PGMcpp: PRIMED Grid Modelling (in C++)

Generated by Doxygen 1.9.1

1 Hierarchical Index	1
1.1 Class Hierarchy	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation 7	7
4.1 Combustion Class Reference	7
4.1.1 Detailed Description	9
4.1.2 Constructor & Destructor Documentation	9
4.1.2.1 Combustion() [1/2] 9	9
4.1.2.2 Combustion() [2/2] 9	9
4.1.2.3 ~Combustion()	)
4.1.3 Member Function Documentation	)
4.1.3.1 commit()	1
4.1.3.2 getEmissionskg()	2
4.1.3.3 getFuelConsumptionL()	3
4.1.3.4 requestProductionkW()	3
4.1.4 Member Data Documentation	3
4.1.4.1 CH4_emissions_intensity_kgL	4
4.1.4.2 CH4_emissions_vec_kg	4
4.1.4.3 CO2_emissions_intensity_kgL	4
4.1.4.4 CO2_emissions_vec_kg	4
4.1.4.5 CO_emissions_intensity_kgL	4
4.1.4.6 CO_emissions_vec_kg	4
4.1.4.7 fuel_consumption_vec_L	5
4.1.4.8 fuel_cost_L	5
4.1.4.9 fuel_cost_vec	5
4.1.4.10 linear_fuel_intercept_LkWh	5
4.1.4.11 linear_fuel_slope_LkWh	5
4.1.4.12 NOx_emissions_intensity_kgL	5
4.1.4.13 NOx_emissions_vec_kg	3
4.1.4.14 PM_emissions_intensity_kgL	3
4.1.4.15 PM_emissions_vec_kg	3
4.1.4.16 SOx_emissions_intensity_kgL	3
4.1.4.17 SOx_emissions_vec_kg	3
4.1.4.18 type	
4.2 CombustionInputs Struct Reference	
4.2.1 Detailed Description	7
4.2.2 Member Data Documentation	7

4.2.2.1 production_inputs	17
4.3 Controller Class Reference	18
4.3.1 Detailed Description	18
4.3.2 Constructor & Destructor Documentation	18
4.3.2.1 Controller()	18
4.3.2.2 ~Controller()	18
4.3.3 Member Function Documentation	19
4.3.3.1 clear()	19
4.3.4 Member Data Documentation	19
4.3.4.1 control_mode	19
4.4 Diesel Class Reference	19
4.4.1 Detailed Description	20
4.4.2 Constructor & Destructor Documentation	21
<b>4.4.2.1 Diesel()</b> [1/2]	21
<b>4.4.2.2</b> Diesel() [2/2]	21
4.4.2.3 ∼Diesel()	22
4.4.3 Member Function Documentation	22
4.4.3.1 commit()	22
4.4.3.2 requestProductionkW()	23
4.4.4 Member Data Documentation	23
4.4.4.1 minimum_load_ratio	24
4.4.4.2 minimum_runtime_hrs	24
4.4.4.3 time_since_last_start_hrs	24
4.5 DieselInputs Struct Reference	24
4.5.1 Detailed Description	25
4.5.2 Member Data Documentation	25
4.5.2.1 capital_cost	26
4.5.2.2 CH4_emissions_intensity_kgL	26
4.5.2.3 CO2_emissions_intensity_kgL	26
4.5.2.4 CO_emissions_intensity_kgL	26
4.5.2.5 combustion_inputs	26
4.5.2.6 fuel_cost_L	26
4.5.2.7 linear_fuel_intercept_LkWh	27
4.5.2.8 linear_fuel_slope_LkWh	27
4.5.2.9 minimum_load_ratio	27
4.5.2.10 minimum_runtime_hrs	27
4.5.2.11 NOx_emissions_intensity_kgL	27
4.5.2.12 operation_maintenance_cost_kWh	28
4.5.2.13 PM_emissions_intensity_kgL	28
4.5.2.14 replace_running_hrs	28
4.5.2.15 SOx_emissions_intensity_kgL	28
4.6 ElectricalLoad Class Reference	28

4.6.1 Detailed Description	29
4.6.2 Constructor & Destructor Documentation	29
4.6.2.1 ElectricalLoad() [1/2]	30
4.6.2.2 ElectricalLoad() [2/2]	30
4.6.2.3 ∼ElectricalLoad()	30
4.6.3 Member Function Documentation	30
4.6.3.1 clear()	30
4.6.3.2 readLoadData()	31
4.6.4 Member Data Documentation	32
4.6.4.1 dt_vec_hrs	32
4.6.4.2 load_vec_kW	32
4.6.4.3 max_load_kW	32
4.6.4.4 mean_load_kW	32
4.6.4.5 min_load_kW	33
4.6.4.6 n_points	33
4.6.4.7 n_years	33
4.6.4.8 path_2_electrical_load_time_series	33
4.6.4.9 time_vec_hrs	33
4.7 Emissions Struct Reference	33
4.7.1 Detailed Description	34
4.7.2 Member Data Documentation	34
4.7.2.1 CH4_kg	34
4.7.2.2 CO2_kg	34
4.7.2.3 CO_kg	34
4.7.2.4 NOx_kg	35
4.7.2.5 PM_kg	35
4.7.2.6 SOx_kg	35
4.8 Lilon Class Reference	35
4.8.1 Detailed Description	36
4.8.2 Constructor & Destructor Documentation	36
4.8.2.1 Lilon()	36
4.8.2.2 ~Lilon()	37
4.9 Model Class Reference	37
4.9.1 Detailed Description	38
4.9.2 Constructor & Destructor Documentation	38
4.9.2.1 Model() [1/2]	38
4.9.2.2 Model() [2/2]	38
4.9.2.3 ∼Model()	39
4.9.3 Member Function Documentation	39
4.9.3.1 addResource()	39
4.9.3.2 clear()	40
4.9.3.3 reset()	40

4.9.4 Member Data Documentation	40
4.9.4.1 combustion_ptr_vec	41
4.9.4.2 controller	41
4.9.4.3 electrical_load	41
4.9.4.4 renewable_ptr_vec	41
4.9.4.5 resources	41
4.9.4.6 storage_ptr_vec	41
4.10 ModelInputs Struct Reference	42
4.10.1 Detailed Description	42
4.10.2 Member Data Documentation	42
4.10.2.1 control_mode	42
4.10.2.2 path_2_electrical_load_time_series	42
4.11 Production Class Reference	43
4.11.1 Detailed Description	44
4.11.2 Constructor & Destructor Documentation	44
4.11.2.1 Production() [1/2]	45
<b>4.11.2.2 Production()</b> [2/2]	45
4.11.2.3 ∼Production()	46
4.11.3 Member Function Documentation	46
4.11.3.1 commit()	46
4.11.4 Member Data Documentation	47
4.11.4.1 capacity_kW	47
4.11.4.2 capital_cost	47
4.11.4.3 capital_cost_vec	47
4.11.4.4 curtailment_vec_kW	48
4.11.4.5 dispatch_vec_kW	48
4.11.4.6 is_running	48
4.11.4.7 is_running_vec	48
4.11.4.8 is_sunk	48
4.11.4.9 levellized_cost_of_energy_kWh	48
4.11.4.10 n_points	49
4.11.4.11 n_replacements	49
4.11.4.12 n_starts	49
4.11.4.13 net_present_cost	49
4.11.4.14 operation_maintenance_cost_kWh	49
4.11.4.15 operation_maintenance_cost_vec	49
4.11.4.16 print_flag	50
4.11.4.17 production_vec_kW	50
4.11.4.18 real_discount_annual	50
4.11.4.19 replace_running_hrs	50
4.11.4.20 running_hours	50
4.11.4.21 storage_vec_kW	50

4.11.4.22 type_str	51
4.12 ProductionInputs Struct Reference	51
4.12.1 Detailed Description	51
4.12.2 Member Data Documentation	51
4.12.2.1 capacity_kW	52
4.12.2.2 is_sunk	52
4.12.2.3 nominal_discount_annual	52
4.12.2.4 nominal_inflation_annual	52
4.12.2.5 print_flag	52
4.12.2.6 replace_running_hrs	52
4.13 Renewable Class Reference	53
4.13.1 Detailed Description	54
4.13.2 Constructor & Destructor Documentation	54
4.13.2.1 Renewable() [1/2]	54
<b>4.13.2.2</b> Renewable() [2/2]	54
4.13.2.3 ∼Renewable()	55
4.13.3 Member Function Documentation	55
4.13.3.1 commit()	55
4.13.3.2 computeProductionkW() [1/2]	56
4.13.3.3 computeProductionkW() [2/2]	56
4.13.4 Member Data Documentation	56
4.13.4.1 resource_key	56
4.13.4.2 type	57
4.14 RenewableInputs Struct Reference	57
4.14.1 Detailed Description	57
4.14.2 Member Data Documentation	57
4.14.2.1 production_inputs	58
4.15 Resources Class Reference	58
4.15.1 Detailed Description	58
4.15.2 Constructor & Destructor Documentation	58
4.15.2.1 Resources()	59
4.15.2.2 ∼Resources()	59
4.15.3 Member Function Documentation	59
4.15.3.1 addResource1D()	59
4.15.3.2 addResource2D()	60
4.15.3.3 clear()	60
4.15.4 Member Data Documentation	60
4.15.4.1 path_map_1D	60
4.15.4.2 path_map_2D	61
4.15.4.3 resource_map_1D	61
4.15.4.4 resource_map_2D	61
4.16 Solar Class Reference	61

4.16.1 Detailed Description	62
4.16.2 Constructor & Destructor Documentation	62
<b>4.16.2.1 Solar()</b> [1/2]	63
<b>4.16.2.2 Solar()</b> [2/2]	64
4.16.2.3 ∼Solar()	64
4.16.3 Member Function Documentation	65
4.16.3.1 commit()	65
4.16.3.2 computeProductionkW()	65
4.16.4 Member Data Documentation	66
4.16.4.1 derating	66
4.17 SolarInputs Struct Reference	66
4.17.1 Detailed Description	67
4.17.2 Member Data Documentation	67
4.17.2.1 capital_cost	68
4.17.2.2 derating	68
4.17.2.3 operation_maintenance_cost_kWh	68
4.17.2.4 renewable_inputs	68
4.17.2.5 resource_key	68
4.18 Storage Class Reference	69
4.18.1 Detailed Description	69
4.18.2 Constructor & Destructor Documentation	69
4.18.2.1 Storage()	69
4.18.2.2 ∼Storage()	70
4.19 Tidal Class Reference	70
4.19.1 Detailed Description	71
4.19.2 Constructor & Destructor Documentation	71
<b>4.19.2.1 Tidal()</b> [1/2]	72
<b>4.19.2.2 Tidal()</b> [2/2]	72
4.19.2.3 ∼Tidal()	72
4.19.3 Member Function Documentation	73
4.19.3.1 commit()	73
4.19.3.2 computeProductionkW()	73
4.19.4 Member Data Documentation	74
4.19.4.1 design_speed_ms	74
4.19.4.2 power_model	75
4.20 TidalInputs Struct Reference	75
4.20.1 Detailed Description	76
4.20.2 Member Data Documentation	76
4.20.2.1 capital_cost	76
4.20.2.2 design_speed_ms	76
4.20.2.3 operation_maintenance_cost_kWh	76
4.20.2.4 power_model	77

4.20.2.5 renewable_inputs	77
4.20.2.6 resource_key	77
4.21 Wave Class Reference	77
4.21.1 Detailed Description	78
4.21.2 Constructor & Destructor Documentation	78
<b>4.21.2.1 Wave()</b> [1/2]	79
<b>4.21.2.2 Wave()</b> [2/2]	79
4.21.2.3 ∼Wave()	80
4.21.3 Member Function Documentation	80
4.21.3.1 commit()	80
4.21.3.2 computeProductionkW()	81
4.21.4 Member Data Documentation	82
4.21.4.1 design_energy_period_s	82
4.21.4.2 design_significant_wave_height_m	82
4.21.4.3 power_model	82
4.22 WaveInputs Struct Reference	82
4.22.1 Detailed Description	83
4.22.2 Member Data Documentation	83
4.22.2.1 capital_cost	83
4.22.2.2 design_energy_period_s	83
4.22.2.3 design_significant_wave_height_m	84
4.22.2.4 operation_maintenance_cost_kWh	84
4.22.2.5 power_model	84
4.22.2.6 renewable_inputs	84
4.22.2.7 resource_key	84
4.23 Wind Class Reference	85
4.23.1 Detailed Description	86
4.23.2 Constructor & Destructor Documentation	86
<b>4.23.2.1 Wind()</b> [1/2]	86
<b>4.23.2.2 Wind()</b> [2/2]	86
4.23.2.3 $\sim$ Wind()	87
4.23.3 Member Function Documentation	87
4.23.3.1 commit()	87
4.23.3.2 computeProductionkW()	88
4.23.4 Member Data Documentation	89
4.23.4.1 design_speed_ms	89
4.23.4.2 power_model	89
4.24 WindInputs Struct Reference	90
4.24.1 Detailed Description	90
4.24.2 Member Data Documentation	91
4.24.2.1 capital_cost	91
4.24.2.2 design_speed_ms	91

4.24.2.3 operation_maintenance_cost_kWh	 91
4.24.2.4 power_model	 91
4.24.2.5 renewable_inputs	 91
4.24.2.6 resource_key	 91
5 File Documentation	93
5.1 header/Controller.h File Reference	 93
5.1.1 Detailed Description	
5.1.2 Enumeration Type Documentation	
5.1.2.1 ControlMode	
5.2 header/ElectricalLoad.h File Reference	
5.2.1 Detailed Description	
5.3 header/Model.h File Reference	
5.3.1 Detailed Description	 96
5.4 header/Production/Combustion/Combustion.h File Reference	 96
5.4.1 Detailed Description	 97
5.4.2 Enumeration Type Documentation	
5.4.2.1 CombustionType	 97
5.5 header/Production/Combustion/Diesel.h File Reference	 98
5.5.1 Detailed Description	 98
5.6 header/Production/Production.h File Reference	 99
5.6.1 Detailed Description	 99
5.7 header/Production/Renewable/Renewable.h File Reference	 99
5.7.1 Detailed Description	 100
5.7.2 Enumeration Type Documentation	 100
5.7.2.1 RenewableType	 100
5.8 header/Production/Renewable/Solar.h File Reference	 101
5.8.1 Detailed Description	 101
5.9 header/Production/Renewable/Tidal.h File Reference	 102
5.9.1 Detailed Description	 102
5.9.2 Enumeration Type Documentation	 103
5.9.2.1 TidalPowerProductionModel	 103
5.10 header/Production/Renewable/Wave.h File Reference	 103
5.10.1 Detailed Description	 104
5.10.2 Enumeration Type Documentation	 104
5.10.2.1 WavePowerProductionModel	 104
5.11 header/Production/Renewable/Wind.h File Reference	 105
5.11.1 Detailed Description	 106
5.11.2 Enumeration Type Documentation	 106
5.11.2.1 WindPowerProductionModel	 106
5.12 header/Resources.h File Reference	 106
5.12.1 Detailed Description	 107

5.13 header/std_includes.h File Reference
5.13.1 Detailed Description
5.14 header/Storage/Lilon.h File Reference
5.14.1 Detailed Description
5.15 header/Storage/Storage.h File Reference
5.15.1 Detailed Description
5.16 pybindings/PYBIND11_PGM.cpp File Reference
5.16.1 Detailed Description
5.16.2 Function Documentation
5.16.2.1 PYBIND11_MODULE()
5.17 source/Controller.cpp File Reference
5.17.1 Detailed Description
5.18 source/ElectricalLoad.cpp File Reference
5.18.1 Detailed Description
5.19 source/Model.cpp File Reference
5.19.1 Detailed Description
5.20 source/Production/Combustion/Combustion.cpp File Reference
5.20.1 Detailed Description
5.21 source/Production/Combustion/Diesel.cpp File Reference
5.21.1 Detailed Description
5.22 source/Production/Production.cpp File Reference
5.22.1 Detailed Description
5.23 source/Production/Renewable/Renewable.cpp File Reference
5.23.1 Detailed Description
5.24 source/Production/Renewable/Solar.cpp File Reference
5.24.1 Detailed Description
5.25 source/Production/Renewable/Tidal.cpp File Reference
5.25.1 Detailed Description
5.26 source/Production/Renewable/Wave.cpp File Reference
5.26.1 Detailed Description
5.27 source/Production/Renewable/Wind.cpp File Reference
5.27.1 Detailed Description
5.28 source/Resources.cpp File Reference
5.28.1 Detailed Description
5.29 source/Storage/Lilon.cpp File Reference
5.29.1 Detailed Description
5.30 source/Storage/Storage.cpp File Reference
5.30.1 Detailed Description
5.31 test/source/Production/Combustion/test_Combustion.cpp File Reference
5.31.1 Detailed Description
5.31.2 Function Documentation
5.31.2.1 main()

5.32 test/source/Production/Combustion/test_Diesel.cpp File Reference	20
5.32.1 Detailed Description	21
5.32.2 Function Documentation	21
5.32.2.1 main()	21
5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference	26
5.33.1 Detailed Description	26
5.33.2 Function Documentation	26
5.33.2.1 main()	27
5.34 test/source/Production/Renewable/test_Solar.cpp File Reference	27
5.34.1 Detailed Description	28
5.34.2 Function Documentation	28
5.34.2.1 main()	28
5.35 test/source/Production/Renewable/test_Tidal.cpp File Reference	31
5.35.1 Detailed Description	32
5.35.2 Function Documentation	32
5.35.2.1 main()	32
5.36 test/source/Production/Renewable/test_Wave.cpp File Reference	35
5.36.1 Detailed Description	36
5.36.2 Function Documentation	36
5.36.2.1 main()	36
5.37 test/source/Production/Renewable/test_Wind.cpp File Reference	39
5.37.1 Detailed Description	39
5.37.2 Function Documentation	39
5.37.2.1 main()	10
5.38 test/source/Production/test_Production.cpp File Reference	12
5.38.1 Detailed Description	13
5.38.2 Function Documentation	13
5.38.2.1 main()	13
5.39 test/source/Storage/test_Lilon.cpp File Reference	15
5.39.1 Detailed Description	15
5.39.2 Function Documentation	16
5.39.2.1 main()	16
5.40 test/source/Storage/test_Storage.cpp File Reference	16
5.40.1 Detailed Description	17
5.40.2 Function Documentation	17
5.40.2.1 main()	17
5.41 test/source/test_Controller.cpp File Reference	17
5.41.1 Detailed Description	18
5.41.2 Function Documentation	18
5.41.2.1 main()	18
5.42 test/source/test_ElectricalLoad.cpp File Reference	18
5.42.1 Detailed Description	19

	5.42.2 Function Documentation	149
	5.42.2.1 main()	149
5	5.43 test/source/test_Model.cpp File Reference	151
	5.43.1 Detailed Description	152
	5.43.2 Function Documentation	152
	5.43.2.1 main()	152
5	5.44 test/source/test_Resources.cpp File Reference	155
	5.44.1 Detailed Description	155
	5.44.2 Function Documentation	155
	5.44.2.1 main()	156
5	5.45 test/utils/testing_utils.cpp File Reference	157
	5.45.1 Detailed Description	157
	5.45.2 Function Documentation	157
	5.45.2.1 expectedErrorNotDetected()	158
	5.45.2.2 printGold()	158
	5.45.2.3 printGreen()	158
	5.45.2.4 printRed()	159
	5.45.2.5 testFloatEquals()	159
	5.45.2.6 testGreaterThan()	160
	5.45.2.7 testGreaterThanOrEqualTo()	160
	5.45.2.8 testLessThan()	161
	5.45.2.9 testLessThanOrEqualTo()	162
	5.45.2.10 testTruth()	162
5	5.46 test/utils/testing_utils.h File Reference	163
	5.46.1 Detailed Description	164
	5.46.2 Macro Definition Documentation	164
	5.46.2.1 FLOAT_TOLERANCE	164
	5.46.3 Function Documentation	164
	5.46.3.1 expectedErrorNotDetected()	164
	5.46.3.2 printGold()	165
	5.46.3.3 printGreen()	165
	5.46.3.4 printRed()	165
	5.46.3.5 testFloatEquals()	166
	5.46.3.6 testGreaterThan()	166
	5.46.3.7 testGreaterThanOrEqualTo()	167
	5.46.3.8 testLessThan()	168
	5.46.3.9 testLessThanOrEqualTo()	168
	5.46.3.10 testTruth()	169
Inde	.X	171
muc	n.	

