_readwrite("time_vec_hrs", &ElectricalLoad::time_vec_hrs)

    .def(pybind11::init<std::string>());
*/
// ================ END ElectricalLoad ================ //



// ================ Model ================ //
/*
pybind11::class_<Model>(m, "Model")
    .def(
        pybind11::init<
            ElectricalLoad*,
            RenewableResources*
        >()
    );
*/
// ================ END Model ================ //



// ================ RenewableResources ================ //
/*
pybind11::class_<RenewableResources>(m, "RenewableResources")
    .def(pybind11::init());
    /*
    .def(pybind11::init<>());
    */
*/
// ================ END RenewableResources ================ //

}   /* 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.

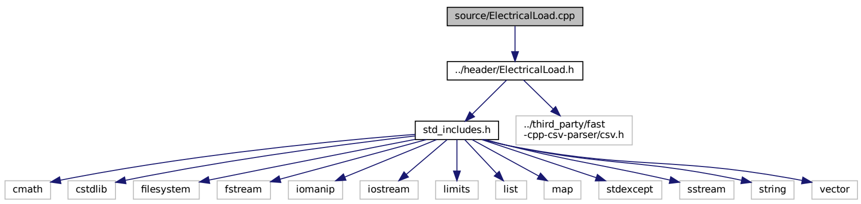
A class which contains a various dispatch control logic. Intended to serve as a component class of Controller.

## 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

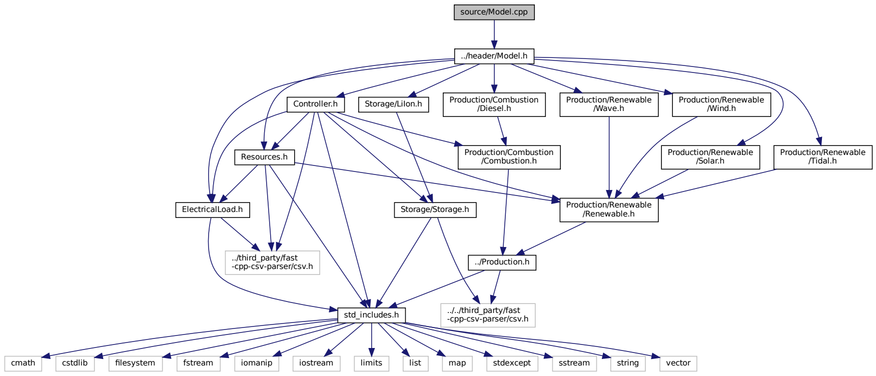
Implementation file for the ElectricalLoad class.

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

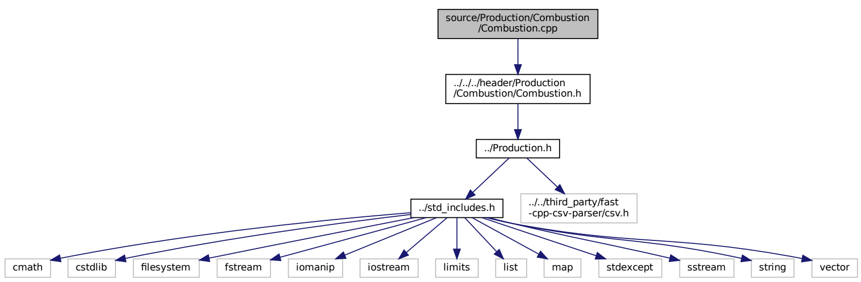
Implementation file for the Model class.

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

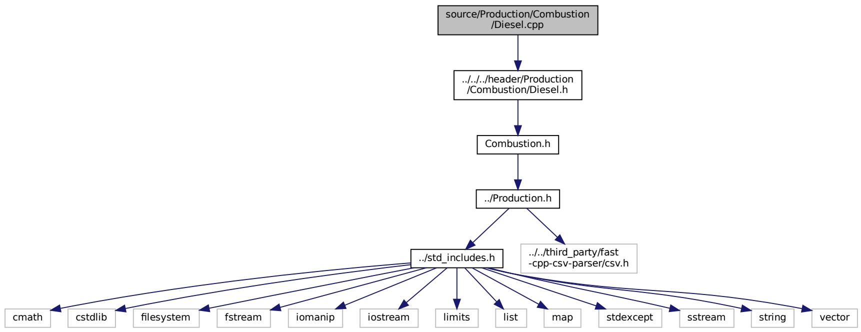
Implementation file for the Combustion class.

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

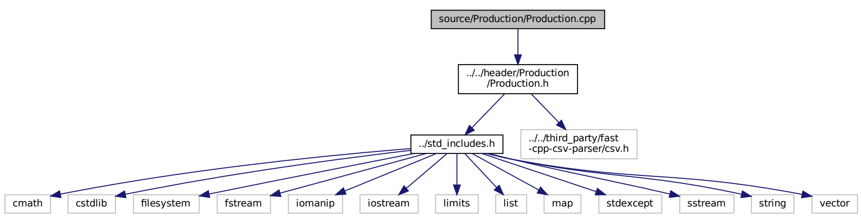
Implementation file for the Diesel class.

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.

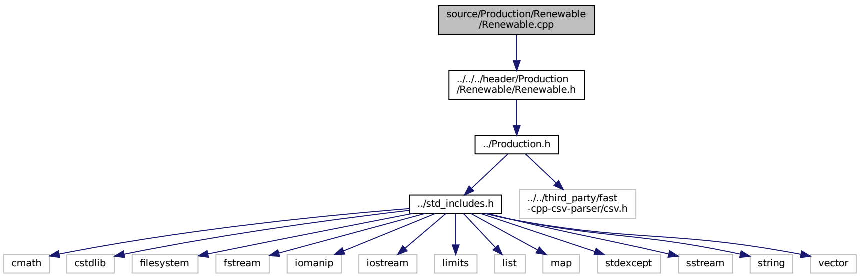
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

## 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.

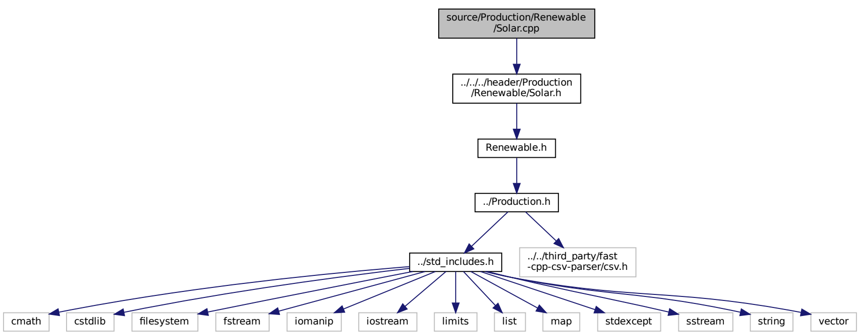
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

## 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.

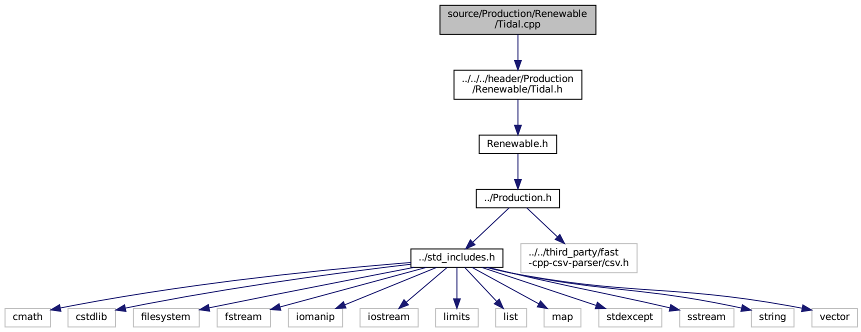
A derived class of the Renewable branch of Production which models solar production.

## 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.

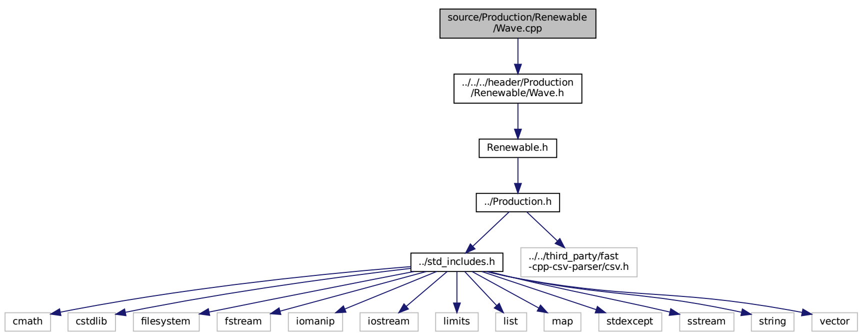
A derived class of the Renewable branch of Production which models tidal production.

## 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.

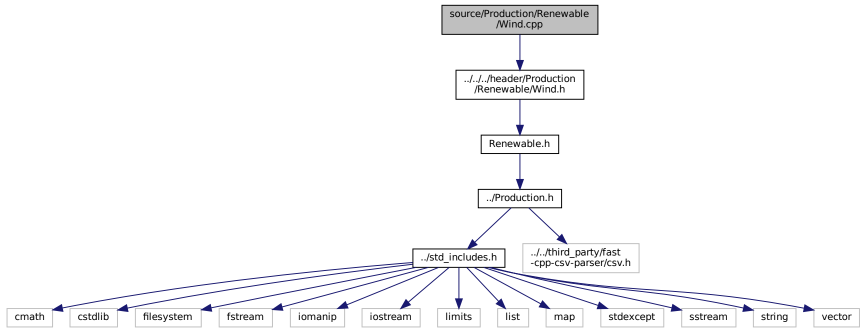
A derived class of the Renewable branch of Production which models wave production.