# **Chapter 1**

# **Hierarchical Index**

## 1.1 Class Hierarchy

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

CombustionInputs	7
Controller	8
DieselInputs	4
ElectricalLoad	8
Emissions	3
Model	7
ModelInputs	2
Production	3
Combustion	7
Diesel	9
Renewable	3
Solar	1
Tidal	
Wave	7
Wind	5
ProductionInputs	1
RenewableInputs	7
Resources	8
SolarInputs	6
Storage	9
Lilon	5
TidalInputs	
WaveInputs	-
Windlands	

2 Hierarchical Index

# **Chapter 2**

# **Class Index**

## 2.1 Class List

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

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	7
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	17
Controller	
A class which contains a various dispatch control logic. Intended to serve as a component class of Model	18
Diesel	
A derived class of the Combustion branch of Production which models production using a diesel generator	19
Diesellnputs	
A structure which bundles the necessary inputs for the Diesel constructor. Provides default values for every necessary input. Note that this structure encapsulates CombustionInputs	24
ElectricalLoad	
A class which contains time and electrical load data. Intended to serve as a component class of Model	28
Emissions	
A structure which bundles the emitted masses of various emissions chemistries Lilon	33
A derived class of Storage which models energy storage by way of lithium-ion batteries Model	35
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	37
ModelInputs	0,
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)	42
Production	
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise	43
ProductionInputs	
A structure which bundles the necessary inputs for the Production constructor. Provides default	
values for every necessary input	51

Class Index

Renewa	able	
	The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy	53
Renewa	ableInputs	
	A structure which bundles the necessary inputs for the Renewable constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs	57
Resour	ces	
	A class which contains renewable resource data. Intended to serve as a component class of Model	58
Solar		
	A derived class of the Renewable branch of Production which models solar production	61
SolarIn	puts	
	A structure which bundles the necessary inputs for the Solar constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs	66
Storage		
	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	69
Tidal		
	A derived class of the Renewable branch of Production which models tidal production	70
TidalInp		
	A structure which bundles the necessary inputs for the Tidal constructor. Provides default values for every necessary input. Note that this structure encapsulates RenewableInputs	75
Wave		
	A derived class of the Renewable branch of Production which models wave production	77
WaveIn	puts	
	A structure which bundles the necessary inputs for the Wave constructor. Provides default values	
	for every necessary input. Note that this structure encapsulates RenewableInputs	82
Wind		
	A derived class of the Renewable branch of Production which models wind production	85
WindIn		
	A structure which bundles the necessary inputs for the Wind constructor. Provides default values	
	for every necessary input. Note that this structure encapsulates RenewableInputs	90

# **Chapter 3**

# File Index

## 3.1 File List

Here is a list of all files with brief descriptions:

header/Controller.h
Header file the Controller class
header/ElectricalLoad.h
Header file the ElectricalLoad class
header/Model.h
Header file the Model class
header/Resources.h
Header file the Resources class
header/std_includes.h
Header file which simply batches together the usual, standard includes
header/Production/Production.h
Header file the Production class
header/Production/Combustion.h
Header file the Combustion class
header/Production/Combustion/Diesel.h
Header file the Diesel class
header/Production/Renewable/Renewable.h
Header file the Renewable class
header/Production/Renewable/Solar.h
Header file the Solar class
header/Production/Renewable/Tidal.h
Header file the Tidal class
header/Production/Renewable/Wave.h
Header file the Wave class
header/Production/Renewable/Wind.h
Header file the Wind class
header/Storage/Lilon.h
Header file the Lilon class
header/Storage/Storage.h
Header file the Storage class
pybindings/PYBIND11_PGM.cpp
Python 3 bindings file for PGMcpp
source/Controller.cpp
Implementation file for the Controller class
source/ElectricalLoad.cpp
Implementation file for the ElectricalLoad class

6 File Index

source/Model.cpp	
Implementation file for the Model class	112
source/Resources.cpp	
Implementation file for the Resources class	117
source/Production/Production.cpp	
Implementation file for the Production class	114
source/Production/Combustion/Combustion.cpp	
Implementation file for the Combustion class	113
source/Production/Combustion/Diesel.cpp	
Implementation file for the Diesel class	113
source/Production/Renewable/Renewable.cpp	
Implementation file for the Renewable class	114
source/Production/Renewable/Solar.cpp	
Implementation file for the Solar class	115
source/Production/Renewable/Tidal.cpp	
Implementation file for the Tidal class	115
source/Production/Renewable/Wave.cpp	
Implementation file for the Wave class	116
source/Production/Renewable/Wind.cpp	
Implementation file for the Wind class	116
source/Storage/Lilon.cpp	
Implementation file for the Lilon class	117
source/Storage/Storage.cpp	440
Implementation file for the Storage class	118
test/source/test_Controller.cpp	4 4 7
Testing suite for Controller class	147
test/source/test_ElectricalLoad.cpp	4.40
Testing suite for ElectricalLoad class	148
test/source/test_Model.cpp	4.54
Testing suite for Model class	151
test/source/test_Resources.cpp  Testing suite for Resources class	155
	155
test/source/Production/test_Production.cpp  Testing suite for Production class	142
test/source/Production/Combustion/test Combustion.cpp	142
Testing suite for Combustion class	118
test/source/Production/Combustion/test_Diesel.cpp	110
Testing suite for Diesel class	120
test/source/Production/Renewable/test_Renewable.cpp	120
Testing suite for Renewable class	126
test/source/Production/Renewable/test_Solar.cpp	120
Testing suite for Solar class	127
test/source/Production/Renewable/test_Tidal.cpp	121
Testing suite for Tidal class	131
test/source/Production/Renewable/test_Wave.cpp	
Testing suite for Wave class	135
test/source/Production/Renewable/test_Wind.cpp	.00
Testing suite for Wind class	139
test/source/Storage/test_Lilon.cpp	
Testing suite for Lilon class	145
test/source/Storage/test_Storage.cpp	
Testing suite for Storage class	146
test/utils/testing_utils.cpp	
Header file for various PGMcpp testing utilities	157
test/utils/testing_utils.h	
Header file for various PGMcpp testing utilities	163

## **Chapter 4**

## **Class Documentation**

## 4.1 Combustion Class Reference

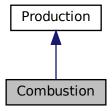
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

#include <Combustion.h>

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



#### **Public Member Functions**

· Combustion (void)

Constructor (dummy) for the Combustion class.

Combustion (int, CombustionInputs)

Constructor (intended) for the Combustion class.

- virtual double requestProductionkW (int, double, double)
- virtual double commit (int, double, double, double)

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

double getFuelConsumptionL (double, double)

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

• Emissions getEmissionskg (double)

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

virtual ∼Combustion (void)

Destructor for the Combustion class.

#### **Public Attributes**

• CombustionType type

The type (CombustionType) of the asset.

double fuel cost L

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

· double linear fuel slope LkWh

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

double linear\_fuel\_intercept\_LkWh

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

· double CO2 emissions intensity kgL

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

· double CO\_emissions\_intensity\_kgL

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

· double NOx emissions intensity kgL

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

• double SOx\_emissions\_intensity\_kgL

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

double CH4 emissions intensity kgL

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

double PM\_emissions\_intensity\_kgL

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

• std::vector< double > fuel\_consumption\_vec\_L

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

std::vector< double > fuel cost vec

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

std::vector< double > CO2\_emissions\_vec\_kg

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

std::vector< double > CO\_emissions\_vec\_kg

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

std::vector< double > NOx emissions vec kg

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

std::vector< double > SOx\_emissions\_vec\_kg

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

std::vector< double > CH4\_emissions\_vec\_kg

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

std::vector< double > PM\_emissions\_vec\_kg

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

### 4.1.1 Detailed Description

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

#### 4.1.2 Constructor & Destructor Documentation

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

Constructor (dummy) for the Combustion class.

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

## 4.1.2.2 Combustion() [2/2]

Constructor (intended) for the Combustion class.

#### **Parameters**

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

```
79
80 Production(n_points, combustion_inputs.production_inputs)
81 {
82
        // 1. check inputs
        this->__checkInputs(combustion_inputs);
84
       // 2. set attributes
8.5
       this->fuel_cost_L = 0;
86
87
88
        this->linear_fuel_slope_LkWh = 0;
89
        this->linear_fuel_intercept_LkWh = 0;
90
91
        this->CO2_emissions_intensity_kgL = 0;
       this->CO_emissions_intensity_kgL = 0;
this->NOx_emissions_intensity_kgL = 0;
92
93
        this->SOx_emissions_intensity_kgL = 0;
94
95
        this->CH4_emissions_intensity_kgL = 0;
96
        this->PM_emissions_intensity_kgL = 0;
97
98
        \label{lem:consumption_vec_L.resize} this \verb|-> fuel_consumption_vec_L.resize(this \verb|-> n_points, 0);
        this->fuel_cost_vec.resize(this->n_points, 0);
99
100
101
         this->CO2_emissions_vec_kg.resize(this->n_points, 0);
102
         this->CO_emissions_vec_kg.resize(this->n_points, 0);
103
         this->NOx_emissions_vec_kg.resize(this->n_points, 0);
         this->SOx_emissions_vec_kg.resize(this->n_points, 0);
104
        this->CH4_emissions_vec_kg.resize(this->n_points, 0);
this->PM_emissions_vec_kg.resize(this->n_points, 0);
105
106
107
108
         // 3. construction print
         if (this->print_flag) {
    std::cout « "Combustion object constructed at " « this « std::endl;
109
110
111
112
113
         return;
114 }
        /* Combustion() */
```

### 4.1.2.3 ∼Combustion()

```
\label{eq:combustion:combustion} \mbox{Combustion::$$\sim$Combustion (} \mbox{void ) [virtual]}
```

#### Destructor for the Combustion class.

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

#### 4.1.3 Member Function Documentation

## 4.1.3.1 commit()

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

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

#### **Parameters**

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

#### Returns

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

Reimplemented from Production.

#### Reimplemented in Diesel.

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

## 4.1.3.2 getEmissionskg()

```
\begin{tabular}{ll} {\tt Emissions} & {\tt Combustion::getEmissionskg} & (\\ & & {\tt double} & {\tt fuel\_consumed\_L} & ) \end{tabular}
```

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

#### **Parameters**

fuel_consumed↔	The volume of fuel consumed [L].
_L	

#### Returns

A structure containing the mass spectrum of resulting emissions.

```
226
                                                                                             {
227
           Emissions emissions;
228
229
           \verb|emissions.CO2_kg| = \verb|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;
230
231
232
233
           emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
234
           emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
235
236
           return emissions:
237 }
          /* getEmissionskg() */
```

## 4.1.3.3 getFuelConsumptionL()

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].

#### 4.1.3.4 requestProductionkW()

Reimplemented in Diesel.

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

#### 4.1.4 Member Data Documentation

### 4.1.4.1 CH4\_emissions\_intensity\_kgL

 $\verb|double Combustion::CH4_emissions_intensity_kgL|\\$ 

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

#### 4.1.4.2 CH4\_emissions\_vec\_kg

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

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

## 4.1.4.3 CO2\_emissions\_intensity\_kgL

double Combustion::CO2\_emissions\_intensity\_kgL

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

## 4.1.4.4 CO2\_emissions\_vec\_kg

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

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

#### 4.1.4.5 CO emissions intensity kgL

double Combustion::CO\_emissions\_intensity\_kgL

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

## 4.1.4.6 CO\_emissions\_vec\_kg

std::vector<double> Combustion::CO\_emissions\_vec\_kg

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

#### 4.1.4.7 fuel\_consumption\_vec\_L

std::vector<double> Combustion::fuel\_consumption\_vec\_L

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

#### 4.1.4.8 fuel\_cost\_L

double Combustion::fuel\_cost\_L

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

#### 4.1.4.9 fuel\_cost\_vec

std::vector<double> Combustion::fuel\_cost\_vec

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

## 4.1.4.10 linear\_fuel\_intercept\_LkWh

double Combustion::linear\_fuel\_intercept\_LkWh

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

#### 4.1.4.11 linear\_fuel\_slope\_LkWh

double Combustion::linear\_fuel\_slope\_LkWh

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

#### 4.1.4.12 NOx\_emissions\_intensity\_kgL

 $\verb|double Combustion::NOx\_emissions\_intensity\_kgL|\\$ 

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

#### 4.1.4.13 NOx\_emissions\_vec\_kg

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

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

#### 4.1.4.14 PM\_emissions\_intensity\_kgL

```
\verb|double Combustion::PM_emissions_intensity_kgL|\\
```

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

#### 4.1.4.15 PM\_emissions\_vec\_kg

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

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

### 4.1.4.16 SOx\_emissions\_intensity\_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

## 4.1.4.17 SOx\_emissions\_vec\_kg

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

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

#### 4.1.4.18 type

CombustionType Combustion::type

The type (CombustionType) of the asset.

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

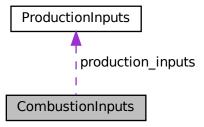
- header/Production/Combustion/Combustion.h
- source/Production/Combustion.cpp

## 4.2 CombustionInputs Struct Reference

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

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

Collaboration diagram for CombustionInputs:



#### **Public Attributes**

ProductionInputs production\_inputs
 An encapsulated ProductionInputs instance.

## 4.2.1 Detailed Description

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

#### 4.2.2 Member Data Documentation

#### 4.2.2.1 production\_inputs

ProductionInputs CombustionInputs::production\_inputs

An encapsulated ProductionInputs instance.

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

• header/Production/Combustion/Combustion.h

## 4.3 Controller Class Reference

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

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

#### **Public Member Functions**

• Controller (void)

Constructor for the Controller class.

void clear (void)

Method to clear all attributes of the Controller object.

Controller (void)

Destructor for the Controller class.

## **Public Attributes**

• ControlMode control\_mode

## 4.3.1 Detailed Description

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

### 4.3.2 Constructor & Destructor Documentation

### 4.3.2.1 Controller()

## Constructor for the Controller class.

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

## 4.3.2.2 $\sim$ Controller()

```
Controller::\simController ( void )
```

#### Destructor for the Controller class.

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

4.4 Diesel Class Reference

## 4.3.3 Member Function Documentation

### 4.3.3.1 clear()

Method to clear all attributes of the Controller object.

## 4.3.4 Member Data Documentation

#### 4.3.4.1 control mode

```
ControlMode Controller::control_mode
```

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

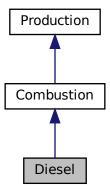
- · header/Controller.h
- source/Controller.cpp

## 4.4 Diesel Class Reference

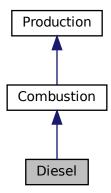
A derived class of the Combustion branch of Production which models production using a diesel generator.

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

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



## **Public Member Functions**

• Diesel (void)

Constructor (dummy) for the Diesel class.

- Diesel (int, DieselInputs)
- double requestProductionkW (int, double, double)

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

• double commit (int, double, double, double)

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

∼Diesel (void)

Destructor for the Diesel class.

## **Public Attributes**

• double minimum\_load\_ratio

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

· double minimum runtime hrs

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

• double time\_since\_last\_start\_hrs

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

## 4.4.1 Detailed Description

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

4.4 Diesel Class Reference 21

#### 4.4.2 Constructor & Destructor Documentation

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

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

Constructor (dummy) for the Diesel class.

Constructor (intended) for the Diesel class.

#### **Parameters**

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

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

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

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

#### 4.4.2.3 ∼Diesel()

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

#### Destructor for the Diesel class.

### 4.4.3 Member Function Documentation

#### 4.4.3.1 commit()

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

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

#### **Parameters**

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

### Reimplemented from Combustion.

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

4.4 Diesel Class Reference 23

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

## 4.4.3.2 requestProductionkW()

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

### **Parameters**

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

#### Reimplemented from Combustion.

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

### 4.4.4 Member Data Documentation

#### 4.4.4.1 minimum\_load\_ratio

```
double Diesel::minimum_load_ratio
```

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

## 4.4.4.2 minimum\_runtime\_hrs

```
double Diesel::minimum_runtime_hrs
```

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

### 4.4.4.3 time\_since\_last\_start\_hrs

```
double Diesel::time_since_last_start_hrs
```

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

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

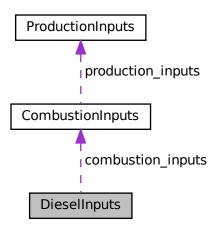
- · header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

# 4.5 DieselInputs Struct Reference

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

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

Collaboration diagram for DieselInputs:



#### **Public Attributes**

CombustionInputs combustion inputs

An encapsulated CombustionInputs instance.

double replace running hrs = 30000

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

• double capital\_cost = -1

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

double operation maintenance cost kWh = -1

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

• double fuel\_cost\_L = 1.70

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

• double minimum load ratio = 0.2

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

• double minimum\_runtime\_hrs = 4

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

double linear\_fuel\_slope\_LkWh = -1

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

• double linear\_fuel\_intercept\_LkWh = -1

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

double CO2\_emissions\_intensity\_kgL = 2.7

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

double CO emissions intensity kgL = 0.0178

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

double NOx\_emissions\_intensity\_kgL = 0.0014

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

• double SOx\_emissions\_intensity\_kgL = 0.0042

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

• double CH4\_emissions\_intensity\_kgL = 0.0007

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

double PM\_emissions\_intensity\_kgL = 0.0001

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

## 4.5.1 Detailed Description

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

### 4.5.2 Member Data Documentation

### 4.5.2.1 capital\_cost

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

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

## 4.5.2.2 CH4\_emissions\_intensity\_kgL

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

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

### 4.5.2.3 CO2\_emissions\_intensity\_kgL

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

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

## 4.5.2.4 CO\_emissions\_intensity\_kgL

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

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

### 4.5.2.5 combustion\_inputs

```
CombustionInputs DieselInputs::combustion_inputs
```

An encapsulated CombustionInputs instance.

## 4.5.2.6 fuel\_cost\_L

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

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

## 4.5.2.7 linear\_fuel\_intercept\_LkWh

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

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

# 4.5.2.8 linear\_fuel\_slope\_LkWh

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

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

## 4.5.2.9 minimum\_load\_ratio

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

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

## 4.5.2.10 minimum\_runtime\_hrs

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

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

### 4.5.2.11 NOx\_emissions\_intensity\_kgL

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

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

### 4.5.2.12 operation\_maintenance\_cost\_kWh

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

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

# 4.5.2.13 PM\_emissions\_intensity\_kgL

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

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

### 4.5.2.14 replace\_running\_hrs

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

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

# 4.5.2.15 SOx\_emissions\_intensity\_kgL

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

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

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

• header/Production/Combustion/Diesel.h

## 4.6 ElectricalLoad Class Reference

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

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

#### **Public Member Functions**

· ElectricalLoad (void)

Constructor (dummy) for the ElectricalLoad class.