## 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.

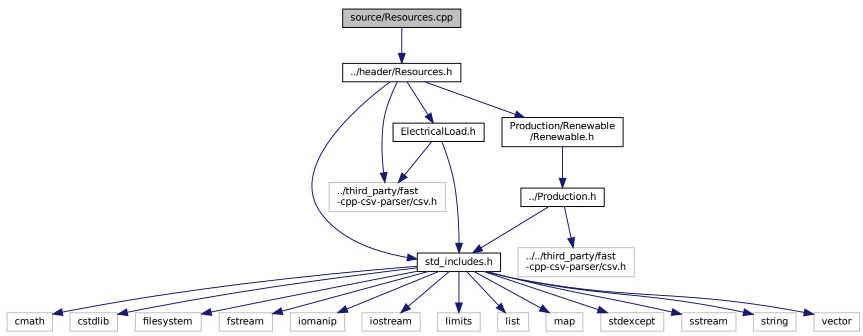
A derived class of the Renewable branch of Production which models wind production.

## 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

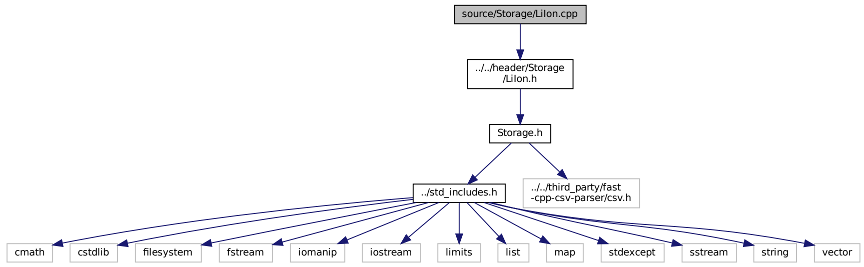
Implementation file for the Resources class.

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

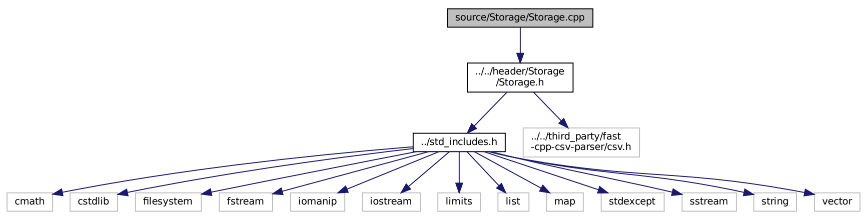
Implementation file for the LiIon class.

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.

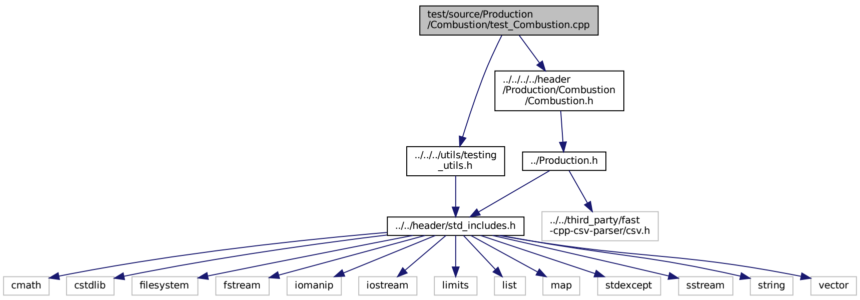
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

## 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, 1, combustion_inputs);

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



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

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

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

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

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

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

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

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

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

testFloatEquals(
```

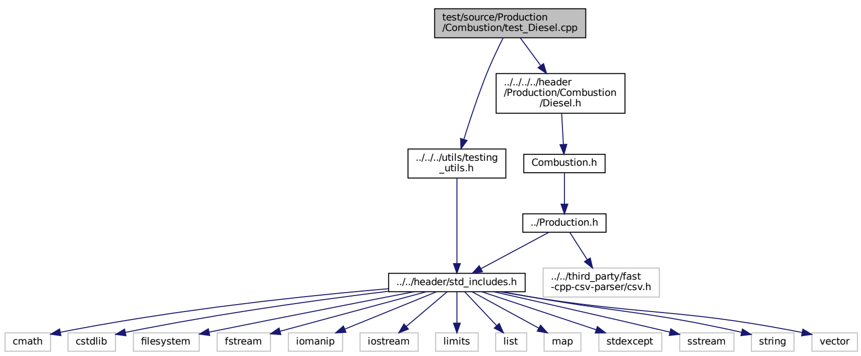
```
    test_combustion.PM_emissions_vec_kg.size(),
    8760,
    __FILE__,
    __LINE__
);

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

}   /* try */


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

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


printGold(" .............. ");
printGreen("PASS");
std::cout « std::endl;
return 0;

}   /* main() */
```

## 5.32 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 )
```

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

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

    srand(time(NULL));


    Combustion* test_diesel_ptr;

try {

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

bool error_flag = true;

try {
    DieselInputs bad_diesel_inputs;
    bad_diesel_inputs.fuel_cost_L = -1;

    Diesel bad_diesel(8760, 1, bad_diesel_inputs);

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

DieselInputs diesel_inputs;

test_diesel_ptr =  new Diesel(8760, 1, diesel_inputs);


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



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

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

testFloatEquals(
    test_diesel_ptr->type,
    CombustionType :: DIESEL,
    __FILE__,
    __LINE__
);

testTruth(
    test_diesel_ptr->type_str == "DIESEL",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_diesel_ptr->linear_fuel_slope_LkWh,
    0.265675,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_diesel_ptr->linear_fuel_intercept_LkWh,
```

```
    0.026676,
    __FILE__,
     __LINE__
);

testFloatEquals(
    test_diesel_ptr->capital_cost,
    94125.375446,
    __FILE__,
    __LINE__
);

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

testFloatEquals(
    ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
    0.2,
    __FILE__,
    __LINE__
);

testFloatEquals(
    ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
    4,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_diesel_ptr->replace_running_hrs,
    30000,
    __FILE__,
    __LINE__
);

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



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

//  test capacity constraint
testFloatEquals(
    test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
    test_diesel_ptr->capacity_kW,
    __FILE__,
    __LINE__
);

//  test minimum load ratio constraint
testFloatEquals(
    test_diesel_ptr->requestProductionkW(
        0,
        1,
        0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
            test_diesel_ptr->capacity_kW
    ),
    ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
    __FILE__,
    __LINE__
);

//  test commit()
std::vector<double> dt_vec_hrs (48, 1);

std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
};

std::vector<bool> expected_is_running_vec = {
    1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
};

double load_kW = 0;
double production_kW = 0;
double roll = 0;
```

```
for (int i = 0; i < 48; i++) {
    roll = (double)rand() / RAND_MAX;

    if (roll >= 0.95) {
        roll = 1.25;
    }

    load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
    load_kW = load_vec_kW[i];

    production_kW = test_diesel_ptr->requestProductionkW(
        i,
        dt_vec_hrs[i],
        load_kW
    );

    load_kW = test_diesel_ptr->commit(
        i,
        dt_vec_hrs[i],
        production_kW,
        load_kW
    );

    // load_kW <= load_vec_kW (i.e., after vs before)
    testLessThanOrEqualTo(
        load_kW,
        load_vec_kW[i],
        __FILE__,
        __LINE__
    );

    // production = dispatch + storage + curtailment
    testFloatEquals(
        test_diesel_ptr->production_vec_kW[i] -
        test_diesel_ptr->dispatch_vec_kW[i] -
        test_diesel_ptr->storage_vec_kW[i] -
        test_diesel_ptr->curtailment_vec_kW[i],
        0,
        __FILE__,
        __LINE__
    );

    // capacity constraint
    if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
        testFloatEquals(
            test_diesel_ptr->production_vec_kW[i],
            test_diesel_ptr->capacity_kW,
            __FILE__,
            __LINE__
        );
    }

    // minimum load ratio constraint
    else if (
        test_diesel_ptr->is_running and
        test_diesel_ptr->production_vec_kW[i] > 0 and
        load_vec_kW[i] <
        ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
    ) {
        testFloatEquals(
            test_diesel_ptr->production_vec_kW[i],
            ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
                test_diesel_ptr->capacity_kW,
            __FILE__,
            __LINE__
        );
    }

    // minimum runtime constraint
    testFloatEquals(
        test_diesel_ptr->is_running_vec[i],
        expected_is_running_vec[i],
        __FILE__,
        __LINE__
    );

    // O&M, fuel consumption, and emissions > 0 whenever diesel is running
    if (test_diesel_ptr->is_running) {
        testGreaterThan(
            test_diesel_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
```

```
            test_diesel_ptr->fuel_consumption_vec_L[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->fuel_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->CO2_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->CO_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->NOx_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->SOx_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->CH4_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_diesel_ptr->PM_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
    else {
        testFloatEquals(
            test_diesel_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->fuel_consumption_vec_L[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->fuel_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->CO2_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
```

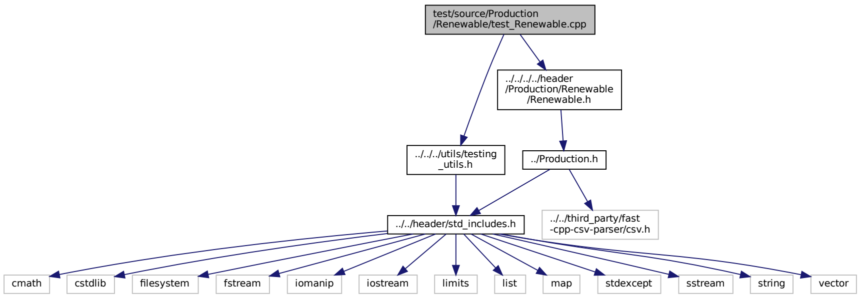
```
            test_diesel_ptr->CO_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->NOx_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->SOx_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->CH4_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_diesel_ptr->PM_emissions_vec_kg[i],
            0,
            __FILE__,
            __LINE__
        );
    }
}