ElectricalLoad (std::string)

Constructor (intended) for the ElectricalLoad class.

void readLoadData (std::string)

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

void clear (void)

Method to clear all attributes of the ElectricalLoad object.

∼ElectricalLoad (void)

Destructor for the ElectricalLoad class.

### **Public Attributes**

• int n points

The number of points in the modelling time series.

double n years

The number of years being modelled (inferred from time\_vec\_hrs).

· double min\_load\_kW

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

· double mean load kW

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

double max\_load\_kW

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

• std::string path\_2\_electrical\_load\_time\_series

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

std::vector< double > time\_vec\_hrs

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

std::vector< double > dt vec hrs

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

std::vector< double > load\_vec\_kW

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

### 4.6.1 Detailed Description

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

## 4.6.2 Constructor & Destructor Documentation

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

Constructor (dummy) for the ElectricalLoad class.

## 4.6.2.2 ElectricalLoad() [2/2]

Constructor (intended) for the ElectricalLoad class.

#### **Parameters**

path\_2\_electrical\_load\_time\_series A string defining the path (either relative or absolute) to the given electrical load time series.

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

## 4.6.2.3 ∼ElectricalLoad()

Destructor for the ElectricalLoad class.

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

# 4.6.3 Member Function Documentation

### 4.6.3.1 clear()

Method to clear all attributes of the ElectricalLoad object.

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

## 4.6.3.2 readLoadData()

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

#### **Parameters**

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

```
79 {
80
       // 1. clear
81
       this->clear();
82
       // 2. init CSV reader, record path
83
       io::CSVReader<2> CSV(path_2_electrical_load_time_series);
85
86
       CSV.read_header(
           io::ignore_extra_column,
"Time (since start of data) [hrs]",
"Electrical Load [kW]"
87
88
89
90
92
       this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;
93
       // 3. read in time and load data, increment n_points, track min and max load
94
95
       double time_hrs = 0;
96
       double load_kW = 0;
       double load_sum_kW = 0;
98
99
       this->n_points = 0;
100
        this->min_load_kW = std::numeric_limits<double>::infinity();
101
        this->max_load_kW = -1 * std::numeric_limits<double>::infinity();
102
103
104
        while (CSV.read_row(time_hrs, load_kW)) {
105
             this->time_vec_hrs.push_back(time_hrs);
106
            this->load_vec_kW.push_back(load_kW);
107
108
            load_sum_kW += load_kW;
109
110
111
            if (this->min_load_kW > load_kW) {
112
                 this->min_load_kW = load_kW;
113
114
115
116
            if (this->max_load_kW < load_kW) {</pre>
117
                 this->max_load_kW = load_kW;
118
119
120
121
        // 4. compute mean load
        this->mean_load_kW = load_sum_kW / this->n_points;
122
123
124
        // 5. set number of years (assuming 8,760 hours per year)
125
        this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;
126
```

```
127
          // 6. populate dt_vec_hrs
this->dt_vec_hrs.resize(n_points, 0);
128
129
          for (int i = 0; i < n_points; i++) {
    if (i == n_points - 1) {
        this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
}
130
131
132
133
134
135
               else {
                     double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];
136
137
138
                    this->dt_vec_hrs[i] = dt_hrs;
               }
139
140
141
         return;
/* readLoadData() */
142
143 }
```

# 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].

4.8 Lilon Class Reference 35

# 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:

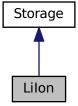
• header/Production/Combustion/Combustion.h

# 4.8 Lilon Class Reference

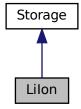
A derived class of Storage which models energy storage by way of lithium-ion batteries.

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

Inheritance diagram for Lilon:



Collaboration diagram for Lilon:



# **Public Member Functions**

• Lilon (void)

Constructor for the Lilon class.

• ∼Lilon (void)

Destructor for the Lilon class.

# 4.8.1 Detailed Description

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

# 4.8.2 Constructor & Destructor Documentation

# 4.8.2.1 Lilon()

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

# Constructor for the Lilon class.

4.9 Model Class Reference 37

### 4.8.2.2 ∼Lilon()

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

Destructor for the Lilon class.

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

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

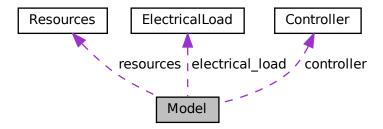
- header/Storage/Lilon.h
- source/Storage/Lilon.cpp

# 4.9 Model Class Reference

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

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

Collaboration diagram for Model:



# **Public Member Functions**

· Model (void)

Constructor (dummy) for the Model class.

Model (ModelInputs)

Constructor (intended) for the Model class.

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

A method to add a renewable resource time series to the Model.

void reset (void)

Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors; it leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

void clear (void)

Method to clear all attributes of the Model object.

∼Model (void)

Destructor for the Model class.

## **Public Attributes**

· Controller controller

Controller component of Model.

· ElectricalLoad electrical\_load

ElectricalLoad component of Model.

Resources resources

Resources component of Model.

• std::vector< Combustion \* > combustion\_ptr\_vec

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

std::vector< Renewable \* > renewable\_ptr\_vec

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

std::vector< Storage \* > storage\_ptr\_vec

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

# 4.9.1 Detailed Description

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

#### 4.9.2 Constructor & Destructor Documentation

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

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

Constructor (dummy) for the Model class.

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

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

Constructor (intended) for the Model class.

## **Parameters**

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

4.9 Model Class Reference 39

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

## 4.9.2.3 ∼Model()

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

### Destructor for the Model class.

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

## 4.9.3 Member Function Documentation

#### 4.9.3.1 addResource()

A method to add a renewable resource time series to the Model.

### **Parameters**

renewable_type	The type of renewable resource being added to the Model.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time
	series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.

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

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

## 4.9.3.2 clear()

Method to clear all attributes of the Model object.

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

# 4.9.3.3 reset()

Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors; it leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

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

## 4.9.4 Member Data Documentation

4.9 Model Class Reference 41

## 4.9.4.1 combustion\_ptr\_vec

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

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

## 4.9.4.2 controller

Controller Model::controller

Controller component of Model.

## 4.9.4.3 electrical\_load

ElectricalLoad Model::electrical\_load

ElectricalLoad component of Model.

# 4.9.4.4 renewable\_ptr\_vec

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

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

# 4.9.4.5 resources

Resources Model::resources

Resources component of Model.

# 4.9.4.6 storage\_ptr\_vec

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

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

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

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

# 4.10 ModelInputs Struct Reference

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path\_2\_electrical\_load\_time\_series, for which a valid input must be provided).

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

## **Public Attributes**

- std::string path\_2\_electrical\_load\_time\_series = ""
  - A string defining the path (either relative or absolute) to the given electrical load time series.
- ControlMode control\_mode = ControlMode :: LOAD\_FOLLOWING

The control mode to be applied by the Controller object.

# 4.10.1 Detailed Description

A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path 2 electrical load time series, for which a valid input must be provided).

## 4.10.2 Member Data Documentation

### 4.10.2.1 control\_mode

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

The control mode to be applied by the Controller object.

# 4.10.2.2 path\_2\_electrical\_load\_time\_series

```
std::string ModelInputs::path_2_electrical_load_time_series = ""
```

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

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

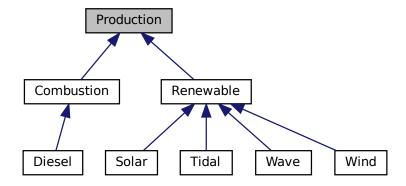
· header/Model.h

# 4.11 Production Class Reference

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

#include <Production.h>

Inheritance diagram for Production:



# **Public Member Functions**

• Production (void)

Constructor (dummy) for the Production class.

• Production (int, ProductionInputs)

Constructor (intended) for the Production class.

• virtual double commit (int, double, double, double)

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

virtual ∼Production (void)

Destructor for the Production class.

## **Public Attributes**

· bool print\_flag

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

bool is\_running

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

bool is\_sunk

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

• int n\_points

The number of points in the modelling time series.

• int n\_starts

The number of times the asset has been started.

· int n replacements

The number of times the asset has been replaced.

· double running hours

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

double replace\_running\_hrs

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

double capacity kW

The rated production capacity [kW] of the asset.

double real discount annual

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

· double capital cost

The capital cost of the asset (undefined currency).

double operation\_maintenance\_cost\_kWh

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.

• double net\_present\_cost

The net present cost of this asset.

• double levellized\_cost\_of\_energy\_kWh

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

· std::string type\_str

A string describing the type of the asset.

std::vector< bool > is running vec

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

std::vector< double > production\_vec\_kW

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

std::vector< double > dispatch\_vec\_kW

A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.

• std::vector< double > storage\_vec\_kW

A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.

• std::vector< double > curtailment\_vec\_kW

A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.

std::vector< double > capital\_cost\_vec

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

• std::vector< double > operation\_maintenance\_cost\_vec

A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are nominal costs).

## 4.11.1 Detailed Description

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

## 4.11.2 Constructor & Destructor Documentation

#### 4.11.2.1 Production() [1/2]

Constructor (dummy) for the Production class.

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

## 4.11.2.2 Production() [2/2]

Constructor (intended) for the Production class.

#### **Parameters**

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

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

### 4.11.2.3 ∼Production()

return;

/\* ~Production() \*/

## 4.11.3 Member Function Documentation

### 4.11.3.1 commit()

306

307 }

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

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

### **Parameters**

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

## Returns

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

### Reimplemented in Wind, Wave, Tidal, Solar, Renewable, Diesel, and Combustion.

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

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

## 4.11.4 Member Data Documentation

## 4.11.4.1 capacity\_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

### 4.11.4.2 capital\_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

## 4.11.4.3 capital\_cost\_vec

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

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

#### 4.11.4.4 curtailment\_vec\_kW

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

A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.

## 4.11.4.5 dispatch\_vec\_kW

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

A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.

### 4.11.4.6 is\_running

bool Production::is\_running

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

# 4.11.4.7 is\_running\_vec

```
std::vector<bool> Production::is_running_vec
```

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

## 4.11.4.8 is\_sunk

bool Production::is\_sunk

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

## 4.11.4.9 levellized\_cost\_of\_energy\_kWh

```
\verb|double Production::levellized_cost_of_energy_kWh|\\
```

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

### 4.11.4.10 n\_points

int Production::n\_points

The number of points in the modelling time series.

## 4.11.4.11 n\_replacements

int Production::n\_replacements

The number of times the asset has been replaced.

# 4.11.4.12 n\_starts

 $\verb"int Production": n\_starts"$ 

The number of times the asset has been started.

## 4.11.4.13 net\_present\_cost

double Production::net\_present\_cost

The net present cost of this asset.

## 4.11.4.14 operation\_maintenance\_cost\_kWh

double Production::operation\_maintenance\_cost\_kWh

The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.

## 4.11.4.15 operation\_maintenance\_cost\_vec

 $\verb|std::vector<| double > Production::operation_maintenance_cost\_vec| \\$ 

A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are nominal costs).

## 4.11.4.16 print\_flag

```
bool Production::print_flag
```

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

# 4.11.4.17 production\_vec\_kW

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

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

## 4.11.4.18 real\_discount\_annual

```
double Production::real_discount_annual
```

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

# 4.11.4.19 replace\_running\_hrs

```
double Production::replace_running_hrs
```

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

## 4.11.4.20 running\_hours

```
double Production::running_hours
```

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

## 4.11.4.21 storage\_vec\_kW

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

A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.

### 4.11.4.22 type\_str

```
std::string Production::type_str
```

A string describing the type of the asset.

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

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

# 4.12 ProductionInputs Struct Reference

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

```
#include <Production.h>
```

#### **Public Attributes**

• bool print flag = false

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

bool is\_sunk = false

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

• double capacity kW = 100

The rated production capacity [kW] of the asset.

• double nominal\_inflation\_annual = 0.02

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

• double nominal discount annual = 0.04

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

• double replace\_running\_hrs = 90000

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

# 4.12.1 Detailed Description

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

# 4.12.2 Member Data Documentation

## 4.12.2.1 capacity\_kW

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

The rated production capacity [kW] of the asset.

## 4.12.2.2 is sunk

```
bool ProductionInputs::is_sunk = false
```

A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).

### 4.12.2.3 nominal\_discount\_annual

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

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

# 4.12.2.4 nominal\_inflation\_annual

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

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

# 4.12.2.5 print\_flag

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

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

## 4.12.2.6 replace\_running\_hrs

```
double ProductionInputs::replace_running_hrs = 90000
```

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

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

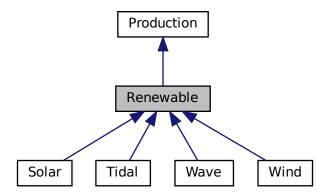
header/Production/Production.h

# 4.13 Renewable Class Reference

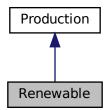
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

#include <Renewable.h>

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



# **Public Member Functions**

- Renewable (void)
  - Constructor (dummy) for the Renewable class.
- Renewable (int, RenewableInputs)
- virtual double computeProductionkW (int, double, double)
- virtual double computeProductionkW (int, double, double, double)
- virtual double commit (int, double, double, double)

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

virtual ∼Renewable (void)

Destructor for the Renewable class.

# **Public Attributes**

· RenewableType type

The type (RenewableType) of the asset.

int resource\_key

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

# 4.13.1 Detailed Description

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

#### 4.13.2 Constructor & Destructor Documentation

### 4.13.2.1 Renewable() [1/2]

Constructor (dummy) for the Renewable class.

Constructor (intended) for the Renewable class.

#### **Parameters**

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

## 4.13.2.2 Renewable() [2/2]

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

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

## 4.13.2.3 ∼Renewable()

198 return; 199 } /\* ~Renewable() \*/

## 4.13.3 Member Function Documentation

### 4.13.3.1 commit()

197

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

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

### **Parameters**

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

### Returns

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

Reimplemented from Production.

```
Reimplemented in Wind, Wave, Tidal, and Solar.
```

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

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

# 4.13.3.2 computeProductionkW() [1/2]

# Reimplemented in Wind, Tidal, and Solar.

84 {return 0;}

## 4.13.3.3 computeProductionkW() [2/2]

### Reimplemented in Wave.

85 {return 0;}

#### 4.13.4 Member Data Documentation

## 4.13.4.1 resource\_key

int Renewable::resource\_key

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

### 4.13.4.2 type

RenewableType Renewable::type

The type (RenewableType) of the asset.

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

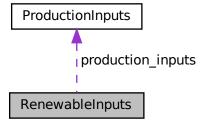
- · header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

# 4.14 RenewableInputs Struct Reference

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

#include <Renewable.h>

Collaboration diagram for RenewableInputs:



## **Public Attributes**

ProductionInputs production\_inputs
 An encapsulated ProductionInputs instance.

# 4.14.1 Detailed Description

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

## 4.14.2 Member Data Documentation

### 4.14.2.1 production\_inputs

ProductionInputs RenewableInputs::production\_inputs

An encapsulated ProductionInputs instance.

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

· header/Production/Renewable/Renewable.h

# 4.15 Resources Class Reference

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

```
#include <Resources.h>
```

#### **Public Member Functions**

· Resources (void)

Constructor for the Resources class.

void addResource1D (RenewableType, std::string, int, ElectricalLoad \*)

A method to add a 1D renewable resource time series to Resources. Checks against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

void addResource2D (RenewableType, std::string, int, ElectricalLoad \*)

A method to add a 2D renewable resource time series to Resources. Checks against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

void clear (void)

Method to clear all attributes of the Resources object.

∼Resources (void)

Destructor for the Resources class.

# **Public Attributes**

```
• std::map< int, std::vector< double >> resource map 1D
```

- std::map< int, std::string > path\_map\_1D
- std::map< int, std::vector< std::vector< double >>> resource\_map\_2D
- std::map< int, std::string > path map 2D

# 4.15.1 Detailed Description

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

### 4.15.2 Constructor & Destructor Documentation

### 4.15.2.1 Resources()

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

Constructor for the Resources class.

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

# 4.15.2.2 ∼Resources()

```
Resources::\simResources ( void )
```

Destructor for the Resources class.

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

# 4.15.3 Member Function Documentation

# 4.15.3.1 addResource1D()

A method to add a 1D renewable resource time series to Resources. Checks against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

# **Parameters**

renewable_type	The type of renewable resource being added to Resources.	
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time	
	series.	
resource_key	A key used to index into the Resources object, used to associate Renewable assets	
	with the corresponding resource.	
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.	

# 4.15.3.2 addResource2D()

A method to add a 2D renewable resource time series to Resources. Checks against ElectricalLoad to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

#### **Parameters**

renewable_type	The type of renewable resource being added to Resources.
path_2_resource_data	A string defining the path (either relative or absolute) to the given resource time
	series.
resource_key	A key used to index into the Resources object, used to associate Renewable assets with the corresponding resource.
	with the corresponding resource.
electrical_load_ptr	A pointer to the Model's ElectricalLoad object.

# 4.15.3.3 clear()

Method to clear all attributes of the Resources object.

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

# 4.15.4 Member Data Documentation

# 4.15.4.1 path\_map\_1D

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

4.16 Solar Class Reference 61

# 4.15.4.2 path\_map\_2D

std::map<int, std::string> Resources::path\_map\_2D

# 4.15.4.3 resource\_map\_1D

std::map<int, std::vector<double> > Resources::resource\_map\_1D

# 4.15.4.4 resource\_map\_2D

std::map<int, std::vector<std::vector<double> > Resources::resource\_map\_2D

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

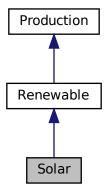
- · header/Resources.h
- source/Resources.cpp

# 4.16 Solar Class Reference

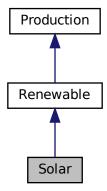
A derived class of the Renewable branch of Production which models solar production.

#include <Solar.h>

Inheritance diagram for Solar:



Collaboration diagram for Solar:



# **Public Member Functions**

• Solar (void)

Constructor (dummy) for the Solar class.

- Solar (int, SolarInputs)
- double computeProductionkW (int, double, double)

Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.

• double commit (int, double, double, double)

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

∼Solar (void)

Destructor for the Solar class.

# **Public Attributes**

· double derating

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

# 4.16.1 Detailed Description

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

# 4.16.2 Constructor & Destructor Documentation

4.16 Solar Class Reference 63

# 4.16.2.1 Solar() [1/2]

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

Constructor (dummy) for the Solar class.

Constructor (intended) for the Solar class.

#### **Parameters**

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

# 4.16.2.2 Solar() [2/2]

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

# 4.16.2.3 ∼Solar()

```
Solar::∼Solar (
void )
```

# Destructor for the Solar class.

4.16 Solar Class Reference 65

# 4.16.3 Member Function Documentation

# 4.16.3.1 commit()

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

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

#### **Parameters**

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

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

# 4.16.3.2 computeProductionkW()

Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.

ref: https://www.homerenergy.com/products/pro/docs/3.11/how\_homer\_calculates←
 \_the\_pv\_array\_power\_output.html

#### **Parameters**

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
solar_resource_kWm2	Solar resource (i.e. irradiance) [kW/m2].

#### Returns

The production [kW] of the solar PV array.

#### Reimplemented from Renewable.

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

# 4.16.4 Member Data Documentation

## 4.16.4.1 derating

```
double Solar::derating
```

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

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

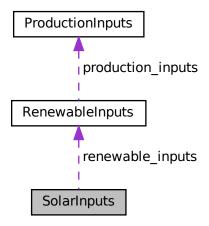
- · header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

# 4.17 SolarInputs Struct Reference

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

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

Collaboration diagram for SolarInputs:



# **Public Attributes**

• RenewableInputs renewable\_inputs

An encapsulated RenewableInputs instance.

• int resource\_key = 0

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

• double capital cost = -1

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

• double operation\_maintenance\_cost\_kWh = -1

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

• double derating = 0.8

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

# 4.17.1 Detailed Description

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

# 4.17.2 Member Data Documentation

### 4.17.2.1 capital\_cost

```
double SolarInputs::capital_cost = -1
```

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

# 4.17.2.2 derating

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

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

## 4.17.2.3 operation\_maintenance\_cost\_kWh

```
double SolarInputs::operation_maintenance_cost_kWh = -1
```

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

#### 4.17.2.4 renewable\_inputs

RenewableInputs SolarInputs::renewable\_inputs

An encapsulated RenewableInputs instance.

#### 4.17.2.5 resource\_key

```
int SolarInputs::resource_key = 0
```

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

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

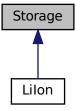
• header/Production/Renewable/Solar.h

# 4.18 Storage Class Reference

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

```
#include <Storage.h>
```

Inheritance diagram for Storage:



# **Public Member Functions**

• Storage (void)

Constructor for the Storage class.

virtual ∼Storage (void)

Destructor for the Storage class.

# 4.18.1 Detailed Description

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

# 4.18.2 Constructor & Destructor Documentation

# 4.18.2.1 Storage()

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

Constructor for the Storage class.

# 4.18.2.2 ∼Storage()

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

Destructor for the Storage class.

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

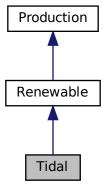
- header/Storage/Storage.h
- source/Storage/Storage.cpp

# 4.19 Tidal Class Reference

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

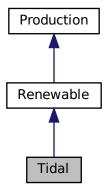
```
#include <Tidal.h>
```

Inheritance diagram for Tidal:



4.19 Tidal Class Reference 71

Collaboration diagram for Tidal:



#### **Public Member Functions**

• Tidal (void)

Constructor (dummy) for the Tidal class.

- Tidal (int, TidalInputs)
- double computeProductionkW (int, double, double)

Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.

• double commit (int, double, double, double)

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

∼Tidal (void)

Destructor for the Tidal class.

# **Public Attributes**

• double design\_speed\_ms

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

• TidalPowerProductionModel power\_model

The tidal power production model to be applied.

# 4.19.1 Detailed Description

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

# 4.19.2 Constructor & Destructor Documentation

# 4.19.2.1 Tidal() [1/2]

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

Constructor (dummy) for the Tidal class.

Constructor (intended) for the Tidal class.

#### **Parameters**

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

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

# 4.19.2.2 Tidal() [2/2]

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

# 4.19.2.3 ∼Tidal()

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

4.19 Tidal Class Reference 73

#### Destructor for the Tidal class.

# 4.19.3 Member Function Documentation

#### 4.19.3.1 commit()

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

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

#### **Parameters**

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

# Returns

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

# Reimplemented from Renewable.

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

# 4.19.3.2 computeProductionkW()

```
double dt_hrs,
double tidal_resource_ms ) [virtual]
```

Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.

#### **Parameters**

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
tidal_resource_ms	Tidal resource (i.e. tidal stream speed) [m/s].

#### Returns

The production [kW] of the tidal turbine.

#### Reimplemented from Renewable.

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

# 4.19.4 Member Data Documentation

# 4.19.4.1 design\_speed\_ms

```
double Tidal::design_speed_ms
```

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

### 4.19.4.2 power\_model

TidalPowerProductionModel Tidal::power\_model

The tidal power production model to be applied.

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

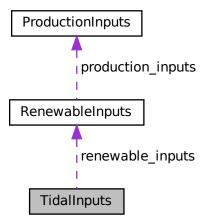
- · header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

# 4.20 TidalInputs Struct Reference

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

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

Collaboration diagram for TidalInputs:



# **Public Attributes**

• RenewableInputs renewable\_inputs

An encapsulated RenewableInputs instance.

• int resource\_key = 0

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

• double capital cost = -1

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

double operation\_maintenance\_cost\_kWh = -1

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

• double design\_speed\_ms = 3

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

TidalPowerProductionModel power model = TidalPowerProductionModel :: TIDAL POWER CUBIC

The tidal power production model to be applied.

# 4.20.1 Detailed Description

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

#### 4.20.2 Member Data Documentation

# 4.20.2.1 capital\_cost

```
double TidalInputs::capital_cost = -1
```

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

#### 4.20.2.2 design\_speed\_ms

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

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

#### 4.20.2.3 operation\_maintenance\_cost\_kWh

```
double TidalInputs::operation_maintenance_cost_kWh = -1
```

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

4.21 Wave Class Reference 77

#### 4.20.2.4 power\_model

TidalPowerProductionModel TidalInputs::power\_model = TidalPowerProductionModel :: TIDAL\_POWER\_CUBIC

The tidal power production model to be applied.

# 4.20.2.5 renewable\_inputs

RenewableInputs TidalInputs::renewable\_inputs

An encapsulated RenewableInputs instance.

# 4.20.2.6 resource\_key

```
int TidalInputs::resource_key = 0
```

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

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

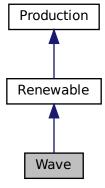
· header/Production/Renewable/Tidal.h

# 4.21 Wave Class Reference

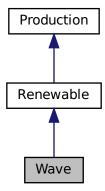
A derived class of the Renewable branch of Production which models wave production.

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

Inheritance diagram for Wave:



Collaboration diagram for Wave:



#### **Public Member Functions**

Wave (void)

Constructor (dummy) for the Wave class.

- Wave (int, WaveInputs)
- double computeProductionkW (int, double, double, double)

Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.

• double commit (int, double, double, double)

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

∼Wave (void)

Destructor for the Wave class.

# **Public Attributes**

• double design\_significant\_wave\_height\_m

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

• double design\_energy\_period\_s

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

· WavePowerProductionModel power model

The wave power production model to be applied.

# 4.21.1 Detailed Description

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

#### 4.21.2 Constructor & Destructor Documentation

#### 4.21.2.1 Wave() [1/2]

Constructor (dummy) for the Wave class.

Constructor (intended) for the Wave class.

#### **Parameters**

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

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

#### 4.21.2.2 Wave() [2/2]

```
Wave::Wave (
               int n_points,
               WaveInputs wave_inputs )
258 Renewable(n_points, wave_inputs.renewable_inputs)
259 {
        // 1. check inputs
this->__checkInputs(wave_inputs);
260
261
262
263
         // 2. set attributes
        this->type = RenewableType :: WAVE;
this->type_str = "WAVE";
264
265
266
267
        this->resource_key = wave_inputs.resource_key;
268
269
        this->design_significant_wave_height_m =
270
             wave_inputs.design_significant_wave_height_m;
271
        this->design_energy_period_s = wave_inputs.design_energy_period_s;
2.72
273
        this->power_model = wave_inputs.power_model;
274
275
        if (wave_inputs.capital_cost < 0) {</pre>
276
             this->capital_cost = this->__getGenericCapitalCost();
277
278
279
        if (wave_inputs.operation_maintenance_cost_kWh < 0) {</pre>
280
            this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
281
282
283
        if (this->is_sunk) {
             this->capital_cost_vec[0] = this->capital_cost;
284
285
286
287
        // 3. construction print
288
        if (this->print_flag) {
            std::cout « "Wave object constructed at " « this « std::endl;
289
290
291
292
        return;
293 }
        /* Renewable() */
```

### 4.21.2.3 ∼Wave()

```
Wave::\simWave ( void )
```

#### Destructor for the Wave class.

# 4.21.3 Member Function Documentation

# 4.21.3.1 commit()

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

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

## **Parameters**

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

# Returns

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

#### Reimplemented from Renewable.

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

4.21 Wave Class Reference 81

# 4.21.3.2 computeProductionkW()

```
double Wave::computeProductionkW (
    int timestep,
    double dt_hrs,
    double significant_wave_height_m,
    double energy_period_s ) [virtual]
```

Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.

#### **Parameters**

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
signficiant_wave_height↔	The significant wave height (wave statistic) [m].
_m	
energy_period_s	The energy period (wave statistic) [s].

#### Returns

The production [kW] of the wave turbine.

#### Reimplemented from Renewable.

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

# 4.21.4 Member Data Documentation

# 4.21.4.1 design\_energy\_period\_s

```
double Wave::design_energy_period_s
```

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

# 4.21.4.2 design\_significant\_wave\_height\_m

```
double Wave::design_significant_wave_height_m
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

## 4.21.4.3 power\_model

WavePowerProductionModel Wave::power\_model

The wave power production model to be applied.

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

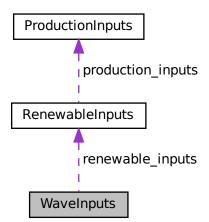
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

# 4.22 WaveInputs Struct Reference

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

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

Collaboration diagram for WaveInputs:



#### **Public Attributes**

· RenewableInputs renewable\_inputs

An encapsulated RenewableInputs instance.

• int resource\_key = 0

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

double capital\_cost = -1

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

double operation\_maintenance\_cost\_kWh = -1

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

• double design\_significant\_wave\_height\_m = 3

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

double design energy period s = 10

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

• WavePowerProductionModel power\_model = WavePowerProductionModel :: WAVE\_POWER\_PARABOLOID

The wave power production model to be applied.

# 4.22.1 Detailed Description

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

#### 4.22.2 Member Data Documentation

#### 4.22.2.1 capital\_cost

```
double WaveInputs::capital_cost = -1
```

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

#### 4.22.2.2 design\_energy\_period\_s

```
double WaveInputs::design_energy_period_s = 10
```

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

#### 4.22.2.3 design\_significant\_wave\_height\_m

```
double WaveInputs::design_significant_wave_height_m = 3
```

The significant wave height [m] at which the wave energy converter achieves its rated capacity.

#### 4.22.2.4 operation\_maintenance\_cost\_kWh

```
double WaveInputs::operation_maintenance_cost_kWh = -1
```

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

## 4.22.2.5 power\_model

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

The wave power production model to be applied.

# 4.22.2.6 renewable\_inputs

RenewableInputs WaveInputs::renewable\_inputs

An encapsulated RenewableInputs instance.

# 4.22.2.7 resource\_key

```
int WaveInputs::resource_key = 0
```

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

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

• header/Production/Renewable/Wave.h

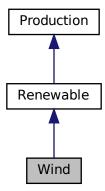
4.23 Wind Class Reference 85

# 4.23 Wind Class Reference

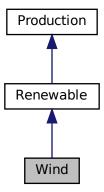
A derived class of the Renewable branch of Production which models wind production.

#include <Wind.h>

Inheritance diagram for Wind:



Collaboration diagram for Wind:



# **Public Member Functions**

• Wind (void)

Constructor (dummy) for the Wind class.

• Wind (int, WindInputs)

double computeProductionkW (int, double, double)

Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.

• double commit (int, double, double, double)

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

∼Wind (void)

Destructor for the Wind class.

# **Public Attributes**

• double design\_speed\_ms

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

• WindPowerProductionModel power\_model

The wind power production model to be applied.

# 4.23.1 Detailed Description

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

#### 4.23.2 Constructor & Destructor Documentation

#### 4.23.2.1 Wind() [1/2]

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

Constructor (dummy) for the Wind class.

Constructor (intended) for the Wind class.

## **Parameters**

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

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

# 4.23.2.2 Wind() [2/2]

4.23 Wind Class Reference 87

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

# 4.23.2.3 ∼Wind()

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

# Destructor for the Wind class.

# 4.23.3 Member Function Documentation

## 4.23.3.1 commit()

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

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

#### **Parameters**

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

# Returns

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

# Reimplemented from Renewable.

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

# 4.23.3.2 computeProductionkW()

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.

4.23 Wind Class Reference 89

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

# 4.23.4 Member Data Documentation

# 4.23.4.1 design\_speed\_ms

```
double Wind::design_speed_ms
```

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

# 4.23.4.2 power\_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

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

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

# 4.24 WindInputs Struct Reference

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

```
#include <Wind.h>
```

Collaboration diagram for WindInputs:



# **Public Attributes**

· RenewableInputs renewable\_inputs

An encapsulated RenewableInputs instance.

• int resource\_key = 0

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

• double capital\_cost = -1

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

• double operation\_maintenance\_cost\_kWh = -1

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

• double design\_speed\_ms = 8

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

• WindPowerProductionModel power\_model = WindPowerProductionModel :: WIND\_POWER\_EXPONENTIAL The wind power production model to be applied.

# 4.24.1 Detailed Description

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

#### 4.24.2 Member Data Documentation

#### 4.24.2.1 capital cost

```
double WindInputs::capital_cost = -1
```

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

## 4.24.2.2 design\_speed\_ms

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

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

# 4.24.2.3 operation\_maintenance\_cost\_kWh

```
\label{lower_double_windInputs::operation_maintenance_cost_kWh = -1} \\
```

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

# 4.24.2.4 power\_model

WindPowerProductionModel WindInputs::power\_model = WindPowerProductionModel :: WIND\_POWER\_EXPONENTIAL

The wind power production model to be applied.

# 4.24.2.5 renewable\_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

#### 4.24.2.6 resource key

```
int WindInputs::resource_key = 0
```

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

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

· header/Production/Renewable/Wind.h

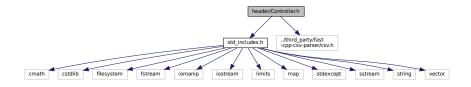
# **Chapter 5**

# **File Documentation**

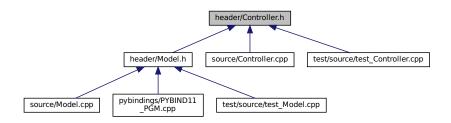
# 5.1 header/Controller.h File Reference

Header file the Controller class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for Controller.h:
```



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



# **Classes**

class Controller

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

94 File Documentation

# **Enumerations**

• enum ControlMode { LOAD\_FOLLOWING , CYCLE\_CHARGING , N\_CONTROL\_MODES }

An enumeration of the types of control modes supported by PGMcpp.

# 5.1.1 Detailed Description

Header file the Controller class.

# **5.1.2 Enumeration Type Documentation**

#### 5.1.2.1 ControlMode

```
enum ControlMode
```

An enumeration of the types of control modes supported by PGMcpp.

#### Enumerator

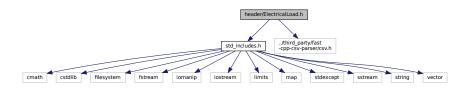
LOAD_FOLLOWING	Load following control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
CYCLE_CHARGING	Cycle charging control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
N_CONTROL_MODES	A simple hack to get the number of elements in ControlMode.

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

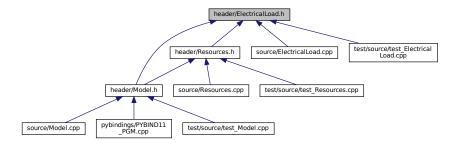
# 5.2 header/ElectricalLoad.h File Reference

Header file the ElectricalLoad class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for ElectricalLoad.h:
```



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



#### **Classes**

· class ElectricalLoad

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

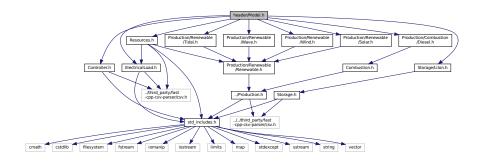
# 5.2.1 Detailed Description

Header file the ElectricalLoad class.

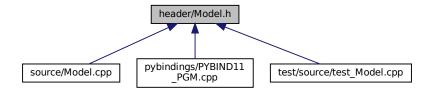
## 5.3 header/Model.h File Reference

Header file the Model class.

```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
#include "Production/Renewable/Wave.h"
#include "Production/Renewable/Wind.h"
#include "Storage/LiIon.h"
Include dependency graph for Model.h:
```



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



#### **Classes**

struct ModelInputs

A structure which bundles the necessary inputs for the <u>Model</u> constructor. Provides default values for every necessary input (except path\_2\_electrical\_load\_time\_series, for which a valid input must be provided).

class Model

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

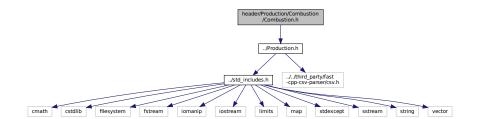
## 5.3.1 Detailed Description

Header file the Model class.

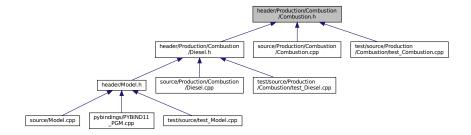
# 5.4 header/Production/Combustion/Combustion.h File Reference

Header file the Combustion class.

#include "../Production.h"
Include dependency graph for Combustion.h:



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



## **Classes**

struct CombustionInputs

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

• struct Emissions

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

• class Combustion

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

#### **Enumerations**

enum CombustionType { DIESEL , N\_COMBUSTION\_TYPES }

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

## 5.4.1 Detailed Description

Header file the Combustion class.

# 5.4.2 Enumeration Type Documentation

#### 5.4.2.1 CombustionType

enum CombustionType

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

#### Enumerator

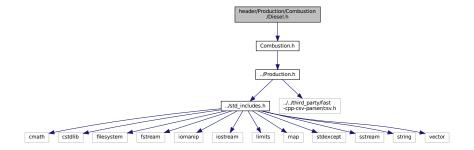
DIESEL	A diesel generator.
N_COMBUSTION_TYPES	A simple hack to get the number of elements in CombustionType.

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

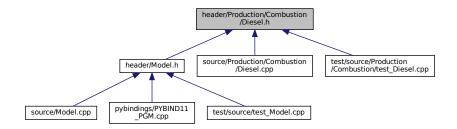
# 5.5 header/Production/Combustion/Diesel.h File Reference

Header file the Diesel class.

```
#include "Combustion.h"
Include dependency graph for Diesel.h:
```



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



#### **Classes**

struct DieselInputs

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

· class Diesel

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

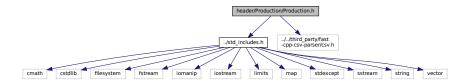
# 5.5.1 Detailed Description

Header file the Diesel class.

## 5.6 header/Production/Production.h File Reference

Header file the Production class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for Production.h:
```



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



#### **Classes**

• struct ProductionInputs

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

class Production

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

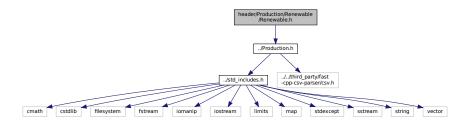
#### 5.6.1 Detailed Description

Header file the Production class.

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

Header file the Renewable class.

#include "../Production.h"
Include dependency graph for Renewable.h:



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



#### **Classes**

• struct RenewableInputs

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

class Renewable

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

## **Enumerations**

enum RenewableType {
 SOLAR, TIDAL, WAVE, WIND,
 N\_RENEWABLE\_TYPES}

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

# 5.7.1 Detailed Description

Header file the Renewable class.

# 5.7.2 Enumeration Type Documentation

#### 5.7.2.1 RenewableType

enum RenewableType

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

#### Enumerator

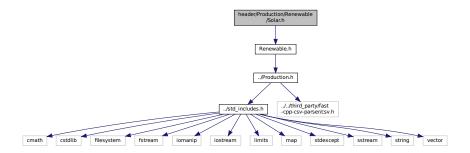
SOLAR	A solar photovoltaic (PV) array.
TIDAL	A tidal stream turbine (or tidal energy converter, TEC)
WAVE	A wave energy converter (WEC)
WIND	A wind turbine.
N_RENEWABLE_TYPES	A simple hack to get the number of elements in RenewableType.