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

}   /* try */


catch (...) {
    delete test_diesel_ptr;

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


delete test_diesel_ptr;

printGold(" ... ");
printGreen("PASS");
std::cout « std::endl;
return 0;

}   /* main() */
```

## 5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference

Testing suite for Renewable class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Renewable.h"
```

Include dependency graph for test_Renewable.cpp:




## Functions

- int main (int argc, char ∗∗argv)

## 5.33.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

## 5.33.2 Function Documentation

### 5.33.2.1 main()

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

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

    srand(time(NULL));


try {

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

RenewableInputs renewable_inputs;

Renewable test_renewable(8760, 1, renewable_inputs);

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



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

testTruth(
```

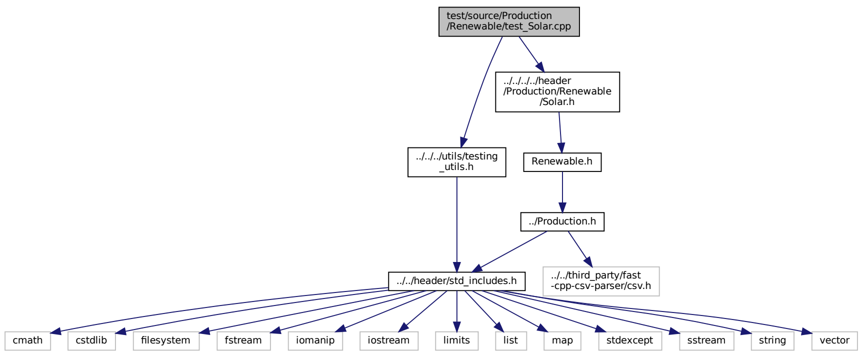
```
    not renewable_inputs.production_inputs.print_flag,
    __FILE__,
    __LINE__
);

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

}   /* try */


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

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


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

## 5.34 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for Solar class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Solar.h"
```

Include dependency graph for test_Solar.cpp:




### Functions

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

### 5.34.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar class.

### 5.34.2 Function Documentation

#### 5.34.2.1 main()

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

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

    srand(time(NULL));

    Renewable* test_solar_ptr;

try {

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

bool error_flag = true;

try {
    SolarInputs bad_solar_inputs;
    bad_solar_inputs.derating = -1;

    Solar bad_solar(8760, 1, bad_solar_inputs);

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

SolarInputs solar_inputs;

test_solar_ptr = new Solar(8760, 1, solar_inputs);

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



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

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

testFloatEquals(
    test_solar_ptr->type,
    RenewableType :: SOLAR,
    __FILE__,
    __LINE__
);

testTruth(
    test_solar_ptr->type_str == "SOLAR",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_solar_ptr->capital_cost,
    350118.723363,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_solar_ptr->operation_maintenance_cost_kWh,
    0.01,
    __FILE__,
```

```
    __LINE__
);

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



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

// test production constraints
testFloatEquals(
    test_solar_ptr->computeProductionkW(0, 1, 2),
    100,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_solar_ptr->computeProductionkW(0, 1, -1),
    0,
    __FILE__,
    __LINE__
);

// test commit()
std::vector<double> dt_vec_hrs (48, 1);

std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
};

double load_kW = 0;
double production_kW = 0;
double roll = 0;
double solar_resource_kWm2 = 0;

for (int i = 0; i < 48; i++) {
    roll = (double)rand() / RAND_MAX;

    solar_resource_kWm2 = roll;

    roll = (double)rand() / RAND_MAX;

    if (roll <= 0.1) {
        solar_resource_kWm2 = 0;
    }

    else if (roll >= 0.95) {
        solar_resource_kWm2 = 1.25;
    }

    roll = (double)rand() / RAND_MAX;

    if (roll >= 0.95) {
        roll = 1.25;
    }

    load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
    load_kW = load_vec_kW[i];

    production_kW = test_solar_ptr->computeProductionkW(
        i,
        dt_vec_hrs[i],
        solar_resource_kWm2
    );

    load_kW = test_solar_ptr->commit(
        i,
        dt_vec_hrs[i],
        production_kW,
        load_kW
    );

    // is running (or not) as expected
    if (solar_resource_kWm2 > 0) {
        testTruth(
            test_solar_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    else {
        testTruth(
```

```
                not test_solar_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_solar_ptr->production_vec_kW[i] -
            test_solar_ptr->dispatch_vec_kW[i] -
            test_solar_ptr->storage_vec_kW[i] -
            test_solar_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );

        // capacity constraint
        if (solar_resource_kWm2 > 1) {
            testFloatEquals(
                test_solar_ptr->production_vec_kW[i],
                test_solar_ptr->capacity_kW,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M > 0 whenever solar is running (i.e., producing)
        if (test_solar_ptr->is_running) {
            testGreaterThan(
                solar_resource_kWm2,
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_solar_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M = 0 whenever solar is not running (i.e., not producing)
        else {
            testFloatEquals(
                solar_resource_kWm2,
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_solar_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }


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

}   /* try */


catch (...) {
    delete test_solar_ptr;

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


delete test_solar_ptr;
```

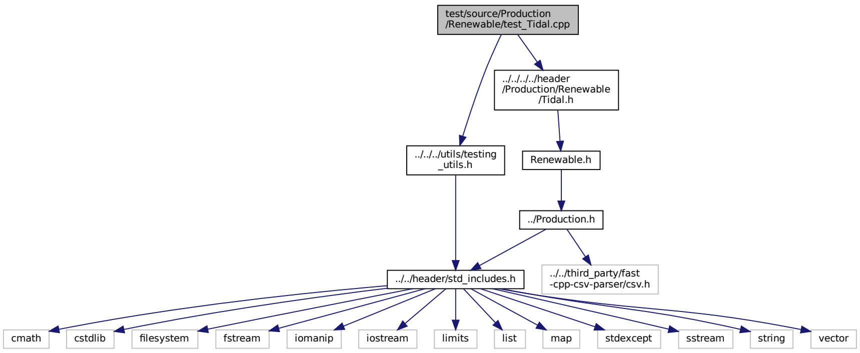
```
printGold(" ..... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

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

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

    srand(time(NULL));

    Renewable* test_tidal_ptr;

try {

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

bool error_flag = true;

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

    Tidal bad_tidal(8760, 1, bad_tidal_inputs);

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

TidalInputs tidal_inputs;

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

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



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

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

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

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

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

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

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



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

```

```
// test production constraints
testFloatEquals(
    test_tidal_ptr->computeProductionkW(0, 1, 1e6),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_tidal_ptr->computeProductionkW(
        0,
        1,
        ((Tidal*)test_tidal_ptr)->design_speed_ms
    ),
    test_tidal_ptr->capacity_kW,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_tidal_ptr->computeProductionkW(0, 1, -1),
    0,
    __FILE__,
    __LINE__
);

// test commit()
std::vector<double> dt_vec_hrs (48, 1);

std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
};

double load_kW = 0;
double production_kW = 0;
double roll = 0;
double tidal_resource_ms = 0;

for (int i = 0; i < 48; i++) {
    roll = (double)rand() / RAND_MAX;

    tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;

    roll = (double)rand() / RAND_MAX;

    if (roll <= 0.1) {
        tidal_resource_ms = 0;
    }

    else if (roll >= 0.95) {
        tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
    }

    roll = (double)rand() / RAND_MAX;

    if (roll >= 0.95) {
        roll = 1.25;
    }

    load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
    load_kW = load_vec_kW[i];

    production_kW = test_tidal_ptr->computeProductionkW(
        i,
        dt_vec_hrs[i],
        tidal_resource_ms
    );

    load_kW = test_tidal_ptr->commit(
        i,
        dt_vec_hrs[i],
        production_kW,
        load_kW
    );

    // is running (or not) as expected
    if (production_kW > 0) {
        testTruth(
            test_tidal_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