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

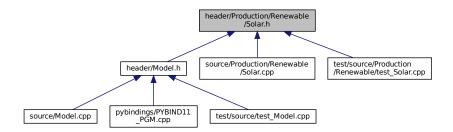
## 5.8 header/Production/Renewable/Solar.h File Reference

Header file the Solar class.

#include "Renewable.h"
Include dependency graph for Solar.h:



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



#### **Classes**

struct SolarInputs

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

· class Solar

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

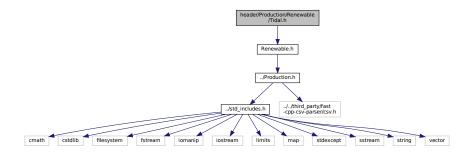
## 5.8.1 Detailed Description

Header file the Solar class.

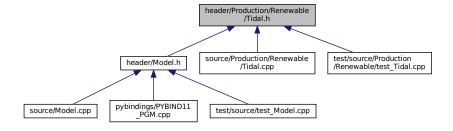
# 5.9 header/Production/Renewable/Tidal.h File Reference

Header file the Tidal class.

#include "Renewable.h"
Include dependency graph for Tidal.h:



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



#### **Classes**

struct TidalInputs

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

· class Tidal

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

## **Enumerations**

 enum TidalPowerProductionModel { TIDAL\_POWER\_CUBIC , TIDAL\_POWER\_EXPONENTIAL , TIDAL POWER LOOKUP, N TIDAL POWER PRODUCTION MODELS }

# 5.9.1 Detailed Description

Header file the Tidal class.

# 5.9.2 Enumeration Type Documentation

#### 5.9.2.1 TidalPowerProductionModel

enum TidalPowerProductionModel

#### Enumerator

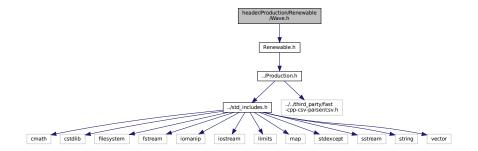
TIDAL_POWER_CUBIC	A cubic power production model.
TIDAL_POWER_EXPONENTIAL	An exponential power production model.
TIDAL_POWER_LOOKUP	Lookup from a given set of power curve data.
N_TIDAL_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in
	TidalPowerProductionModel.

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

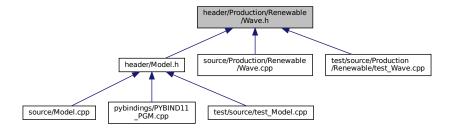
# 5.10 header/Production/Renewable/Wave.h File Reference

Header file the Wave class.

```
#include "Renewable.h"
Include dependency graph for Wave.h:
```



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



#### **Classes**

struct WaveInputs

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

· class Wave

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

#### **Enumerations**

enum WavePowerProductionModel { WAVE\_POWER\_GAUSSIAN , WAVE\_POWER\_PARABOLOID , WAVE\_POWER\_LOOKUP, N\_WAVE\_POWER\_PRODUCTION\_MODELS }

# 5.10.1 Detailed Description

Header file the Wave class.

# 5.10.2 Enumeration Type Documentation

## 5.10.2.1 WavePowerProductionModel

enum WavePowerProductionModel

## Enumerator

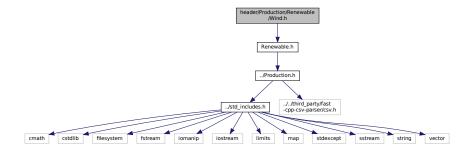
WAVE_POWER_GAUSSIAN	A Gaussian power production model.
WAVE_POWER_PARABOLOID	A paraboloid power production model.
WAVE_POWER_LOOKUP	Lookup from a given performance matrix.
N_WAVE_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in
	WavePowerProductionModel.

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

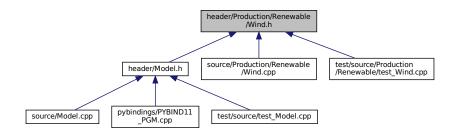
## 5.11 header/Production/Renewable/Wind.h File Reference

Header file the Wind class.

```
#include "Renewable.h"
Include dependency graph for Wind.h:
```



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



#### Classes

struct WindInputs

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

· class Wind

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

#### **Enumerations**

enum WindPowerProductionModel { WIND\_POWER\_EXPONENTIAL , WIND\_POWER\_LOOKUP , N\_WIND\_POWER\_PRODUCTION\_MODELS }

# 5.11.1 Detailed Description

Header file the Wind class.

# 5.11.2 Enumeration Type Documentation

#### 5.11.2.1 WindPowerProductionModel

enum WindPowerProductionModel

#### Enumerator

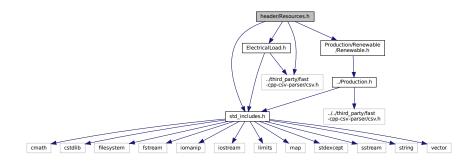
WIND_POWER_EXPONENTIAL	An exponential power production model.
WIND_POWER_LOOKUP	Lookup from a given set of power curve data.
N_WIND_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in
	WindPowerProductionModel.

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

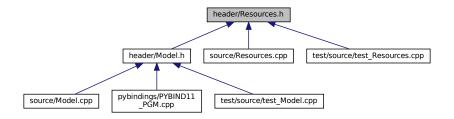
# 5.12 header/Resources.h File Reference

Header file the Resources class.

```
#include "std_includes.h"
#include "ElectricalLoad.h"
#include "Production/Renewable/Renewable.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for Resources.h:
```



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



#### **Classes**

class Resources

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

## 5.12.1 Detailed Description

Header file the Resources class.

# 5.13 header/std\_includes.h File Reference

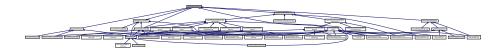
Header file which simply batches together the usual, standard includes.

```
#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <ioomanip>
#include <iiostream>
#include <limits>
#include <map>
#include <stdexcept>
#include <sstream>
#include <sstream>
#include <string>
#include <vector>
```

Include dependency graph for std\_includes.h:



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



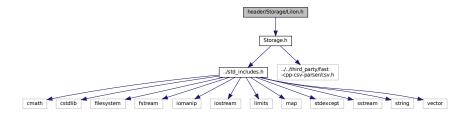
# 5.13.1 Detailed Description

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

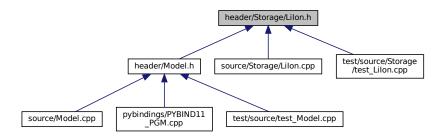
# 5.14 header/Storage/Lilon.h File Reference

Header file the Lilon class.

#include "Storage.h"
Include dependency graph for Lilon.h:



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



## Classes

· class Lilon

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

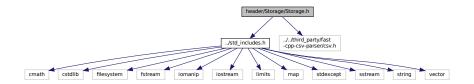
# 5.14.1 Detailed Description

Header file the Lilon class.

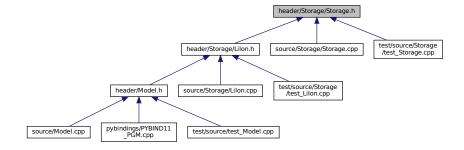
# 5.15 header/Storage/Storage.h File Reference

Header file the Storage class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for Storage.h:
```



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



#### Classes

• class Storage

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

# 5.15.1 Detailed Description

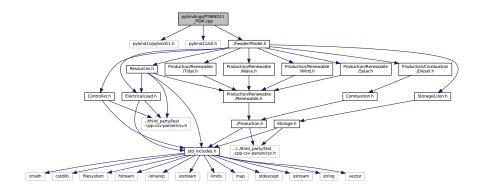
Header file the Storage class.

# 5.16 pybindings/PYBIND11\_PGM.cpp File Reference

Python 3 bindings file for PGMcpp.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
```

#include "../header/Model.h"
Include dependency graph for PYBIND11\_PGM.cpp:



## **Functions**

• PYBIND11\_MODULE (PGMcpp, m)

## 5.16.1 Detailed Description

Python 3 bindings file for PGMcpp.

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

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

#### 5.16.2 Function Documentation

#### 5.16.2.1 PYBIND11\_MODULE()

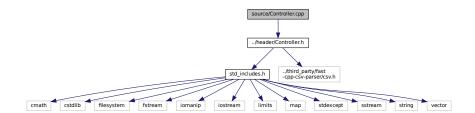
```
PYBIND11_MODULE (
                  PGMcpp ,
30
31
32 // =
         33 /*
34 pybindll::class_<Controller>(m, "Controller")
         .def(pybind11::init());
36 */
37
    // ======= END Controller ======= //
38
39
41 //
                ====== ElectricalLoad ======= //
43 pybindl1::class_<ElectricalLoad>(m, "ElectricalLoad")
44 .def_readwrite("n_points", &ElectricalLoad::n_points)
45 .def_readwrite("max_load_kW", &ElectricalLoad::max_load_kW)
46 .def_readwrite("mean_load_kW", &ElectricalLoad::mean_load_kW)
         .def_readwrite("min_load_kW", &ElectricalLoad::min_load_kW)
```

```
48
       .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)
50
51
       .def(pybind11::init<std::string>());
52
53 */
54 // ========= END ElectricalLoad ======= //
55
56
57
58 // ======== Model ====== //
59 /*
60 pybind11::class_<Model>(m, "Model")
62
          pybind11::init<
63
               ElectricalLoad*,
64
               RenewableResources*
          >()
65
66
68 // ======= END Model ====== //
69
70
72 // ======== RenewableResources ======= //
74 pybind11::class_<RenewableResources>(m, "RenewableResources")
75
      .def(pybind11::init());
76
77
       .def(pybind11::init<>());
78
80 // ====== END RenewableResources ====== //
82 }
      /* PYBIND11_MODULE() */
```

# 5.17 source/Controller.cpp File Reference

Implementation file for the Controller class.

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



## 5.17.1 Detailed Description

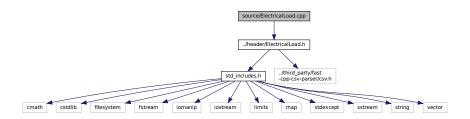
Implementation file for the Controller class.

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

# 5.18 source/ElectricalLoad.cpp File Reference

Implementation file for the ElectricalLoad class.

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



## 5.18.1 Detailed Description

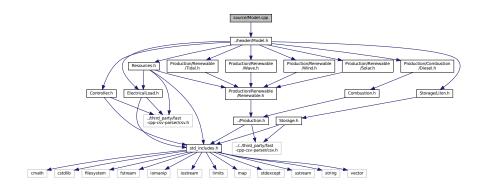
Implementation file for the ElectricalLoad class.

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

# 5.19 source/Model.cpp File Reference

Implementation file for the Model class.

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



## 5.19.1 Detailed Description

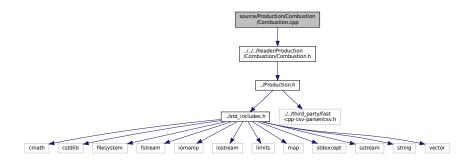
Implementation file for the Model class.

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

# 5.20 source/Production/Combustion/Combustion.cpp File Reference

Implementation file for the Combustion class.

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



## 5.20.1 Detailed Description

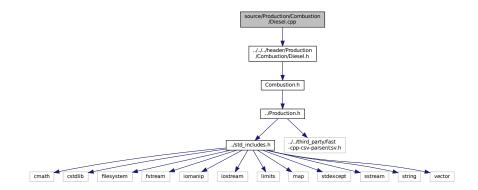
Implementation file for the Combustion class.

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

# 5.21 source/Production/Combustion/Diesel.cpp File Reference

Implementation file for the Diesel class.

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



# 5.21.1 Detailed Description

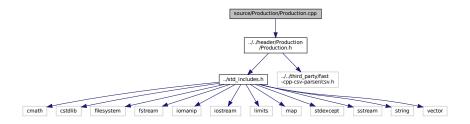
Implementation file for the Diesel class.

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

# 5.22 source/Production/Production.cpp File Reference

Implementation file for the Production class.

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



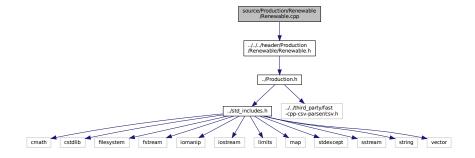
## 5.22.1 Detailed Description

Implementation file for the Production class.

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

# 5.23 source/Production/Renewable/Renewable.cpp File Reference

Implementation file for the Renewable class.



## 5.23.1 Detailed Description

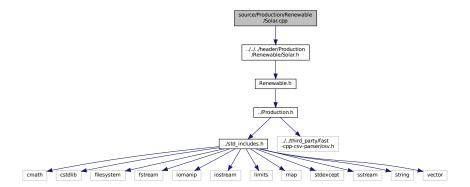
Implementation file for the Renewable class.

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

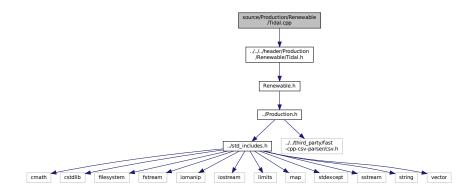
Implementation file for the Solar class.

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

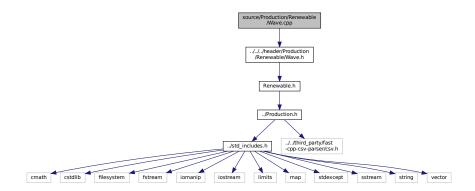
Implementation file for the Tidal class.

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

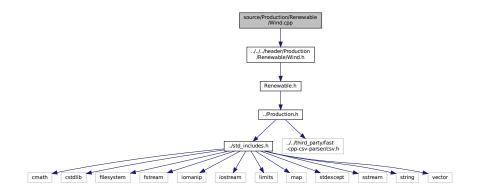
Implementation file for the Wave class.

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

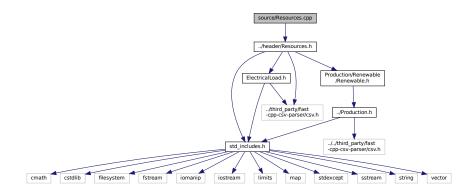
Implementation file for the Wind class.

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

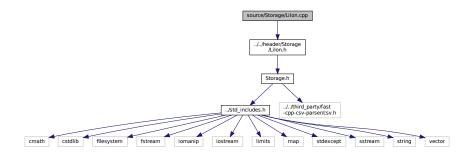
Implementation file for the Resources class.

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

# 5.29 source/Storage/Lilon.cpp File Reference

Implementation file for the Lilon class.

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



## 5.29.1 Detailed Description

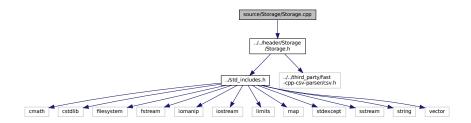
Implementation file for the Lilon class.

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

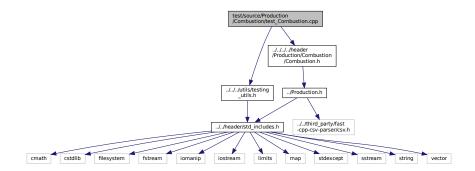
Implementation file for the Storage class.

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();
29
30
    #endif /* _WIN32 */
31
    printGold("\tTesting Production <-- Combustion");</pre>
     srand(time(NULL));
35
36
37 try {
38
39 // ----- CONSTRUCTION -----//
40
41 CombustionInputs combustion_inputs;
43 Combustion test_combustion(8760, combustion_inputs);
44
45 // ====== END CONSTRUCTION =========
48
49 // ====== ATTRIBUTES ========= //
50
     not combustion_inputs.production_inputs.print_flag,
     __LINE_
54
55);
56
57 testFloatEquals(
      test_combustion.fuel_consumption_vec_L.size(),
59
      8760,
60
     ___FILE___,
      __LINE__
61
62);
63
64 testFloatEquals(
      test_combustion.fuel_cost_vec.size(),
66
      8760,
67
     ___FILE___,
      __LINE_
68
69);
71 testFloatEquals(
      test_combustion.CO2_emissions_vec_kg.size(),
73
      8760,
74
      ___FILE_
      __LINE_
75
76);
```

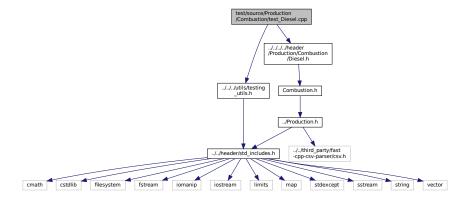
```
78 testFloatEquals(
       test_combustion.CO_emissions_vec_kg.size(),
80
       8760,
81
      ___FILE_
       __LINE_
82
83);
85 testFloatEquals(
       test_combustion.NOx_emissions_vec_kg.size(),
87
       8760,
      ___FILE_
88
       __LINE__
89
90);
92 testFloatEquals(
       {\tt test\_combustion.SOx\_emissions\_vec\_kg.size(),}
94
       8760.
       __FILE_
95
       __LINE__
96
97);
98
99 testFloatEquals(
100
        test_combustion.CH4_emissions_vec_kg.size(),
        8760.
101
        __FILE_
102
103
        __LINE__
104);
105
106 testFloatEquals(
107
        test_combustion.PM_emissions_vec_kg.size(),
108
        8760,
        __FILE_
109
110
111 );
112
113 // ====== END ATTRIBUTES ============ //
114
115 } /* try */
116
117
118 catch (...) {
119
120
       printGold(" .....");
printRed("FAIL");
121
122
123
       std::cout « std::endl;
124
125 }
126
127
128 printGold(" .....");
129 printGreen("PASS");
130 std::cout « std::endl;
131 return 0;
132
133 } /* main() */
```

# 5.32 test/source/Production/Combustion/test\_Diesel.cpp File Reference

Testing suite for Diesel class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Combustion/Diesel.h"
```

Include dependency graph for test\_Diesel.cpp:



#### **Functions**

• int main (int argc, char \*\*argv)

# 5.32.1 Detailed Description

Testing suite for Diesel class.

A suite of tests for the Diesel class.

#### 5.32.2 Function Documentation

#### 5.32.2.1 main()

```
int main (
             int argc,
             char ** argv )
27 {
28
      #ifdef _WIN32
         activateVirtualTerminal();
30
      #endif /* _WIN32 */
31
32
33
     printGold("\tTesting Production <-- Combustion <-- Diesel");</pre>
34
     srand(time(NULL));
35
37
      Combustion* test_diesel_ptr;
38
39 try {
40
41 // ----- CONSTRUCTION -----//
43 bool error_flag = true;
44
45 try
      DieselInputs bad_diesel_inputs;
46
47
      bad_diesel_inputs.fuel_cost_L = -1;
```

```
49
       Diesel bad_diesel(8760, bad_diesel_inputs);
51
       error_flag = false;
52 } catch (...) {
53  // Task failed successfully! =P
54 }
55 if (not error_flag) {
      expectedErrorNotDetected(__FILE__, __LINE__);
57 }
58
59 DieselInputs diesel_inputs;
60
61 test_diesel_ptr = new Diesel(8760, diesel_inputs);
63
64 // ----- END CONSTRUCTION ------//
65
66
68 // ----- ATTRIBUTES ------//
70 testTruth(
71
      not diesel_inputs.combustion_inputs.production_inputs.print_flag,
      ___FILE___,
72
73
       __LINE__
74);
75
76 testFloatEquals(
77
      test_diesel_ptr->type,
      CombustionType :: DIESEL,
78
      __FILE__,
79
80
       __LINE__
81 );
82
83 testTruth(
      test_diesel_ptr->type_str == "DIESEL",
84
       ___FILE___,
85
      __LINE__
86
87);
88
89 testFloatEquals(
      test_diesel_ptr->linear_fuel_slope_LkWh, 0.265675,
90
91
      __FILE__,
92
      __LINE__
93
94);
95
96 testFloatEquals(
      test_diesel_ptr->linear_fuel_intercept_LkWh,
97
98
       0.026676,
      __FILE__,
99
       __LINE__
100
101 );
102
103 testFloatEquals(
       test_diesel_ptr->capital_cost,
104
105
       94125.375446,
       __FILE__,
106
107
       __LINE__
108);
109
110 testFloatEquals(
111
       test_diesel_ptr->operation_maintenance_cost_kWh,
112
       0.069905,
       __FILE__,
113
       __LINE__
114
115 );
116
117 testFloatEquals(
        ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
118
119
       0.2,
       ___FILE___,
120
       __LINE__
121
122 );
123
124 testFloatEquals(
125
        ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
126
       __FILE__,
127
128
       __LINE__
129);
130
131 testFloatEquals(
132
       test_diesel_ptr->replace_running_hrs,
133
       30000,
       ___FILE_
134
        __LINE
135
```

```
136);
137
138 // ====== END ATTRIBUTES ======== //
139
140
141
142 // ====== METHODS =========
143
144 // test capacity constraint
145 testFloatEquals(
        test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
146
        {\tt test\_diesel\_ptr->} {\tt capacity\_kW},
147
        ___FILE___,
148
149
150);
151
152 // test minimum load ratio constraint
153 testFloatEquals(
154
        test_diesel_ptr->requestProductionkW(
155
             Ο,
156
157
             0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
158
                 test_diesel_ptr->capacity_kW
159
160
        ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
161
        __FILE__,
162
        __LINE__
163);
164
165 // test commit()
166 std::vector<double> dt vec hrs (48, 1);
167
168 std::vector<double> load_vec_kW = {
169
       1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
170
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
171
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
172
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
173 };
174
175 std::vector<bool> expected_is_running_vec = {
        1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
176
177
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
178
179
180 };
181
182 double load_kW = 0;
183 double production_kW = 0;
184 double roll = 0;
185
186 for (int i = 0; i < 48; i++) {
187
        roll = (double)rand() / RAND_MAX;
188
        if (roll >= 0.95) {
    roll = 1.25;
189
190
        }
191
192
193
        load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
194
        load_kW = load_vec_kW[i];
195
196
        production_kW = test_diesel_ptr->requestProductionkW(
197
198
             dt_vec_hrs[i],
199
             load_kW
200
        );
201
        load_kW = test_diesel_ptr->commit(
202
203
             i.
204
             dt_vec_hrs[i],
205
            production_kW,
206
             load_kW
207
208
        // load_kW <= load_vec_kW (i.e., after vs before)</pre>
209
        testLessThanOrEqualTo(
210
211
             load_kW,
212
            load_vec_kW[i],
213
            ___FILE___,
214
             __LINE__
215
        ):
216
217
        // production = dispatch + storage + curtailment
218
        testFloatEquals(
219
            test_diesel_ptr->production_vec_kW[i] -
220
             test_diesel_ptr->dispatch_vec_kW[i] -
            test_diesel_ptr->storage_vec_kW[i] -
test_diesel_ptr->curtailment_vec_kW[i],
221
222
```

```
223
             Ο,
             __FILE__,
224
225
             __LINE__
226
        );
2.2.7
228
        // capacity constraint
        if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
229
230
             testFloatEquals(
231
                 test_diesel_ptr->production_vec_kW[i],
232
                 test_diesel_ptr->capacity_kW,
233
                 ___FILE___,
234
                  LINE
235
            );
236
237
238
        // minimum load ratio constraint
239
        else if (
240
            test_diesel_ptr->is_running and
             test_diesel_ptr->production_vec_kW[i] > 0 and
241
242
             load_vec_kW[i] <
243
             ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
244
2.45
             testFloatEquals(
                 test_diesel_ptr->production_vec_kW[i],
((Diesel*)test_diesel_ptr)->minimum_load_ratio *
246
247
248
                     test_diesel_ptr->capacity_kW,
                 ___FILE___,
249
250
                 __LINE__
2.51
             );
252
        }
253
254
         // minimum runtime constraint
255
        testFloatEquals(
256
             test_diesel_ptr->is_running_vec[i],
257
             expected_is_running_vec[i],
258
             ___FILE___,
             __LINE__
259
260
        );
261
262
         // O&M, fuel consumption, and emissions > 0 whenever diesel is running
263
         if (test_diesel_ptr->is_running) {
             testGreaterThan(
2.64
265
                 test_diesel_ptr->operation_maintenance_cost_vec[i],
266
                 Ο,
                 ___FILE___,
267
268
                 __LINE__
269
             );
270
271
             testGreaterThan(
272
                 test_diesel_ptr->fuel_consumption_vec_L[i],
273
                 Ο,
274
                 __FILE__,
275
                 __LINE__
276
            );
277
278
             testGreaterThan(
279
                 test_diesel_ptr->fuel_cost_vec[i],
280
281
                 ___FILE___,
282
                 __LINE__
283
             );
284
285
             testGreaterThan(
286
                 test_diesel_ptr->CO2_emissions_vec_kg[i],
287
                 Ο,
                 ___FILE___,
288
289
                 __LINE__
290
             );
291
292
             testGreaterThan(
293
                 test_diesel_ptr->CO_emissions_vec_kg[i],
294
                 Ο,
                 __FILE___,
295
296
                 __LINE__
297
             );
298
299
             testGreaterThan(
300
                 test_diesel_ptr->NOx_emissions_vec_kg[i],
                 0,
__FILE_
301
302
303
                 __LINE__
304
             );
305
306
             testGreaterThan(
307
                 test_diesel_ptr->S0x_emissions_vec_kg[i],
308
                 Ο,
                 __FILE__,
309
```

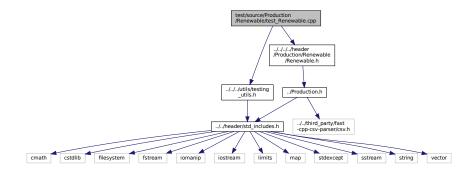
```
310
                __LINE__
311
312
313
            testGreaterThan(
314
                test_diesel_ptr->CH4_emissions_vec_kg[i],
315
316
317
                __LINE__
318
            );
319
320
            testGreaterThan(
                test_diesel_ptr->PM_emissions_vec_kg[i],
321
322
                0,
                ___FILE___,
323
324
                __LINE__
325
            );
326
327
328
        // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
329
330
            testFloatEquals(
331
                test_diesel_ptr->operation_maintenance_cost_vec[i],
332
                Ο,
                ___FILE___,
333
334
                __LINE_
335
            );
336
337
            testFloatEquals(
338
                test_diesel_ptr->fuel_consumption_vec_L[i],
339
                Ο,
                __FILE__,
340
341
                __LINE_
342
343
344
            testFloatEquals(
                test_diesel_ptr->fuel_cost_vec[i],
345
346
                ___FILE___,
347
348
                __LINE__
349
350
351
            testFloatEquals(
                test_diesel_ptr->CO2_emissions_vec_kg[i],
352
353
                Ο,
                ___FILE___,
354
355
                __LINE__
356
357
            testFloatEquals(
358
359
                test_diesel_ptr->CO_emissions_vec_kg[i],
360
                Ο,
361
                __FILE__,
362
                __LINE__
363
            );
364
            testFloatEquals(
365
366
                test_diesel_ptr->NOx_emissions_vec_kg[i],
367
368
                ___FILE___,
369
                __LINE__
370
            );
371
372
            testFloatEquals(
373
                test_diesel_ptr->SOx_emissions_vec_kg[i],
374
                __FILE__,
375
376
                __LINE__
377
            );
378
            testFloatEquals(
380
                test_diesel_ptr->CH4_emissions_vec_kg[i],
381
                Ο,
                ___FILE___,
382
383
                __LINE__
384
            );
385
386
            testFloatEquals(
387
                test_diesel_ptr->PM_emissions_vec_kg[i],
                0,
__FILE__,
388
389
390
                __LINE__
391
            );
392
393 }
394
395 // ====== END METHODS ======= //
396
```

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

# 5.33 test/source/Production/Renewable/test\_Renewable.cpp File Reference

Testing suite for Renewable class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Renewable/Renewable.h"
Include dependency graph for test Renewable.cpp:
```



#### **Functions**

• int main (int argc, char \*\*argv)

## 5.33.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

#### 5.33.2 Function Documentation

#### 5.33.2.1 main()

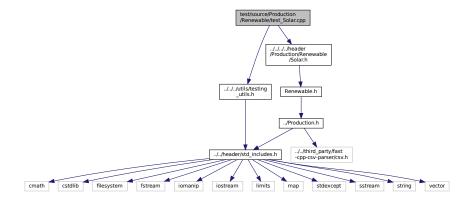
```
int main (
           int argc,
           char ** argv )
27 {
     #ifdef _WIN32
2.8
        activateVirtualTerminal();
29
30
    #endif /* _WIN32 */
31
    printGold("\tTesting Production <-- Renewable");</pre>
34
    srand(time(NULL));
35
36
37 try {
39 // ----- CONSTRUCTION -----//
40
41 RenewableInputs renewable_inputs;
43 Renewable test_renewable(8760, renewable_inputs);
45 // ====== END CONSTRUCTION =========== //
46
47
48
49 // ====== ATTRIBUTES ========== //
    not renewable_inputs.production_inputs.print_flag,
53
     ___FILE___,
54
     __LINE_
55);
57 // ====== END ATTRIBUTES =========== //
59 } /* try */
60
61
62 catch (...) {
     //...
    printGold(" .....");
printRed("FAIL");
std::cout « std::endl;
65
66
67
     throw;
68
69 }
70
72 printGold(" .....");
73 printGreen("PASS");
74 std::cout « std::endl;
75 return 0;
76 } /* main() */
```

# 5.34 test/source/Production/Renewable/test\_Solar.cpp File Reference

Testing suite for Solar class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Renewable/Solar.h"
```

Include dependency graph for test\_Solar.cpp:



## **Functions**

• int main (int argc, char \*\*argv)

# 5.34.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar class.

#### 5.34.2 Function Documentation

#### 5.34.2.1 main()

```
int main (
             int argc,
             char ** argv )
27 {
28
      #ifdef _WIN32
         activateVirtualTerminal();
30
      #endif /* _WIN32 */
31
32
33
      printGold("\tTesting Production <-- Renewable <-- Solar");</pre>
34
      srand(time(NULL));
35
      Renewable* test_solar_ptr;
37
38 try {
39
40 // ----- CONSTRUCTION -----//
42 bool error_flag = true;
43
44 try {
      SolarInputs bad_solar_inputs;
45
46
      bad_solar_inputs.derating = -1;
47
48
      Solar bad_solar(8760, bad_solar_inputs);
```

```
49
       error_flag = false;
51 } catch (...)
     // Task failed successfully! =P
52
53 }
54 if (not error flag) {
      expectedErrorNotDetected(__FILE__, __LINE__);
55
56 }
57
58 SolarInputs solar_inputs;
59
60 test_solar_ptr = new Solar(8760, solar_inputs);
62 // ----- END CONSTRUCTION -----//
64
65
66 // ====== ATTRIBUTES =========
68 testTruth(
69
      not solar_inputs.renewable_inputs.production_inputs.print_flag,
      __FILE__,
70
      __LINE_
71
72 );
73
74 testFloatEquals(
75
       test_solar_ptr->type,
76
      RenewableType :: SOLAR,
      ___FILE___,
77
      __LINE_
78
79);
80
81 testTruth(
82
     test_solar_ptr->type_str == "SOLAR",
83
      ___FILE___,
84
      __LINE_
85);
86
87 testFloatEquals(
88
     test_solar_ptr->capital_cost,
89
      350118.723363,
      __FILE___,
90
      __LINE_
91
92);
94 testFloatEquals(
95
     test_solar_ptr->operation_maintenance_cost_kWh,
96
      0.01,
      ___FILE_
97
      __LINE_
98
99);
100
101 // ====== END ATTRIBUTES ======
102
103
104
105 // ====== METHODS ======== //
106
107 // test production constraints
108 testFloatEquals(
       test_solar_ptr->computeProductionkW(0, 1, 2),
109
110
       100,
       __FILE__,
111
112
        __LINE__
113);
114
115 testFloatEquals(
       test_solar_ptr->computeProductionkW(0, 1, -1),
116
117
       0.
       __FILE__,
118
119
       __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt vec hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
    1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
126
127
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0
128
129
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double solar_resource_kWm2 = 0;
```

```
136
137 for (int i = 0; i < 48; i++) {
138     roll = (double) rand() / RAND_MAX;
139
140
        solar resource kWm2 = roll;
141
142
        roll = (double)rand() / RAND_MAX;
143
144
        if (roll <= 0.1) {</pre>
145
             solar_resource_kWm2 = 0;
146
147
148
        else if (roll >= 0.95) {
149
            solar_resource_kWm2 = 1.25;
150
151
        roll = (double)rand() / RAND_MAX;
152
153
        if (roll >= 0.95) {
154
155
            roll = 1.25;
156
157
        load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
158
        load_kW = load_vec_kW[i];
159
160
161
        production_kW = test_solar_ptr->computeProductionkW(
162
163
             dt_vec_hrs[i],
164
             solar_resource_kWm2
165
        );
166
167
        load_kW = test_solar_ptr->commit(
168
169
             dt_vec_hrs[i],
170
            production_kW,
171
             load_kW
172
        );
173
174
        // is running (or not) as expected
175
        if (solar_resource_kWm2 > 0) {
176
             testTruth(
                 test_solar_ptr->is_running,
__FILE___,
177
178
179
                 __LINE__
180
            );
181
        }
182
183
        else {
            testTruth(
184
185
                not test_solar_ptr->is_running,
186
                 ___FILE___,
187
                 __LINE__
188
            );
189
        }
190
        // load_kW <= load_vec_kW (i.e., after vs before)</pre>
191
192
        testLessThanOrEqualTo(
193
             load_kW,
194
             load_vec_kW[i],
195
            ___FILE___,
             __LINE_
196
197
        );
198
199
        // production = dispatch + storage + curtailment
200
        testFloatEquals(
201
            test_solar_ptr->production_vec_kW[i] -
202
             test_solar_ptr->dispatch_vec_kW[i] -
             test_solar_ptr->storage_vec_kW[i]
203
204
             test_solar_ptr->curtailment_vec_kW[i],
205
             Ο,
            __FILE__,
206
207
             __LINE__
208
        );
209
210
        // capacity constraint
211
        if (solar_resource_kWm2 > 1) {
212
             testFloatEquals(
213
                 test_solar_ptr->production_vec_kW[i],
214
                 test_solar_ptr->capacity_kW,
                 ___FILE___,
215
216
                 __LINE__
217
            );
218
219
220
        // resource, O\&M > 0 whenever solar is running (i.e., producing)
221
        if (test_solar_ptr->is_running) {
222
             testGreaterThan(
```

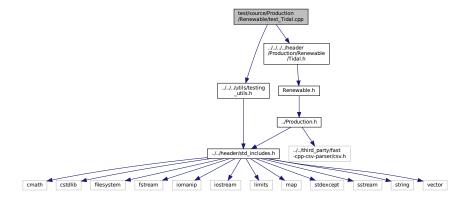
```
solar_resource_kWm2,
                0,
__FILE__,
224
225
                __LINE_
226
2.2.7
           );
228
           testGreaterThan(
230
                test_solar_ptr->operation_maintenance_cost_vec[i],
231
                __FILE__,
232
233
                __LINE__
234
           );
235
       }
236
237
        // resource, O\&M = 0 whenever solar is not running (i.e., not producing)
238
            testFloatEquals(
239
240
                solar_resource_kWm2,
241
                __FILE__,
243
244
           );
245
           testFloatEquals(
246
247
                test_solar_ptr->operation_maintenance_cost_vec[i],
                ___FILE___,
249
250
                __LINE__
           );
251
       }
252
253 }
254
255
256 // ====== END METHODS ======= //
257
258 } /* try */
259
260
261 catch (...) {
      delete test_solar_ptr;
263
       printGold(" .... ");
printRed("FAIL");
2.64
265
266
       std::cout « std::endl;
267
       throw;
268 }
269
270
271 delete test_solar_ptr;
272
273 printGold(" .... ");
274 printGreen("PASS");
275 std::cout « std::endl;
276 return 0;
277 } /* main() */
```

# 5.35 test/source/Production/Renewable/test Tidal.cpp File Reference

Testing suite for Tidal class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Renewable/Tidal.h"
```

Include dependency graph for test\_Tidal.cpp:



# **Functions**

• int main (int argc, char \*\*argv)

# 5.35.1 Detailed Description

Testing suite for Tidal class.

A suite of tests for the Tidal class.

#### 5.35.2 Function Documentation

### 5.35.2.1 main()

```
int main (
             int argc,
             char ** argv )
27 {
28
      #ifdef _WIN32
         activateVirtualTerminal();
30
      #endif /* _WIN32 */
31
32
33
      printGold("\tTesting Production <-- Renewable <-- Tidal");</pre>
34
      srand(time(NULL));
35
      Renewable* test_tidal_ptr;
37
38 try {
39
40 // ----- CONSTRUCTION -----//
42 bool error_flag = true;
43
44 try {
      TidalInputs bad_tidal_inputs;
45
46
      bad_tidal_inputs.design_speed_ms = -1;
47
48
      Tidal bad_tidal(8760, bad_tidal_inputs);
```