```

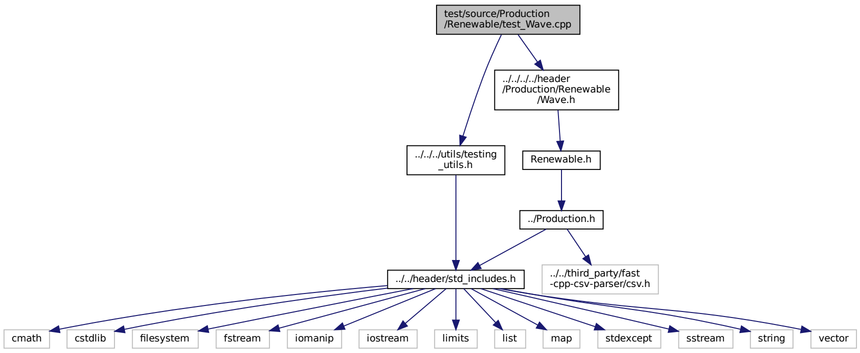
```
    else {
        testTruth(
            not test_tidal_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    // load_kW <= load_vec_kW (i.e., after vs before)
    testLessThanOrEqualTo(
        load_kW,
        load_vec_kW[i],
        __FILE__,
        __LINE__
    );

    // production = dispatch + storage + curtailment
    testFloatEquals(
        test_tidal_ptr->production_vec_kW[i] -
        test_tidal_ptr->dispatch_vec_kW[i] -
        test_tidal_ptr->storage_vec_kW[i] -
        test_tidal_ptr->curtailment_vec_kW[i],
        0,
        __FILE__,
        __LINE__
    );

    // resource, O&M > 0 whenever tidal is running (i.e., producing)
    if (test_tidal_ptr->is_running) {
        testGreaterThan(
            tidal_resource_ms,
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_tidal_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    // O&M = 0 whenever tidal is not running (i.e., not producing)
    else {
        testFloatEquals(
            test_tidal_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }
}


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

}   /* try */


catch (...) {
    delete test_tidal_ptr;

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


delete test_tidal_ptr;

printGold(" ..... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* 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 )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

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

    srand(time(NULL));

    Renewable* test_wave_ptr;

try {

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

bool error_flag = true;

try {
    WaveInputs bad_wave_inputs;
    bad_wave_inputs.design_significant_wave_height_m = -1;
```

```

    Wave bad_wave(8760, 1, bad_wave_inputs);

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

WaveInputs wave_inputs;

test_wave_ptr = new Wave(8760, 1, wave_inputs);

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



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

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

testFloatEquals(
    test_wave_ptr->type,
    RenewableType :: WAVE,
    __FILE__,
    __LINE__
);

testTruth(
    test_wave_ptr->type_str == "WAVE",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wave_ptr->capital_cost,
    850831.063539,
    __FILE__,
    __LINE__
);

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

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



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

// test production constraints
testFloatEquals(
    test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
    0,
    __FILE__,
    __LINE__
);

// test commit()
std::vector<double> dt_vec_hrs (48, 1);

std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
};

double load_kW = 0;
double production_kW = 0;
```

```
double roll = 0;
double significant_wave_height_m = 0;
double energy_period_s = 0;

for (int i = 0; i < 48; i++) {
    roll = (double)rand() / RAND_MAX;

    if (roll <= 0.05) {
        roll = 0;
    }

    significant_wave_height_m = roll *
        ((Wave*)test_wave_ptr)->design_significant_wave_height_m;

    roll = (double)rand() / RAND_MAX;

    if (roll <= 0.05) {
        roll = 0;
    }

    energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;

    roll = (double)rand() / RAND_MAX;

    if (roll >= 0.95) {
        roll = 1.25;
    }

    load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
    load_kW = load_vec_kW[i];

    production_kW = test_wave_ptr->computeProductionkW(
        i,
        dt_vec_hrs[i],
        significant_wave_height_m,
        energy_period_s
    );

    load_kW = test_wave_ptr->commit(
        i,
        dt_vec_hrs[i],
        production_kW,
        load_kW
    );

    // is running (or not) as expected
    if (production_kW > 0) {
        testTruth(
            test_wave_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    else {
        testTruth(
            not test_wave_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    // load_kW <= load_vec_kW (i.e., after vs before)
    testLessThanOrEqualTo(
        load_kW,
        load_vec_kW[i],
        __FILE__,
        __LINE__
    );

    // production = dispatch + storage + curtailment
    testFloatEquals(
        test_wave_ptr->production_vec_kW[i] -
        test_wave_ptr->dispatch_vec_kW[i] -
        test_wave_ptr->storage_vec_kW[i] -
        test_wave_ptr->curtailment_vec_kW[i],
        0,
        __FILE__,
        __LINE__
    );

    // resource, O&M > 0 whenever wave is running (i.e., producing)
    if (test_wave_ptr->is_running) {
        testGreaterThan(
            significant_wave_height_m,
            0,
            __FILE__,
```

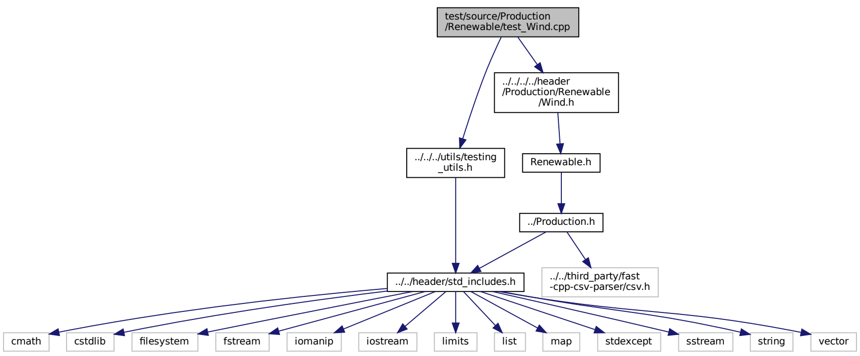
```
            __LINE__
        );

        testGreaterThan(
            energy_period_s,
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_wave_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    // O&M = 0 whenever wave is not running (i.e., not producing)
    else {
        testFloatEquals(
            test_wave_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }
}
// ======== END METHODS ============================================================= //

}   /* try */


catch (...) {
    delete test_wave_ptr;

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


delete test_wave_ptr;

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

## 5.37 test/source/Production/Renewable/test_Wind.cpp File Reference

Testing suite for Wind class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Wind.h"
```

Include dependency graph for test_Wind.cpp:



## Functions

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

### 5.37.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind class.

### 5.37.2 Function Documentation

#### 5.37.2.1 main()

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

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

    srand(time(NULL));

    Renewable* test_wind_ptr;

try {

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

bool error_flag = true;

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

    Wind bad_wind(8760, 1, bad_wind_inputs);

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

WindInputs wind_inputs;

test_wind_ptr = new Wind(8760, 1, wind_inputs);

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



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

testTruth(
    not wind_inputs.renewable_inputs.production_inputs.print_flag,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->type,
    RenewableType :: WIND,
    __FILE__,
```

```
    __LINE__
);

testTruth(
    test_wind_ptr->type_str == "WIND",
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->capital_cost,
    450356.170088,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->operation_maintenance_cost_kWh,
    0.034953,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ========================================================== //


// ======== METHODS ================================================================= //

// test production constraints
testFloatEquals(
    test_wind_ptr->computeProductionkW(0, 1, 1e6),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->computeProductionkW(
        0,
        1,
        ((Wind*)test_wind_ptr)->design_speed_ms
    ),
    test_wind_ptr->capacity_kW,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_wind_ptr->computeProductionkW(0, 1, -1),
    0,
    __FILE__,
    __LINE__
);

// test commit()
std::vector<double> dt_vec_hrs (48, 1);

std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
};

double load_kW = 0;
double production_kW = 0;
double roll = 0;
double wind_resource_ms = 0;

for (int i = 0; i < 48; i++) {
    roll = (double)rand() / RAND_MAX;

    wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;

    roll = (double)rand() / RAND_MAX;

    if (roll <= 0.1) {
        wind_resource_ms = 0;
    }

    else if (roll >= 0.95) {
        wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
    }

    roll = (double)rand() / RAND_MAX;

```

```
    if (roll >= 0.95) {
        roll = 1.25;
    }

    load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
    load_kW = load_vec_kW[i];

    production_kW = test_wind_ptr->computeProductionkW(
        i,
        dt_vec_hrs[i],
        wind_resource_ms
    );

    load_kW = test_wind_ptr->commit(
        i,
        dt_vec_hrs[i],
        production_kW,
        load_kW
    );

    // is running (or not) as expected
    if (production_kW > 0) {
        testTruth(
            test_wind_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    else {
        testTruth(
            not test_wind_ptr->is_running,
            __FILE__,
            __LINE__
        );
    }

    // load_kW <= load_vec_kW (i.e., after vs before)
    testLessThanOrEqualTo(
        load_kW,
        load_vec_kW[i],
        __FILE__,
        __LINE__
    );

    // production = dispatch + storage + curtailment
    testFloatEquals(
        test_wind_ptr->production_vec_kW[i] -
        test_wind_ptr->dispatch_vec_kW[i] -
        test_wind_ptr->storage_vec_kW[i] -
        test_wind_ptr->curtailment_vec_kW[i],
        0,
        __FILE__,
        __LINE__
    );

    // resource, O&M > 0 whenever wind is running (i.e., producing)
    if (test_wind_ptr->is_running) {
        testGreaterThan(
            wind_resource_ms,
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThan(
            test_wind_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    // O&M = 0 whenever wind is not running (i.e., not producing)
    else {
        testFloatEquals(
            test_wind_ptr->operation_maintenance_cost_vec[i],
            0,
            __FILE__,
            __LINE__
        );
    }
}


// ======== END METHODS ============================================================= //

```

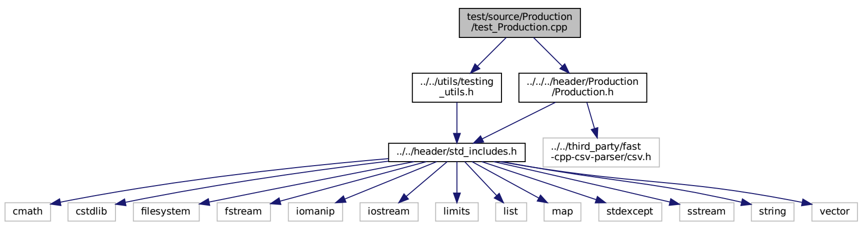
```
}   /* try */


catch (...) {
    delete test_wind_ptr;

    printGold(" ...... ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


delete test_wind_ptr;

printGold(" ...... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* 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 )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\n\tTesting Production");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    ProductionInputs production_inputs;

    Production bad_production(0, 1, production_inputs);

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

ProductionInputs production_inputs;

Production test_production(8760, 1, production_inputs);

// ======== END CONSTRUCTION ======================================================== //



// ======== ATTRIBUTES ============================================================== //

testTruth(
    not production_inputs.print_flag,
    __FILE__,
    __LINE__
);

testFloatEquals(
    production_inputs.nominal_inflation_annual,
    0.02,
    __FILE__,
    __LINE__
);

testFloatEquals(
    production_inputs.nominal_discount_annual,
    0.04,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.n_points,
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.capacity_kW,
    100,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.real_discount_annual,
    0.0196078431372549,
    __FILE__,
    __LINE__
);

```

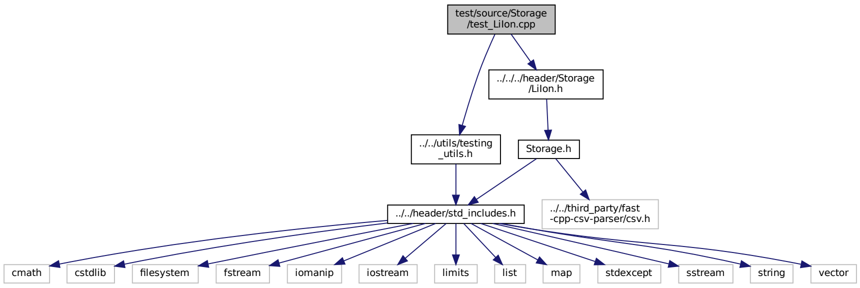
```
testFloatEquals(
    test_production.production_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.dispatch_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.storage_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.curtailment_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.capital_cost_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_production.operation_maintenance_cost_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ========================================================== //

}   /* try */


catch (...) {
    //...

    printGold(" ............................ ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ............................ ");
printGreen("PASS");
std::cout « std::endl;
return 0;

}   /* main() */
```

## 5.39 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 )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Storage <-- LiIon");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    LiIonInputs bad_liion_inputs;
    bad_liion_inputs.min_SOC = -1;

    LiIon bad_liion(8760, 1, bad_liion_inputs);

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
```

```
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

LiIonInputs liion_inputs;

LiIon test_liion(8760, 1, liion_inputs);

// ======== END CONSTRUCTION ========================================================= //



// ======== ATTRIBUTES ============================================================== //

testFloatEquals(
    test_liion.init_SOC,
    0.5,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.min_SOC,
    0.15,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.hysteresis_SOC,
    0.5,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.max_SOC,
    0.9,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.charging_efficiency,
    0.9,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.discharging_efficiency,
    0.9,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.replace_SOH,
    0.8,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.power_kW,
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.SOH_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ========================================================== //



// ======== METHODS ================================================================= //

testFloatEquals(
    test_liion.getAvailablekW(1),
    100,    // hits power capacity constraint
    __FILE__,
```

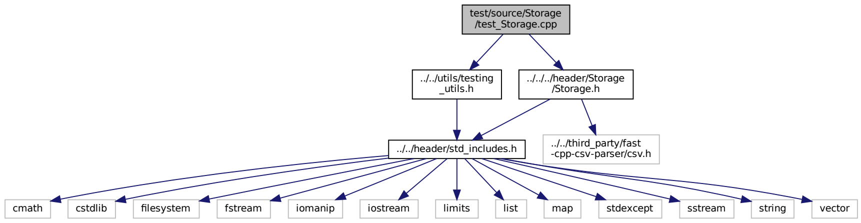
```
    __LINE__
);

testFloatEquals(
    test_liion.getAcceptablekW(1),
    100,    // hits power capacity constraint
    __FILE__,
    __LINE__
);

test_liion.power_kW = 100;

testFloatEquals(
    test_liion.getAvailablekW(1),
    100,    // hits power capacity constraint
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.getAcceptablekW(1),
    100,    // hits power capacity constraint
    __FILE__,
    __LINE__
);

test_liion.power_kW = 1e6;

testFloatEquals(
    test_liion.getAvailablekW(1),
    0,    // is already hitting power capacity constraint
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_liion.getAcceptablekW(1),
    0,    // is already hitting power capacity constraint
    __FILE__,
    __LINE__
);

test_liion.commitCharge(0, 1, 100);

testFloatEquals(
    test_liion.power_kW,
    0,
    __FILE__,
    __LINE__
);

// ======== END METHODS ============================================================= //

} /* try */


catch (...) {
    //...

    printGold(" ...................... ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ...................... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* 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 )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Storage");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    StorageInputs bad_storage_inputs;
    bad_storage_inputs.energy_capacity_kWh = 0;

    Storage bad_storage(8760, 1, bad_storage_inputs);

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}
```

```

StorageInputs storage_inputs;

Storage test_storage(8760, 1, storage_inputs);

// ======== END CONSTRUCTION ======================================================== //



// ======== ATTRIBUTES ============================================================== //

testFloatEquals(
    test_storage.power_capacity_kW,
    100,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.energy_capacity_kWh,
    1000,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.charge_vec_kWh.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.charging_power_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.discharging_power_vec_kW.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.capital_cost_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_storage.operation_maintenance_cost_vec.size(),
    8760,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ========================================================== //



// ======== METHODS ================================================================= //

//...

// ======== END METHODS ============================================================= //

}   /* try */


catch (...) {
    //...

    printGold(" ...................... ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ................................ ");
printGreen("PASS");
std::cout « std::endl;
return 0;
```

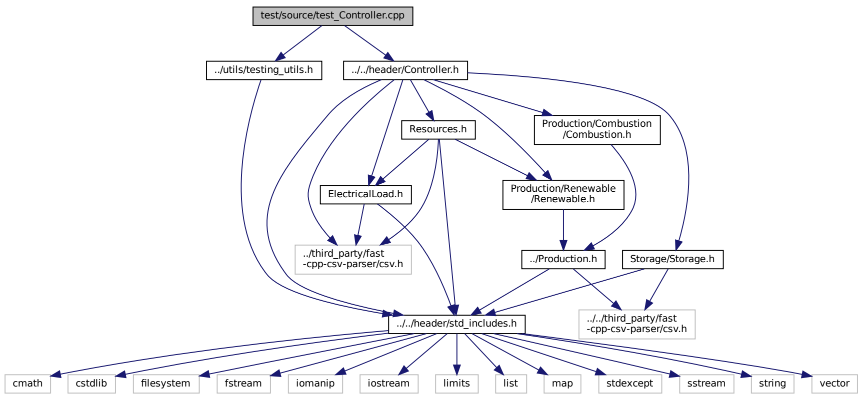
```
}   /* main() */
```

## 5.41 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 )
```

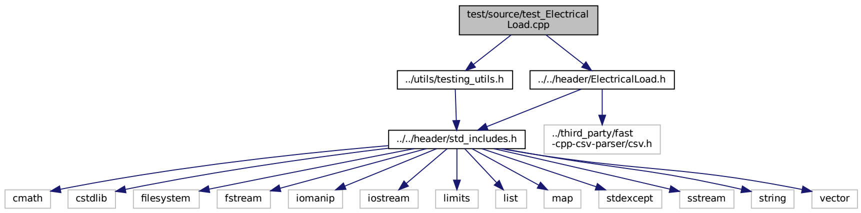
```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Controller");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

Controller test_controller;

// ======== END CONSTRUCTION ========================================================//



// ======== ATTRIBUTES ============================================================== //

//...

// ======== END ATTRIBUTES ========================================================== //



// ======== METHODS ================================================================= //

//...

// ======== END METHODS ============================================================= //

}   /* try */


catch (...) {
    //...

    printGold(" ............................ ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ............................ ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

## 5.42 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 )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting ElectricalLoad");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

std::string path_2_electrical_load_time_series =
    "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);

// ======== END CONSTRUCTION ======================================================== //



// ======== ATTRIBUTES ============================================================== //

testTruth(
    test_electrical_load.path_2_electrical_load_time_series ==
    path_2_electrical_load_time_series,
    __FILE__,
```