```
49
      error_flag = false;
51 } catch (...)
     // Task failed successfully! =P
52
53 }
54 if (not error flag) {
      expectedErrorNotDetected(__FILE__, __LINE__);
55
56 }
57
58 TidalInputs tidal_inputs;
59
60 test_tidal_ptr = new Tidal(8760, tidal_inputs);
62 // ----- END CONSTRUCTION -----//
64
65
66 // ====== ATTRIBUTES ========
68 testTruth(
69
      not tidal_inputs.renewable_inputs.production_inputs.print_flag,
70
      __LINE_
71
72);
73
74 testFloatEquals(
75
      test_tidal_ptr->type,
76
      RenewableType :: TIDAL,
      ___FILE___,
77
      __LINE_
78
79);
80
81 testTruth(
82
     test_tidal_ptr->type_str == "TIDAL",
83
      ___FILE___,
84
      __LINE_
85);
86
87 testFloatEquals(
88
      test_tidal_ptr->capital_cost,
89
      500237.446725,
      __FILE___,
90
      __LINE_
91
92);
94 testFloatEquals(
95
    test_tidal_ptr->operation_maintenance_cost_kWh,
96
      0.069905,
      ___FILE___
97
98
      __LINE
99);
100
101 // ====== END ATTRIBUTES =====
102
103
104
105 // ----- METHODS ----- //
106
107 // test production constraints
108 testFloatEquals(
       test_tidal_ptr->computeProductionkW(0, 1, 1e6),
109
110
       ___FILE___,
111
112
       __LINE__
113 );
114
115 testFloatEquals(
       test_tidal_ptr->computeProductionkW(
116
117
           0.
118
119
           ((Tidal*)test_tidal_ptr)->design_speed_ms
120
121
       test_tidal_ptr->capacity_kW,
122
       ___FILE___,
123
       __LINE__
124);
125
126 testFloatEquals(
127
       test_tidal_ptr->computeProductionkW(0, 1, -1),
128
       0,
       __FILE_
129
       __LINE__
130
131 );
132
133 // test commit()
134 std::vector<double> dt_vec_hrs (48, 1);
135
```

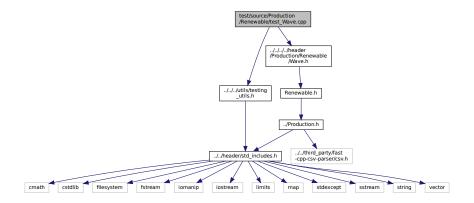
```
136 std::vector<double> load_vec_kW = {
      1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
137
138
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
139
140
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double tidal_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149
        roll = (double) rand() / RAND_MAX;
150
151
        tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
152
153
        roll = (double) rand() / RAND MAX;
154
155
        if (roll <= 0.1) {</pre>
156
             tidal_resource_ms = 0;
        }
157
158
        else if (roll >= 0.95) {
159
160
             tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
161
162
163
        roll = (double)rand() / RAND_MAX;
164
165
        if (roll >= 0.95) {
166
             roll = 1.25;
167
168
169
        load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
170
        load_kW = load_vec_kW[i];
171
172
        production_kW = test_tidal_ptr->computeProductionkW(
173
174
             dt_vec_hrs[i],
175
             tidal_resource_ms
176
        );
177
        load_kW = test_tidal_ptr->commit(
178
179
180
             dt_vec_hrs[i],
181
             production_kW,
182
             load_kW
183
        );
184
185
        // is running (or not) as expected
186
         if (production_kW > 0) {
187
             testTruth(
188
                 test_tidal_ptr->is_running,
189
                 ___FILE___,
                 __LINE
190
191
             );
192
        }
193
194
        else {
195
             testTruth(
196
                not test_tidal_ptr->is_running,
197
                 ___FILE___,
198
                 __LINE__
199
             );
200
201
         // load_kW <= load_vec_kW (i.e., after vs before)</pre>
202
        testLessThanOrEqualTo(
203
204
             load_kW,
205
             load_vec_kW[i],
206
             ___FILE___,
207
             __LINE__
208
        );
209
         // production = dispatch + storage + curtailment
210
211
         testFloatEquals(
212
             test_tidal_ptr->production_vec_kW[i] -
213
             test_tidal_ptr->dispatch_vec_kW[i] -
214
             test_tidal_ptr->storage_vec_kW[i]
             test_tidal_ptr->curtailment_vec_kW[i],
215
216
            Ο,
             __FILE__,
217
218
             __LINE__
219
        );
220
         // resource, O\&M > 0 whenever tidal is running (i.e., producing)
221
        if (test_tidal_ptr->is_running) {
222
```

```
223
            testGreaterThan(
224
                tidal_resource_ms,
225
                Ο,
                ___FILE_
226
2.2.7
                __LINE__
228
            );
230
231
                test_tidal_ptr->operation_maintenance_cost_vec[i],
232
                ___FILE___,
233
234
                 __LINE
235
            );
236
237
238
        // O\&M = 0 whenever tidal is not running (i.e., not producing)
239
            testFloatEquals(
240
241
                test_tidal_ptr->operation_maintenance_cost_vec[i],
                ___FILE___,
243
244
                __LINE__
2.45
            );
246
247 }
248
249
250 // ====== END METHODS ======= //
2.51
252 } /* try */
253
254
255 catch (...) {
256
        delete test_tidal_ptr;
2.57
        printGold(" .... ");
printRed("FAIL");
258
259
260
        std::cout « std::endl;
261
262 }
263
2.64
265 delete test_tidal_ptr;
266
267 printGold(" .... ");
268 printGreen("PASS");
269 std::cout « std::endl;
270 return 0;
       /* main() */
271 }
```

# 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();
29
30
     #endif /* _WIN32 */
31
     printGold("\tTesting Production <-- Renewable <-- Wave");</pre>
     srand(time(NULL));
35
     Renewable* test_wave_ptr;
36
37
38 try {
40 // ----- CONSTRUCTION -----//
42 bool error_flag = true;
43
44 try {
      WaveInputs bad_wave_inputs;
45
     bad_wave_inputs.design_significant_wave_height_m = -1;
     Wave bad_wave(8760, bad_wave_inputs);
48
49
50
     error_flag = false;
51 } catch (...) {
52  // Task failed successfully! =P
54 if (not error_flag) {
5.5
      expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57
58 WaveInputs wave_inputs;
60 test_wave_ptr = new Wave(8760, wave_inputs);
62 // ====== END CONSTRUCTION =========== //
63
64
66 // ----- ATTRIBUTES ----- //
67
68 testTruth(
     not wave_inputs.renewable_inputs.production_inputs.print_flag,
69
     __FILE__,
70
71
72);
73
74 testFloatEquals(
75
     test_wave_ptr->type,
76
     RenewableType :: WAVE,
     ___FILE___,
```

```
__LINE__
78
79);
80
81 testTruth(
      test_wave_ptr->type_str == "WAVE",
82
       __FILE__,
83
      __LINE_
85);
86
87 testFloatEquals(
88
      test_wave_ptr->capital_cost,
      850831.063539,
89
      __FILE__,
90
91
92);
93
94 testFloatEquals(
      test_wave_ptr->operation_maintenance_cost_kWh,
95
      0.069905,
96
      __FILE__,
      __LINE_
98
99);
100
101 // ====== END ATTRIBUTES =======
102
103
104
105 // ----- METHODS -----//
106
107 // test production constraints
108 testFloatEquals(
109
       test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
110
111
       ___FILE___,
       __LINE__
112
113);
114
115 testFloatEquals(
116
       test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
117
       __FILE__,
118
       __LINE__
119
120);
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
       126
127
128
129
       1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double significant_wave_height_m = 0;
136 double energy_period_s = 0;
137
138 for (int i = 0; i < 48; i++) {
       roll = (double) rand() / RAND_MAX;
139
140
141
       if (roll <= 0.05) {</pre>
142
           roll = 0;
143
144
       significant_wave_height_m = roll *
145
           ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
146
147
148
       roll = (double)rand() / RAND_MAX;
149
       if (roll <= 0.05) {
    roll = 0;</pre>
150
151
152
153
154
       energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
155
       roll = (double)rand() / RAND_MAX;
156
157
       if (roll >= 0.95) {
158
159
           roll = 1.25;
160
161
162
       load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
163
       load_kW = load_vec_kW[i];
164
```

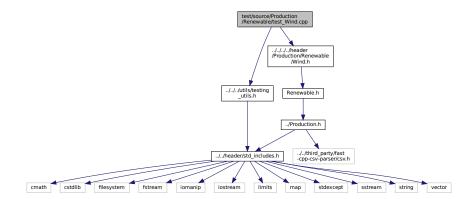
```
165
        production_kW = test_wave_ptr->computeProductionkW(
166
167
            dt_vec_hrs[i],
168
            \verb|significant_wave_height_m|,\\
169
            energy_period_s
170
        );
171
172
        load_kW = test_wave_ptr->commit(
173
174
            dt_vec_hrs[i],
175
            production_kW,
176
            load_kW
177
        );
178
179
        // is running (or not) as expected
180
        if (production_kW > 0) {
181
            testTruth(
182
                test_wave_ptr->is_running,
                __FILE__,
183
184
                __LINE__
185
            );
186
        }
187
188
        else {
189
            testTruth(
190
               not test_wave_ptr->is_running,
191
                ___FILE___,
192
                __LINE__
193
            );
194
        }
195
196
        // load_kW <= load_vec_kW (i.e., after vs before)</pre>
197
        testLessThanOrEqualTo(
198
            load_kW,
199
            load_vec_kW[i],
200
            ___FILE___,
201
            __LINE__
202
        );
203
204
        // production = dispatch + storage + curtailment
205
        testFloatEquals(
            test_wave_ptr->production_vec_kW[i] -
206
            test_wave_ptr->dispatch_vec_kW[i] -
2.07
            test_wave_ptr->storage_vec_kW[i]
208
209
            test_wave_ptr->curtailment_vec_kW[i],
            0,
__FILE__,
210
211
212
            __LINE__
213
       );
214
        // resource, O&M > 0 whenever wave is running (i.e., producing)
215
216
        if (test_wave_ptr->is_running) {
217
            testGreaterThan(
218
                significant_wave_height_m,
219
                0,
                ___FILE_
220
221
                __LINE__
222
            );
223
224
            testGreaterThan(
225
                energy_period_s,
226
                Ο,
                ___FILE___,
227
228
                __LINE__
229
            );
230
2.31
            testGreaterThan(
                test_wave_ptr->operation_maintenance_cost_vec[i],
232
233
                0.
234
                ___FILE___,
235
                __LINE__
236
            );
237
        }
238
239
        // O&M = 0 whenever wave is not running (i.e., not producing)
240
241
            testFloatEquals(
242
                test_wave_ptr->operation_maintenance_cost_vec[i],
                0,
__FILE_
243
244
245
                __LINE__
246
            );
247
248 }
249 // ====== END METHODS =======//
2.50
251 }
       /* try */
```

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

# 5.37 test/source/Production/Renewable/test\_Wind.cpp File Reference

Testing suite for Wind class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Renewable/Wind.h"
Include dependency graph for test Wind.cpp:
```



#### **Functions**

• int main (int argc, char \*\*argv)

# 5.37.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind class.

#### 5.37.2 Function Documentation

#### 5.37.2.1 main()

```
int main (
            int argc,
            char ** argv )
27 {
28
     #ifdef _WIN32
         activateVirtualTerminal();
29
30
      #endif /* _WIN32 */
31
32
     printGold("\tTesting Production <-- Renewable <-- Wind");</pre>
33
     srand(time(NULL));
34
35
36
     Renewable* test wind ptr;
38 try {
39
40 // ----- CONSTRUCTION -----//
41
42 bool error_flag = true;
43
44 try {
45
     WindInputs bad_wind_inputs;
46
     bad_wind_inputs.design_speed_ms = -1;
47
48
    Wind bad_wind(8760, bad_wind_inputs);
49
50
     error_flag = false;
51 } catch (...) {
52
     // Task failed successfully! =P
53 }
54 if (not error_flag) {
     expectedErrorNotDetected(__FILE__, __LINE__);
55
57
58 WindInputs wind_inputs;
59
60 test_wind_ptr = new Wind(8760, wind_inputs);
62 // ====== END CONSTRUCTION ========== //
64
6.5
66 // ----- ATTRIBUTES ----- //
67
68 testTruth(
69
     not wind_inputs.renewable_inputs.production_inputs.print_flag,
     __FILE__,
70
71
     __LINE__
72 );
73
74 testFloatEquals(
      test_wind_ptr->type,
76
      RenewableType :: WIND,
77
     ___FILE___,
     __LINE_
78
79);
80
81 testTruth(
     test_wind_ptr->type_str == "WIND",
83
     ___FILE___,
84
     __LINE__
85);
86
87 testFloatEquals(
   test_wind_ptr->capital_cost,
88
29
     450356.170088,
90
     ___FILE___,
91
     __LINE__
92);
93
94 testFloatEquals(
95
      test_wind_ptr->operation_maintenance_cost_kWh,
96
      0.034953,
     __FILE__,
97
98
      __LINE__
99);
100
101 // ====== END ATTRIBUTES ========
102
103
104
105 // ----- METHODS ------//
```

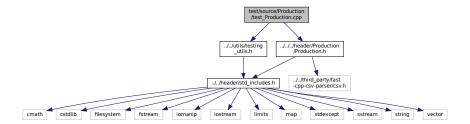
```
107 // test production constraints
108 testFloatEquals(
109
        test_wind_ptr->computeProductionkW(0, 1, 1e6),
110
        Ο,
        ___FILE_
111
        __LINE_
112
113 );
114
115 testFloatEquals(
116
        test_wind_ptr->computeProductionkW(
            Ο,
117
118
            1.
       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

```
194
       else {
195
           testTruth(
196
               not test_wind_ptr->is_running,
197
               ___FILE___,
               __LINE_
198
199
           );
200
201
202
        // load_kW <= load_vec_kW (i.e., after vs before)
203
        testLessThanOrEqualTo(
204
            load_kW,
205
           load_vec_kW[i],
206
            __FILE__,
207
208
209
        // production = dispatch + storage + curtailment
210
211
        testFloatEquals(
           test_wind_ptr->production_vec_kW[i] -
212
            test_wind_ptr->dispatch_vec_kW[i]
214
            test_wind_ptr->storage_vec_kW[i]
215
            test_wind_ptr->curtailment_vec_kW[i],
216
           Ο,
           ___FILE___,
217
218
            __LINE_
219
       );
220
221
        // resource, O&M > 0 whenever wind is running (i.e., producing)
        if (test_wind_ptr->is_running) {
222
223
            testGreaterThan(
224
               wind_resource_ms,
225
               Ο,
226
               __FILE__,
227
                __LINE__
228
           );
229
230
           testGreaterThan(
                test_wind_ptr->operation_maintenance_cost_vec[i],
232
233
                ___FILE___,
234
                __LINE__
           );
235
       }
236
237
        // O&M = 0 whenever wind is not running (i.e., not producing)
239
240
           testFloatEquals(
2.41
                test_wind_ptr->operation_maintenance_cost_vec[i],
               Ο,
242
               ___FILE_
243
244
                __LINE_
245
246
247 }
248
249
250 // ----- END METHODS -----//
251
252 }
       /* try */
253
254
255 catch (...) {
256
       delete test_wind_ptr;
258
        printGold(" ..... ");
        printRed("FAIL");
259
2.60
        std::cout « std::endl;
261
        throw:
262 }
263
264
265 delete test_wind_ptr;
266
267 printGold(" ..... ");
268 printGreen("PASS");
269 std::cout « std::endl;
270 return 0;
271 } /* main() */
```

# 5.38 test/source/Production/test\_Production.cpp File Reference

Testing suite for Production class.

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



#### **Functions**

• int main (int argc, char \*\*argv)

# 5.38.1 Detailed Description

Testing suite for Production class.

A suite of tests for the Production class.

# 5.38.2 Function Documentation

#### 5.38.2.1 main()

```
int main (
             int argc,
             char ** argv )
27 {
28
      #ifdef _WIN32
      activateVirtualTerminal();
#endif /* _WIN32 */
29
30
31
      printGold("\n\tTesting Production");
33
      srand(time(NULL));
34
35
36
37 try {
38
39 // ------ CONSTRUCTION ------//
41 bool error_flag = true;
42
43 try {
      ProductionInputs production_inputs;
      Production bad_production(0, production_inputs);
47
      error_flag = false;
48
49 } catch (...) {
50 // Task failed successfully! =P
```

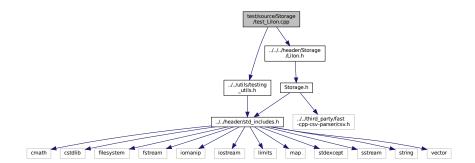
```
52 if (not error_flag) {
       expectedErrorNotDetected(__FILE__, __LINE__);
54 }
55
56 ProductionInputs production_inputs;
58 Production test_production(8760, production_inputs);
60 // ====== END CONSTRUCTION ======== //
61
62
63
64 // ====== ATTRIBUTES ============
66 testTruth(
67
      not production_inputs.print_flag,
       ___FILE___,
68
       __LINE__
69
70);
72 testFloatEquals(
73
       production_inputs.nominal_inflation_annual,
74
       0.02,
       ___FILE
7.5
76
       __LINE_
77 );
78
79 testFloatEquals(
80
       production_inputs.nominal_discount_annual,
81
       0.04.
       ___FILE_
82
83
       __LINE__
84);
8.5
86 testFloatEquals(
87
       {\tt test\_production.n\_points,}
88
       8760,
       __FILE__,
89
90
       __LINE__
91);
92
93 testFloatEquals(
94
       {\tt test\_production.capacity\_kW,}
95
       100,
       __FILE__,
96
97
       __LINE__
98);
99
100 testFloatEquals(
        test_production.real_discount_annual,
101
        0.0196078431372549,
102
103
        ___FILE___,
104
        __LINE__
105);
106
107 testFloatEquals(
108
        test_production.production_vec_kW.size(),
109
        8760,
110
        __FILE_
111
        __LINE__
112);
113
114 testFloatEquals(
115
        test_production.dispatch_vec_kW.size(),
116
        8760.
        ___FILE___,
117
118
        __LINE__
119);
120
121 testFloatEquals(
122
        test_production.storage_vec_kW.size(),
123
        8760,
        ___FILE
124
125
        __LINE__
126);
128 testFloatEquals(
129
        test_production.curtailment_vec_kW.size(),
130
        8760.
        ___FILE_
131
        __LINE__
132
133 );
134
135 testFloatEquals(
136
        {\tt test\_production.capital\_cost\_vec.size(),}
137
        8760,
        __FILE_
138
```

```
__LINE__
139
140);
141
142 testFloatEquals(
       {\tt test\_production.operation\_maintenance\_cost\_vec.size(),}
143
144
145
       __FILE__,
146
       __LINE__
147);
148
149 // ----- END ATTRIBUTES ------ //
150
151 } /* try */
152
153
154 catch (...) {
155
156
157
      printGold(" .....");
158
      printRed("FAIL");
159
       std::cout « std::endl;
160
161 }
162
163
164 printGold(" .... ");
165 printGreen("PASS");
166 std::cout « std::endl;
167 return 0;
168
169 } /* main() */
```

# 5.39 test/source/Storage/test\_Lilon.cpp File Reference

Testing suite for Lilon class.

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



# **Functions**

• int main (int argc, char \*\*argv)

# 5.39.1 Detailed Description

Testing suite for Lilon class.

A suite of tests for the Lilon class.

# 5.39.2 Function Documentation

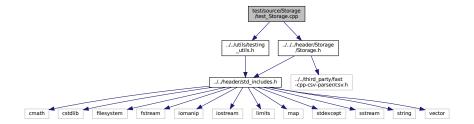
#### 5.39.2.1 main()

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

# 5.40 test/source/Storage/test\_Storage.cpp File Reference

Testing suite for Storage class.

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



### **Functions**

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

# 5.40.1 Detailed Description

Testing suite for Storage class.

A suite of tests for the Storage class.

#### 5.40.2 Function Documentation

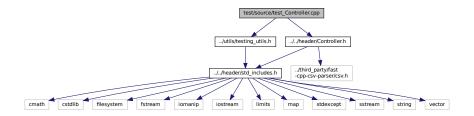
#### 5.40.2.1 main()

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

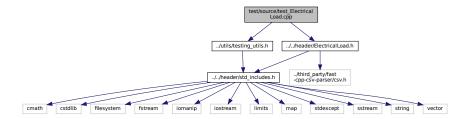
#### 5.41.2.1 main()

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

# 5.42 test/source/test\_ElectricalLoad.cpp File Reference

Testing suite for ElectricalLoad class.

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



#### **Functions**

• int main (int argc, char \*\*argv)

### 5.42.1 Detailed Description

Testing suite for ElectricalLoad class.

A suite of tests for the ElectricalLoad class.