```
    __LINE__
);

testFloatEquals(
    test_electrical_load.n_points,
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_electrical_load.n_years,
    0.999886,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_electrical_load.min_load_kW,
    82.1211213927802,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_electrical_load.mean_load_kW,
    258.373472633202,
    __FILE__,
    __LINE__
);


testFloatEquals(
    test_electrical_load.max_load_kW,
    500,
    __FILE__,
    __LINE__
);


std::vector<double> expected_dt_vec_hrs (48, 1);

std::vector<double> expected_time_vec_hrs = {
     0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
    12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
};

std::vector<double> expected_load_vec_kW = {
    360.253836463674,
    355.171277826775,
    353.776453532298,
    353.75405737934,
    346.592867404975,
    340.132411175118,
    337.354867340578,
    340.644115618736,
    363.639028500678,
    378.787797779238,
    372.215798201712,
    395.093925731298,
    402.325427142659,
    386.907725462306,
    380.709170928091,
    372.062070914977,
    372.328646856954,
    391.841444284136,
    394.029351759596,
    383.369407765254,
    381.093099675206,
    382.604158946193,
    390.744843709034,
    383.13949492437,
    368.150393976985,
    364.629744480226,
    363.572736804082,
    359.854924202248,
    355.207590170267,
    349.094656012401,
    354.365935871597,
    343.380608328546,
    404.673065729266,
    486.296896820126,
    480.225974100847,
    457.318764401085,
    418.177339948609,
```

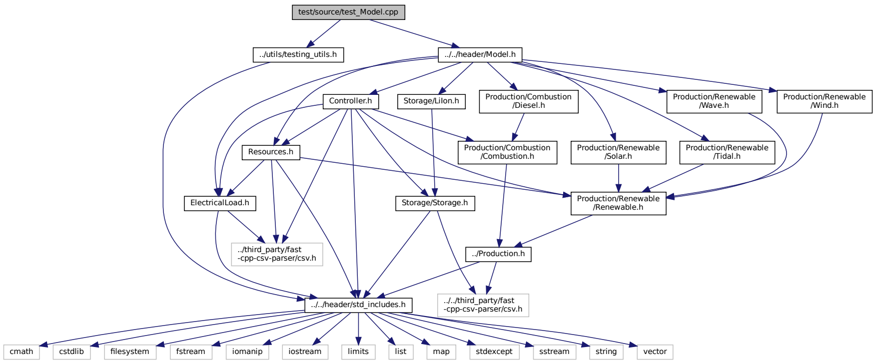
```
    414.399018364126,
    409.678420185754,
    404.768766016563,
    401.699589920585,
    402.44339040654,
    398.138372541906,
    396.010498627646,
    390.165117432277,
    375.850429417013,
    365.567100746484,
    365.429624610923
};

for (int i = 0; i < 48; i++) {
    testFloatEquals(
        test_electrical_load.dt_vec_hrs[i],
        expected_dt_vec_hrs[i],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_electrical_load.time_vec_hrs[i],
        expected_time_vec_hrs[i],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_electrical_load.load_vec_kW[i],
        expected_load_vec_kW[i],
        __FILE__,
        __LINE__
    );
}

// ======== END ATTRIBUTES ========================================================== //

}   /* try */


catch (...) {
    //...

    printGold(" ......................... ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" ......................... ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

## 5.43 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 )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Model");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ============================================================ //

bool error_flag = true;

try {
    ModelInputs bad_model_inputs;
    bad_model_inputs.path_2_electrical_load_time_series =
        "data/test/bad_path_240984069830.csv";

    Model bad_model(bad_model_inputs);

```

```
    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

std::string path_2_electrical_load_time_series =
    "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

ModelInputs test_model_inputs;
test_model_inputs.path_2_electrical_load_time_series =
    path_2_electrical_load_time_series;

Model test_model(test_model_inputs);

// ======== END CONSTRUCTION ======================================================== //


// ======== ATTRIBUTES ============================================================== //

testTruth(
    test_model.electrical_load.path_2_electrical_load_time_series ==
    path_2_electrical_load_time_series,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.electrical_load.n_points,
    8760,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.electrical_load.n_years,
    0.999886,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.electrical_load.min_load_kW,
    82.1211213927802,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.electrical_load.mean_load_kW,
    258.373472633202,
    __FILE__,
    __LINE__
);


testFloatEquals(
    test_model.electrical_load.max_load_kW,
    500,
    __FILE__,
    __LINE__
);


std::vector<double> expected_dt_vec_hrs (48, 1);

std::vector<double> expected_time_vec_hrs = {
     0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
    12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
};

std::vector<double> expected_load_vec_kW = {
    360.253836463674,
    355.171277826775,
    353.776453532298,
    353.75405737934,
    346.592867404975,
    340.132411175118,
    337.354867340578,
    340.644115618736,
    363.639028500678,
    378.787797779238,
    372.215798201712,
```

```
    395.093925731298,
    402.325427142659,
    386.907725462306,
    380.709170928091,
    372.062070914977,
    372.328646856954,
    391.841444284136,
    394.029351759596,
    383.369407765254,
    381.093099675206,
    382.604158946193,
    390.744843709034,
    383.13949492437,
    368.150393976985,
    364.629744480226,
    363.572736804082,
    359.854924202248,
    355.207590170267,
    349.094656012401,
    354.365935871597,
    343.380608328546,
    404.673065729266,
    486.296896820126,
    480.225974100847,
    457.318764401085,
    418.177339948609,
    414.399018364126,
    409.678420185754,
    404.768766016563,
    401.699589920585,
    402.44339040654,
    398.138372541906,
    396.010498627646,
    390.165117432277,
    375.850429417013,
    365.567100746484,
    365.429624610923
};

for (int i = 0; i < 48; i++) {
    testFloatEquals(
        test_model.electrical_load.dt_vec_hrs[i],
        expected_dt_vec_hrs[i],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model.electrical_load.time_vec_hrs[i],
        expected_time_vec_hrs[i],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model.electrical_load.load_vec_kW[i],
        expected_load_vec_kW[i],
        __FILE__,
        __LINE__
    );
}

// ======== END ATTRIBUTES ========================================================== //



// ======== METHODS ================================================================= //

//  add Solar resource
int solar_resource_key = 0;
std::string path_2_solar_resource_data =
    "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

test_model.addResource(
    RenewableType :: SOLAR,
    path_2_solar_resource_data,
    solar_resource_key
);

std::vector<double> expected_solar_resource_vec_kWm2 = {
    0,
    0,
    0,
    0,
    0,
    0,
    8.51702662684015E-05,
```

```
    0.000348341567045,
    0.00213793728593,
    0.004099863613322,
    0.000997135230553,
    0.009534527624657,
    0.022927996790616,
    0.0136071715294,
    0.002535134127751,
    0.005206897515821,
    0.005627658648597,
    0.000701186722215,
    0.00017119827089,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0.000141055102242,
    0.00084525014743,
    0.024893647822702,
    0.091245556190749,
    0.158722176731637,
    0.152859680515876,
    0.149922903895116,
    0.13049996570866,
    0.03081254222795,
    0.001218928911125,
    0.000206092647423,
    0,
    0,
    0,
    0,
    0,
    0
};

for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
    testFloatEquals(
        test_model.resources.resource_map_1D[solar_resource_key][i],
        expected_solar_resource_vec_kWm2[i],
        __FILE__,
        __LINE__
    );
}


//  add Tidal resource
int tidal_resource_key = 1;
std::string path_2_tidal_resource_data =
    "data/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

test_model.addResource(
    RenewableType :: TIDAL,
    path_2_tidal_resource_data,
    tidal_resource_key
);


//  add Wave resource
int wave_resource_key = 2;
std::string path_2_wave_resource_data =
    "data/test/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

test_model.addResource(
    RenewableType :: WAVE,
    path_2_wave_resource_data,
    wave_resource_key
);


//  add Wind resource
int wind_resource_key = 3;
std::string path_2_wind_resource_data =
    "data/test/wind_speed_peak-25ms_1yr_dt-1hr.csv";

test_model.addResource(
    RenewableType :: WIND,
    path_2_wind_resource_data,
    wind_resource_key
);
```

```


// add Diesel assets
DieselInputs diesel_inputs;
diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;

test_model.addDiesel(diesel_inputs);

testFloatEquals(
    test_model.combustion_ptr_vec.size(),
    1,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.combustion_ptr_vec[0]->type,
    CombustionType :: DIESEL,
    __FILE__,
    __LINE__
);

diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;

test_model.addDiesel(diesel_inputs);

diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;

test_model.addDiesel(diesel_inputs);

testFloatEquals(
    test_model.combustion_ptr_vec.size(),
    3,
    __FILE__,
    __LINE__
);

std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};

for (int i = 0; i < 3; i++) {
    testFloatEquals(
        test_model.combustion_ptr_vec[i]->capacity_kW,
        expected_diesel_capacity_vec_kW[i],
        __FILE__,
        __LINE__
    );
}

diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;

for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
    test_model.addDiesel(diesel_inputs);
}


// add Solar asset
SolarInputs solar_inputs;
solar_inputs.resource_key = solar_resource_key;

test_model.addSolar(solar_inputs);

testFloatEquals(
    test_model.renewable_ptr_vec.size(),
    1,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.renewable_ptr_vec[0]->type,
    RenewableType :: SOLAR,
    __FILE__,
    __LINE__
);


// add Tidal asset
TidalInputs tidal_inputs;
tidal_inputs.resource_key = tidal_resource_key;

test_model.addTidal(tidal_inputs);

testFloatEquals(
    test_model.renewable_ptr_vec.size(),
    2,
    __FILE__,
```

```
    __LINE__
);

testFloatEquals(
    test_model.renewable_ptr_vec[1]->type,
    RenewableType :: TIDAL,
    __FILE__,
    __LINE__
);


// add Wave asset
WaveInputs wave_inputs;
wave_inputs.resource_key = wave_resource_key;

test_model.addWave(wave_inputs);

testFloatEquals(
    test_model.renewable_ptr_vec.size(),
    3,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.renewable_ptr_vec[2]->type,
    RenewableType :: WAVE,
    __FILE__,
    __LINE__
);


// add Wind asset
WindInputs wind_inputs;
wind_inputs.resource_key = wind_resource_key;

test_model.addWind(wind_inputs);

testFloatEquals(
    test_model.renewable_ptr_vec.size(),
    4,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.renewable_ptr_vec[3]->type,
    RenewableType :: WIND,
    __FILE__,
    __LINE__
);


// add LiIon asset
LiIonInputs liion_inputs;

test_model.addLiIon(liion_inputs);

testFloatEquals(
    test_model.storage_ptr_vec.size(),
    1,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.storage_ptr_vec[0]->type,
    StorageType :: LIION,
    __FILE__,
    __LINE__
);


// run
test_model.run();


// write results
test_model.writeResults("test/test_results/");


// test post-run attributes
for (int i = 0; i < test_model.electrical_load.n_points; i++) {
    testLessThanOrEqualTo(
        test_model.controller.net_load_vec_kW[i],
        test_model.electrical_load.max_load_kW,
        __FILE__,
```

```
        __LINE__
    );
}

testGreaterThan(
    test_model.net_present_cost,
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_model.total_dispatch_discharge_kWh,
    2263351.62026685,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.levellized_cost_of_energy_kWh,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_fuel_consumed_L,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.CO2_kg,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.CO_kg,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.NOx_kg,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.SOx_kg,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.CH4_kg,
    0,
    __FILE__,
    __LINE__
);

testGreaterThan(
    test_model.total_emissions.PM_kg,
    0,
    __FILE__,
    __LINE__
);

// ======== END METHODS ============================================================= //

}   /* try */


catch (...) {
    //...

    printGold(" .................................. ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}
```

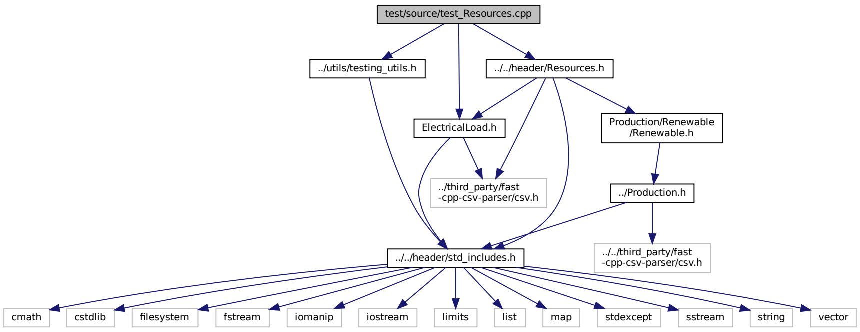
```
printGold(" .................................. ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

# 5.44 test/source/test_Resources.cpp File Reference

Testing suite for Resources class.

```
#include "../utils/testing_utils.h"
#include "../../header/Resources.h"
#include "../../header/ElectricalLoad.h"
```

Include dependency graph for test_Resources.cpp:




## Functions

- int main (int argc, char **argv)

### 5.44.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

### 5.44.2 Function Documentation

#### 5.44.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Resources");

    srand(time(NULL));


try {

// ======== CONSTRUCTION ==================================================== //

std::string path_2_electrical_load_time_series =
    "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);

Resources test_resources;

// ======== END CONSTRUCTION ================================================ //



// ======== ATTRIBUTES ====================================================== //

testFloatEquals(
    test_resources.resource_map_1D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.path_map_1D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.resource_map_2D.size(),
    0,
    __FILE__,
    __LINE__
);

testFloatEquals(
    test_resources.path_map_2D.size(),
    0,
    __FILE__,
    __LINE__
);

// ======== END ATTRIBUTES ================================================== //


// ======== METHODS ========================================================= //

int solar_resource_key = 0;
std::string path_2_solar_resource_data =
    "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

test_resources.addResource(
    RenewableType::SOLAR,
    path_2_solar_resource_data,
    solar_resource_key,
    &test_electrical_load
);

bool error_flag = true;
try {
    test_resources.addResource(
        RenewableType::SOLAR,
        path_2_solar_resource_data,
        solar_resource_key,
        &test_electrical_load
    );

```

```
    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}


try {
    std::string path_2_solar_resource_data_BAD_TIMES =
        "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";

    test_resources.addResource(
        RenewableType::SOLAR,
        path_2_solar_resource_data_BAD_TIMES,
        -1,
        &test_electrical_load
    );

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}


try {
    std::string path_2_solar_resource_data_BAD_LENGTH =
        "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";

    test_resources.addResource(
        RenewableType::SOLAR,
        path_2_solar_resource_data_BAD_LENGTH,
        -2,
        &test_electrical_load
    );

    error_flag = false;
} catch (...) {
    // Task failed successfully! =P
}
if (not error_flag) {
    expectedErrorNotDetected(__FILE__, __LINE__);
}

std::vector<double> expected_solar_resource_vec_kWm2 = {
    0,
    0,
    0,
    0,
    0,
    0,
    8.51702662684015E-05,
    0.000348341567045,
    0.00213793728593,
    0.004099863613322,
    0.000997135230553,
    0.009534527624657,
    0.022927996790616,
    0.0136071715294,
    0.002535134127751,
    0.005206897515821,
    0.005627658648597,
    0.000701186722215,
    0.00017119827089,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0.000141055102242,
    0.00084525014743,
    0.024893647822702,
    0.091245556190749,
    0.158722176731637,
    0.152859680515876,
    0.149922903895116,
```

```
    0.13049996570866,
    0.03081254222795,
    0.001218928911125,
    0.000206092647423,
    0,
    0,
    0,
    0,
    0,
    0
};

for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
    testFloatEquals(
        test_resources.resource_map_1D[solar_resource_key][i],
        expected_solar_resource_vec_kWm2[i],
        __FILE__,
        __LINE__
    );
}


int tidal_resource_key = 1;
std::string path_2_tidal_resource_data =
    "data/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

test_resources.addResource(
    RenewableType::TIDAL,
    path_2_tidal_resource_data,
    tidal_resource_key,
    &test_electrical_load
);

std::vector<double> expected_tidal_resource_vec_ms = {
    0.347439913040533,
    0.770545522195602,
    0.731352084836198,
    0.293389814389542,
    0.209959110813115,
    0.610609623896497,
    1.78067162013604,
    2.53522775118089,
    2.75966627832024,
    2.52101111143895,
    2.05389330201031,
    1.3461515862445,
    0.28909254878384,
    0.897754086048563,
    1.71406453837407,
    1.85047408742869,
    1.71507908595979,
    1.33540349705416,
    0.434586143463003,
    0.500623815700637,
    1.37172172646733,
    1.68294125491228,
    1.56101300975417,
    1.04925834219412,
    0.211395463930223,
    1.03720048903385,
    1.85059536356448,
    1.85203242794517,
    1.4091471616277,
    0.767776539039899,
    0.251464906990961,
    1.47018469375652,
    2.36260493698197,
    2.46653750048625,
    2.12851908739291,
    1.62783753197988,
    0.734594890957439,
    0.441886297300355,
    1.6574418350918,
    2.0684558286637,
    1.87717416992136,
    1.58871262337931,
    1.03451227609235,
    0.193371305159817,
    0.976400122458815,
    1.6583227369707,
    1.76690616570953,
    1.54801328553115
};

for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
    testFloatEquals(
        test_resources.resource_map_1D[tidal_resource_key][i],
```

```
        expected_tidal_resource_vec_ms[i],
        __FILE__,
        __LINE__
    );
}


int wave_resource_key = 2;
std::string path_2_wave_resource_data =
    "data/test/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

test_resources.addResource(
    RenewableType::WAVE,
    path_2_wave_resource_data,
    wave_resource_key,
    &test_electrical_load
);

std::vector<double> expected_significant_wave_height_vec_m = {
    4.26175222125028,
    4.25020976167872,
    4.25656524330349,
    4.27193854786718,
    4.28744955711233,
    4.29421815278154,
    4.2839937266082,
    4.25716982457976,
    4.22419391611483,
    4.19588925217606,
    4.17338788587412,
    4.14672746914214,
    4.10560041173665,
    4.05074966447193,
    3.9953696962433,
    3.95316976150866,
    3.92771018142378,
    3.91129562488595,
    3.89558312094911,
    3.87861093931749,
    3.86538307240754,
    3.86108961027929,
    3.86459448853189,
    3.86796474016882,
    3.86357412779993,
    3.85554872014731,
    3.86044266668675,
    3.89445961915999,
    3.95554798115731,
    4.02265508610476,
    4.07419587011404,
    4.10314247143958,
    4.11738045085928,
    4.12554995596708,
    4.12923992001675,
    4.1229292327442,
    4.10123955307441,
    4.06748827895363,
    4.0336230651344,
    4.01134236393876,
    4.00136570034559,
    3.99368787690411,
    3.97820924247644,
    3.95369335178055,
    3.92742545608532,
    3.90683362771686,
    3.89331520944006,
    3.88256045801583
};

std::vector<double> expected_energy_period_vec_s = {
    10.4456008226821,
    10.4614151137651,
    10.4462827795433,
    10.4127692097884,
    10.3734397942723,
    10.3408599227669,
    10.32637292093,
    10.3245412676322,
    10.310409818185,
    10.2589529840966,
    10.1728100603103,
    10.0862908658929,
    10.03480243813,
    10.023673635806,
    10.0243418565116,
    10.0063487117653,
    9.96050302286607,
```

```
    9.9011999635568,
    9.84451822125472,
    9.79726875879626,
    9.75614594835158,
    9.7173447961368,
    9.68342904390577,
    9.66380508567062,
    9.6674009575699,
    9.68927134575103,
    9.70979984863046,
    9.70967357906908,
    9.68983025704562,
    9.6722855524805,
    9.67973599910003,
    9.71977125328293,
    9.78450442291421,
    9.86532355233449,
    9.96158937600019,
    10.0807018356507,
    10.2291022504937,
    10.39458528356,
    10.5464393581004,
    10.6553277500484,
    10.7245553190084,
    10.7893127285064,
    10.8846512240849,
    11.0148158739075,
    11.1544325654719,
    11.2772785848343,
    11.3744362756187,
    11.4533643503183
};

for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
    testFloatEquals(
        test_resources.resource_map_2D[wave_resource_key][i][0],
        expected_significant_wave_height_vec_m[i],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_resources.resource_map_2D[wave_resource_key][i][1],
        expected_energy_period_vec_s[i],
        __FILE__,
        __LINE__
    );
}


int wind_resource_key = 3;
std::string path_2_wind_resource_data =
    "data/test/wind_speed_peak-25ms_1yr_dt-1hr.csv";

test_resources.addResource(
    RenewableType::WIND,
    path_2_wind_resource_data,
    wind_resource_key,
    &test_electrical_load
);

std::vector<double> expected_wind_resource_vec_ms = {
    6.88566688469997,
    5.02177105466549,
    3.74211715899568,
    5.67169579985362,
    4.90670669971858,
    4.29586955031368,
    7.41155377205065,
    10.2243290476943,
    13.1258696725555,
    13.7016198628274,
    16.2481482330233,
    16.5096744355418,
    13.4354482206162,
    14.0129230731609,
    14.5554549260515,
    13.4454539065912,
    13.3447169512094,
    11.7372615098554,
    12.7200070078013,
    10.6421127908149,
    6.09869498990661,
    5.66355596602321,
    4.97316966910831,
    3.48937138360567,
    2.15917470979169,
```

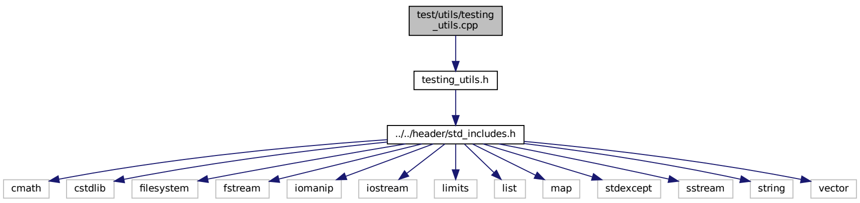
```
    1.29061103587027,
    3.43475751425219,
    4.11706326260927,
    4.28905275747408,
    5.75850263196241,
    8.98293663055264,
    11.7069822941315,
    12.4031987075858,
    15.4096570910089,
    16.6210843829552,
    13.3421219142573,
    15.2112831900548,
    18.350864533037,
    15.8751799822971,
    15.3921198799796,
    15.9729192868434,
    12.4728950178772,
    10.177050481096,
    10.7342247355551,
    8.98846695631389,
    4.14671169124739,
    3.17256452697149,
    3.40036336968628
};

for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
    testFloatEquals(
        test_resources.resource_map_1D[wind_resource_key][i],
        expected_wind_resource_vec_ms[i],
        __FILE__,
        __LINE__
    );
}

// ======== END METHODS ============================================================= //

}   /* try */


catch (...) {
    printGold(" .............................. ");
    printRed("FAIL");
    std::cout « std::endl;
    throw;
}


printGold(" .............................. ");
printGreen("PASS");
std::cout « std::endl;
return 0;
}   /* main() */
```

## 5.45 test/utils/testing_utils.cpp File Reference

Header file for various PGMcpp testing utilities.

`#include "testing_utils.h"`

Include dependency graph for testing_utils.cpp:



## Functions

- void printGreen (std::string input_str)

  *A function that sends green text to std::cout.*
- void printGold (std::string input_str)

  *A function that sends gold text to std::cout.*
- void printRed (std::string input_str)

  *A function that sends red text to std::cout.*
- void testFloatEquals (double x, double y, std::string file, int line)

  *Tests for the equality of two floating point numbers x and y (to within FLOAT_TOLERANCE).*
- void testGreaterThan (double x, double y, std::string file, int line)

  *Tests if $x > y$.*
- void testGreaterThanOrEqualTo (double x, double y, std::string file, int line)

  *Tests if $x >= y$.*
- void testLessThan (double x, double y, std::string file, int line)

  *Tests if $x < y$.*
- void testLessThanOrEqualTo (double x, double y, std::string file, int line)

  *Tests if $x <= y$.*
- void testTruth (bool statement, std::string file, int line)

  *Tests if the given statement is true.*
- void expectedErrorNotDetected (std::string file, int line)

  *A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.*

## 5.45.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

## 5.45.2 Function Documentation

### 5.45.2.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
            std::string file,
            int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

**Parameters**

| | |
|---|---|
| *file* | The file in which the test is applied (you should be able to just pass in "__FILE__"). |
| *line* | The line of the file in which the test is applied (you should be able to just pass in "__LINE__"). |

```
{
    std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
    error_str += std::to_string(line);
```

```
    error_str += " of ";
    error_str += file;

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[33m" « input_str « "\033[0m";
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[32m" « input_str « "\033[0m";
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[31m" « input_str « "\033[0m";
    return;
}   /* 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__"). |

```
{
    if (fabs(x - y) <= FLOAT_TOLERANCE) {
        return;
    }

    std::string error_str = "ERROR: testFloatEquals():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " and ";
    error_str += std::to_string(y);
    error_str += " are not equal to within +/- ";
    error_str += std::to_string(FLOAT_TOLERANCE);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (x > y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (x >= y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
```

```
    return;
}   /* 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__"). |

```
{
    if (x < y) {
        return;
    }

    std::string error_str = "ERROR: testLessThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (x <= y) {
        return;
    }

    std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testLessThanOrEqualTo() */
```

#### 5.45.2.10 testTruth()

```
void testTruth (
            bool statement,
            std::string file,
            int line )
```

Tests if the given statement is true.

**Parameters**

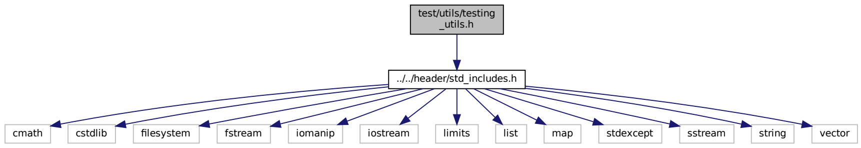
| | |
|---|---|
| *statement* | The statement whose truth is to be tested ("1 == 0", for example). |
| *file* | The file in which the test is applied (you should be able to just pass in "__FILE__"). |
| *line* | The line of the file in which the test is applied (you should be able to just pass in "__LINE__"). |

```
{
    if (statement) {
        return;
    }

    std::string error_str = "ERROR: testTruth():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += "Given statement is not true";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testTruth() */
```

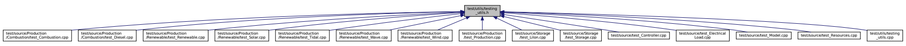
## 5.46 test/utils/testing_utils.h File Reference

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__"). |

```
{
    std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
    error_str += std::to_string(line);
    error_str += " of ";
    error_str += file;

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[33m" « input_str « "\033[0m";
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[32m" « input_str « "\033[0m";
    return;
}   /* 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. |

```
{
    std::cout « "\x1B[31m" « input_str « "\033[0m";
    return;
}   /* 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__"). |

```
{
    if (fabs(x - y) <= FLOAT_TOLERANCE) {
        return;
    }

    std::string error_str = "ERROR: testFloatEquals():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " and ";
    error_str += std::to_string(y);
    error_str += " are not equal to within +/- ";
    error_str += std::to_string(FLOAT_TOLERANCE);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (x > y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

```

```
    throw std::runtime_error(error_str);
    return;
} /* 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__"). |

```
{
    if (x >= y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout << error_str << std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
} /* 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__"). |

```
{
    if (x < y) {
        return;
    }

    std::string error_str = "ERROR: testLessThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (x <= y) {
        return;
    }

    std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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__"). |

```
{
    if (statement) {
        return;
    }

    std::string error_str = "ERROR: testTruth():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += "Given statement is not true";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testTruth() */
```

# Bibliography


Dr. B. Buckham, Dr. C. Crawford, Dr. I. Beya Marshall, and Dr. B. Whitby. Wei Wai Kum Tidal Prefeasibility Study - Tidal Resource Assessment. Technical report, PRIMED, 2023. Internal: P2202E_BRKLYG+WEI WAI KUM_R01_V20230613v3. 155

CIMAC. Guide to Diesel Exhaust Emissions Control of NOx, SOx, Particulates, Smoke, and CO2. Technical report, Conseil International des Machines à Combustion, 2008. Included: docs/refs/diesel_emissions_ref_2.pdf. 51

HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 99, 143

HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 13, 99, 100, 141, 143

HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 42, 43, 51

HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 42, 51

HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 43, 51

HOMER. How HOMER Calculates the PV Array Power Output, 2023f. URL https://www.homerenergy.com/products/pro/docs/latest/how_homer_calculates_the_pv_array_power_output.html. 133

HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 99, 143

HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 100, 141

HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 99, 143

Dr. S.L. MacDougall. Commercial Potential of Marine Renewables in British Columbia. Technical report, S.L. MacDougall Research & Consulting, 2019. Submitted to Natural Resources Canada. 157, 158, 172

NRCan. Auto$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 51

Dr. B. Robertson, Dr. H. Bailey, M. Leary, and Dr. B. Buckham. A methodology for architecture agnostic and time flexible representations of wave energy converter performance. *Applied Energy*, 287, 2021. doi:10.1016/j.apenergy.2021.116588. 171

A. Truelove, Dr. B. Buckham, Dr. C. Crawford, and C. Hiles. Scaling Technology Models for HOMER Pro: Wind, Tidal Stream, and Wave. Technical report, PRIMED, 2019. Included: docs/refs/wind_tidal_wave.pdf. 156, 170, 185

# Index