# 5.42.2 Function Documentation

#### 5.42.2.1 main()

```
int main (
            int argc,
           char ** argv )
27 {
     #ifdef _WIN32
        activateVirtualTerminal();
29
30
     #endif /* _WIN32 */
31
    printGold("\tTesting ElectricalLoad");
32
33
34
     srand(time(NULL));
35
36
37 try {
38
39 // ====== CONSTRUCTION ===
41 std::string path_2_electrical_load_time_series =
      "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
42
43
44 ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
46 // ====== END CONSTRUCTION ========== //
48
49
50 // ----- ATTRIBUTES ----- //
52 testTruth(
```

```
53
        test_electrical_load.path_2_electrical_load_time_series ==
        path_2_electrical_load_time_series,
       __FILE__,
55
        __LINE_
56
57);
58
59 testFloatEquals(
60
        test_electrical_load.n_points,
61
        8760,
       ___FILE___,
62
63
        __LINE__
64);
65
66 testFloatEquals(
67
        test_electrical_load.n_years,
68
        0.999886,
       __FILE__,
69
        __LINE__
70
71);
73 testFloatEquals(
74
        test_electrical_load.min_load_kW,
       82.1211213927802,
7.5
       ___FILE___,
76
        __LINE_
78);
79
80 testFloatEquals(
81
        test_electrical_load.mean_load_kW,
       258.373472633202,
82
       __FILE__,
83
        __LINE_
85);
86
87
88 testFloatEquals(
89
        test_electrical_load.max_load_kW,
90
        500,
       ___FILE___,
91
92
        __LINE__
93);
94
9.5
96 std::vector<double> expected_dt_vec_hrs (48, 1);
98 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
99
100
101
102
103 };
104
105 std::vector<double> expected_load_vec_kW = {
106
        360.253836463674,
355.171277826775,
107
         353.776453532298,
108
109
         353.75405737934,
110
         346.592867404975,
111
         340.132411175118,
112
         337.354867340578.
113
         340.644115618736.
114
         363.639028500678,
115
         378.787797779238,
116
         372.215798201712,
         395.093925731298,
117
         402.325427142659,
118
         386.907725462306,
119
         380.709170928091,
120
         372.062070914977,
121
         372.328646856954,
122
123
         391.841444284136,
124
         394.029351759596,
         383.369407765254,
125
         381.093099675206.
126
         382.604158946193,
127
128
         390.744843709034,
129
         383.13949492437,
130
         368.150393976985,
         364.629744480226,
131
         363.572736804082,
132
133
         359.854924202248,
         355.207590170267,
134
135
         349.094656012401,
136
         354.365935871597,
137
         343.380608328546,
         404.673065729266.
138
139
         486.296896820126,
```

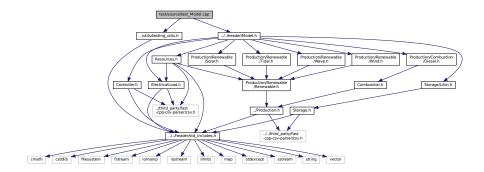
```
140
       480.225974100847,
141
       457.318764401085,
142
       418.177339948609,
143
       414.399018364126,
144
       409.678420185754,
145
       404.768766016563,
       401.699589920585,
146
147
       402.44339040654,
148
       398.138372541906,
149
       396.010498627646,
150
   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

# 5.43 test/source/test\_Model.cpp File Reference

Testing suite for Model class.

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

Include dependency graph for test\_Model.cpp:



# **Functions**

• int main (int argc, char \*\*argv)

# 5.43.1 Detailed Description

Testing suite for Model class.

A suite of tests for the Model class.

# 5.43.2 Function Documentation

#### 5.43.2.1 main()

```
int main (
              int argc,
              char ** argv )
27 {
28
       #ifdef _WIN32
      activateVirtualTerminal();
#endif /* _WIN32 */
29
30
31
32
       printGold("\tTesting Model");
33
       srand(time(NULL));
34
35
36
37 try {
38
39 // ------ CONSTRUCTION ------//
40
41 bool error_flag = true;
42
43 try {
      ModelInputs bad_model_inputs;
bad_model_inputs.path_2_electrical_load_time_series =
44
45
           "data/test/bad_path_240984069830.csv";
46
47
      Model bad_model(bad_model_inputs);
48
49
50
      error_flag = false;
51 } catch (...) {
```

```
// Task failed successfully! =P
54 if (not error_flag) {
55
       expectedErrorNotDetected(__FILE__, __LINE__);
56 }
58 std::string path_2_electrical_load_time_series =
59
       "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
60
61 ModelInputs test_model_inputs;
62 test_model_inputs.path_2_electrical_load_time_series =
63
     path_2_electrical_load_time_series;
65 Model test_model(test_model_inputs);
67 // ====== END CONSTRUCTION ========
68
69
70 // ----- ATTRIBUTES ----- //
73
       test_model.electrical_load.path_2_electrical_load_time_series ==
74
       path_2_electrical_load_time_series,
       ___FILE___,
7.5
       __LINE_
76
77 );
78
79 testFloatEquals(
80
      test_model.electrical_load.n_points,
81
       8760.
       ___FILE
82
83
       __LINE__
84);
8.5
86 testFloatEquals(
87
     test_model.electrical_load.n_years,
88
       0.999886,
      __FILE__,
89
       __LINE__
91);
92
93 testFloatEquals(
      test_model.electrical_load.min_load_kW,
94
95
       82.1211213927802,
       ___FILE___,
97
       __LINE__
98);
99
100 testFloatEquals(
101
       test model.electrical load.mean load kW.
        258.373472633202,
102
103
        ___FILE___,
104
        __LINE__
105);
106
107
108 testFloatEquals(
109
        test_model.electrical_load.max_load_kW,
110
        500,
        ___FILE
111
112
        __LINE__
113);
114
115
116 std::vector<double> expected_dt_vec_hrs (48, 1);
117
118 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,
119
120
121
122
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
123 };
124
125 std::vector<double> expected_load_vec_kW = {
        360.253836463674,
126
127
        355.171277826775,
128
        353.776453532298,
129
        353.75405737934,
130
        346.592867404975.
        340.132411175118.
131
        337.354867340578,
132
        340.644115618736,
133
134
        363.639028500678,
135
        378.787797779238,
136
        372.215798201712,
137
        395.093925731298.
        402.325427142659,
138
```

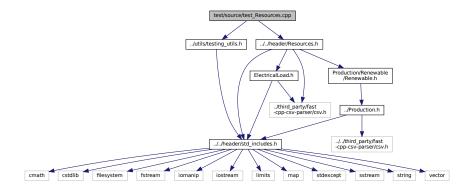
```
139
       386.907725462306,
140
       380.709170928091,
141
       372.062070914977,
142
       372.328646856954,
143
       391.841444284136.
       394.029351759596,
144
       383.369407765254,
145
146
       381.093099675206,
147
       382.604158946193,
148
       390.744843709034,
      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

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

# 5.44 test/source/test\_Resources.cpp File Reference

Testing suite for Resources class.

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



# **Functions**

• int main (int argc, char \*\*argv)

# 5.44.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

# 5.44.2 Function Documentation

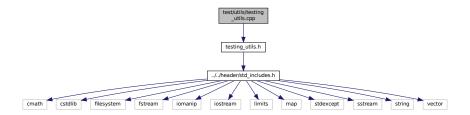
#### 5.44.2.1 main()

```
int main (
           int argc,
           char ** argv )
27 {
     #ifdef _WIN32
2.8
        activateVirtualTerminal();
29
    #endif /* _WIN32 */
30
31
32
    printGold("\tTesting Resources");
33
34
    srand(time(NULL));
35
36
37 try {
39 // ----- CONSTRUCTION -----//
40
41 Resources test_resources;
42
43 // ----- END CONSTRUCTION -----/
45
46
47 // ====== ATTRIBUTES ========== //
48
49 testFloatEquals(
50
     test_resources.resource_map_1D.size(),
51
52
     ___FILE___,
     __LINE__
53
54);
55
56 testFloatEquals(
     test_resources.path_map_1D.size(),
58
59
     ___FILE___,
     __LINE__
60
61);
62
63 testFloatEquals(
64
     test_resources.resource_map_2D.size(),
65
     Ο,
     ___FILE___,
66
     __LINE__
67
68);
70 testFloatEquals(
71
    test_resources.path_map_2D.size(),
     Ο,
72
     ___FILE___,
73
     __LINE__
74
75);
77 // ====== END ATTRIBUTES ========= //
78
79
80 // ----- METHODS -----//
83
84 // ====== END METHODS ========= //
8.5
86 } /* try */
87
88
89 catch (...) {
   printGold(" .....
printRed("FAIL");
90
                     .....");
91
92
     std::cout « std::endl;
93
     throw;
94 }
95
97 printGold(" ..... ");
98 printGreen("PASS");
99 std::cout « std::endl;
100 return 0;
101 } /* 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 < v

• void testLessThanOrEqualTo (double x, double y, std::string file, int line)

Tests if  $x \le y$ .

• 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()

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

#### **Parameters**

file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
432 {
         \verb|std::string| error_str = "\n ERROR failed to throw expected error prior to line";
433
        error_str += std::to_string(line);
error_str += " of ";
error_str += file;
434
435
436
437
438
        #ifdef _WIN32
        std::cout « error_str « std::endl;
#endif
439
440
441
442
        throw std::runtime_error(error_str);
443
444 }
        /* expectedErrorNotDetected() */
```

#### 5.45.2.2 printGold()

A function that sends gold text to std::cout.

#### **Parameters**

```
input_str The text of the string to be sent to std::cout.
```

#### 5.45.2.3 printGreen()

A function that sends green text to std::cout.

input_str	The text of the string to be sent to std::cout.
-----------	---

```
64 {
65     std::cout « "\x1B[32m" « input_str « "\033[0m";
66     return;
67 } /* printGreen() */
```

#### 5.45.2.4 printRed()

A function that sends red text to std::cout.

#### **Parameters**

*input\_str* The text of the string to be sent to std::cout.

#### 5.45.2.5 testFloatEquals()

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT\_TOLERANCE).

X	The first of two numbers to test.
У	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) {</pre>
140
141
142
143
         std::string error_str = "ERROR: testFloatEquals():\t in ";
         error_str += file;
error_str += "\tline ";
144
145
          error_str += std::to_string(line);
error_str += ":\t\n";
146
147
         error_str += std::to_string(x);
error_str += " and ";
148
149
         error_str += std::to_string(y);
error_str += " are not equal to within +/- ";
150
151
         error_str += std::to_string(FLOAT_TOLERANCE);
error_str += "\n";
152
153
154
         #ifdef _WIN32
155
156
              std::cout « error_str « std::endl;
157
```

```
158
159          throw std::runtime_error(error_str);
160          return;
161 }          /* testFloatEquals() */
```

#### 5.45.2.6 testGreaterThan()

#### Tests if x > y.

#### **Parameters**

X	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
191 {
           if (x > y) {
192
193
                 return;
194
195
196
           std::string error_str = "ERROR: testGreaterThan():\t in ";
           std::string error_str = "ERROR: testG
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 ";
197
198
199
200
201
203
           error_str += std::to_string(y);
204
           error_str += "\n";
205
206
           #ifdef _WIN32
207
                std::cout « error_str « std::endl;
208
           #endif
209
210
           throw std::runtime_error(error_str);
211
          return;
/* testGreaterThan() */
212 }
```

# 5.45.2.7 testGreaterThanOrEqualTo()

# Tests if $x \ge y$ .

Х	The first of two numbers to test.
---	-----------------------------------

#### **Parameters**

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

```
242 {
        if (x >= y) {
243
        return;
244
245
246
247
        std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
        error_str += file;
error_str += "\tline ";
248
249
         error_str += std::to_string(line);
250
        error_str += ":\t\n";
251
        error_str += std::to_string(x);
error_str += " is not greater than or equal to ";
252
253
        error_str += std::to_string(y);
error_str += "\n";
254
255
256
257
        #ifdef _WIN32
258
            std::cout « error_str « std::endl;
259
260
261
        throw std::runtime_error(error_str);
262
         return:
263 }
        /* testGreaterThanOrEqualTo() */
```

#### 5.45.2.8 testLessThan()

# Tests if $\mathbf{x} < \mathbf{y}$ .

X	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
293 {
294
         if (x < y) {</pre>
295
              return;
296
297
298
         std::string error_str = "ERROR: testLessThan():\t in ";
         error_str += file;
error_str += "\tline ";
299
300
         error_str += std::to_string(line);
error_str += ":\t\n";
301
302
         error_str += std::to_string(x);
error_str += " is not less than ";
303
304
         error_str += std::to_string(y);
error_str += "\n";
305
306
307
308
         #ifdef _WIN32
309
              std::cout « error_str « std::endl;
310
         #endif
311
312
         throw std::runtime_error(error_str);
313
          return:
```

```
314 } /* testLessThan() */
```

### 5.45.2.9 testLessThanOrEqualTo()

#### Tests if $x \le y$ .

#### **Parameters**

X	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
344 {
345
           if (x \le y) {
346
                return;
347
348
          std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
349
350
          error_str += file;
error_str += "\tline ";
351
          error_str += std::to_string(line);
error_str += ":\t\n";
352
353
          error_str += ":\\\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";
354
355
356
357
359
          #ifdef _WIN32
360
                std::cout « error_str « std::endl;
361
362
          #endif
363
          throw std::runtime_error(error_str);
364
          return;
          /* testLessThanOrEqualTo() */
```

# 5.45.2.10 testTruth()

Tests if the given statement is true.

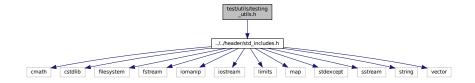
statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
392 {
393
         if (statement) {
394
395
396
397
         std::string error_str = "ERROR: testTruth():\t in ";
398
         error_str += file;
399
         error_str += "\tline ";
        error_str += std::to_string(line);
error_str += ":\t\n";
error_str += "Given statement is not true";
400
401
402
403
404
         #ifdef _WIN32
405
             std::cout « error_str « std::endl;
406
         #endif
407
         throw std::runtime_error(error_str);
408
409
         return;
410 }
        /* testTruth() */
```

# 5.46 test/utils/testing utils.h File Reference

Header file for various PGMcpp testing utilities.

#include "../../header/std\_includes.h"
Include dependency graph for testing\_utils.h:



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



### **Macros**

• #define FLOAT\_TOLERANCE 1e-6

A tolerance for application to floating point equality tests.

#### **Functions**

void printGreen (std::string)

A function that sends green text to std::cout.

void printGold (std::string)

A function that sends gold text to std::cout.

void printRed (std::string)

A function that sends red text to std::cout.

void testFloatEquals (double, double, std::string, int)

Tests for the equality of two floating point numbers x and y (to within FLOAT\_TOLERANCE).

void testGreaterThan (double, double, std::string, int)

```
Tests if x > y.
```

• void testGreaterThanOrEqualTo (double, double, std::string, int)

```
Tests if x >= y.
```

• void testLessThan (double, double, std::string, int)

```
Tests if x < y.
```

void testLessThanOrEqualTo (double, double, std::string, int)

```
Tests if x \le 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()

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

file	The file in which the test is applied (you should be able to just pass in "FILE").	
lin	The line of the file in which the test is applied (you should be able to just pass in "LINE	_").

```
432 {
433
        std::string error_str = "\n ERROR failed to throw expected error prior to line ";
        error_str += std::to_string(line);
error_str += " of ";
434
435
        error_str += file;
436
437
438
        #ifdef _WIN32
439
           std::cout « error_str « std::endl;
440
441
442
        throw std::runtime_error(error_str);
443
        return:
       /* expectedErrorNotDetected() */
444 }
```

### 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.

#### 5.46.3.3 printGreen()

A function that sends green text to std::cout.

#### **Parameters**

*input\_str* The text of the string to be sent to std::cout.

```
64 {
65     std::cout « "\x1B[32m" « input_str « "\033[0m";
66     return;
67 } /* printGreen() */
```

# 5.46.3.4 printRed()

A function that sends red text to std::cout.

#### **Parameters**

*input\_str* The text of the string to be sent to std::cout.

### 5.46.3.5 testFloatEquals()

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT\_TOLERANCE).

#### **Parameters**

Х	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

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

# 5.46.3.6 testGreaterThan()

#### Tests if x > y.

#### **Parameters**

Х	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
191 {
192
          if (x > y) {
             return;
193
194
195
196
          std::string error_str = "ERROR: testGreaterThan():\t in ";
          error_str += file;
error_str += "\tline ";
197
198
          error_str += std::to_string(line);
error_str += ":\t\n";
199
200
         error_str += std::to_string(x);
error_str += " is not greater than ";
error_str += std::to_string(y);
error_str += "\n";
201
202
203
204
205
206
207
               std::cout « error_str « std::endl;
208
          #endif
209
210
          throw std::runtime_error(error_str);
211
          return;
212 }
         /* testGreaterThan() */
```

#### 5.46.3.7 testGreaterThanOrEqualTo()

Tests if  $x \ge y$ .

X	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
242 {
243
         if (x >= y) {
        return;
244
245
246
         std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
247
         error_str += file;
248
         error_str += "\tline ";
249
         error_str += std::to_string(line);
error_str += ":\t\n";
250
251
        error_str += std::to_string(x);
error_str += " is not greater than or equal to ";
252
253
        error_str += std::to_string(y);
error_str += "\n";
254
255
256
        #ifdef _WIN32
257
2.58
            std::cout « error_str « std::endl;
259
        #endif
260
         throw std::runtime_error(error_str);
```

```
262    return;
263 } /* testGreaterThanOrEqualTo() */
```

# 5.46.3.8 testLessThan()

#### Tests if x < y.

#### **Parameters**

Х	The first of two numbers to test.
У	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "FILE").
line	The line of the file in which the test is applied (you should be able to just pass in "LINE").

```
293 {
294
            if (x < y) {
295
296
297
           std::string error_str = "ERROR: testLessThan():\t in ";
error_str += file;
error_str += "\tline ";
298
            error_str += std::to_string(line);
error_str += ":\t\n";
301
302
           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";
303
304
305
306
307
308
           #ifdef _WIN32
           std::cout « error_str « std::endl; #endif
309
310
311
312
            throw std::runtime_error(error_str);
313
314 } /* testLessThan() */
```

# 5.46.3.9 testLessThanOrEqualTo()

### Tests if $x \le y$ .

X	The first of two numbers to test.	
У	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_Generate	d by Doxygen

```
344 {
345
         if (x \le y) {
346
              return;
347
348
         std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
349
         error_str += file;
error_str += "\tline ";
350
351
         error_str += std::to_string(line);
error_str += ":\t\n";
352
353
         error_str += std::to_string(x);
error_str += " is not less than or equal to ";
354
355
         error_str += std::to_string(y);
error_str += "\n";
356
357
358
359
         #ifdef _WIN32
360
         std::cout « error_str « std::endl;
#endif
361
362
363
         throw std::runtime_error(error_str);
364
365 } /* testLessThanOrEqualTo() */
```

#### 5.46.3.10 testTruth()

Tests if the given statement is true.

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").

```
393
         if (statement) {
394
              return;
395
396
397
         std::string error_str = "ERROR: testTruth():\t in ";
         error_str += file;
error_str += "\tline ";
398
399
         error_str += std::to_string(line);
error_str += ":\t\n";
error_str += "Given statement is not true";
400
401
402
403
404
         #ifdef _WIN32
405
             std::cout « error_str « std::endl;
406
         #endif
407
408
         throw std::runtime_error(error_str);
409
         return;
410 }
         /* testTruth() */
```

# Index

$\sim$ Combustion	DieselInputs, 26
Combustion, 10	CH4_emissions_vec_kg
$\sim$ Controller	Combustion, 14
Controller, 18	CH4_kg
$\sim$ Diesel	Emissions, 34
Diesel, 22	clear
$\sim$ ElectricalLoad	Controller, 19
ElectricalLoad, 30	ElectricalLoad, 30
$\sim$ Lilon	Model, 40
Lilon, 36	Resources, 60
$\sim$ Model	CO2_emissions_intensity_kgL
Model, 39	Combustion, 14
$\sim$ Production	DieselInputs, 26
Production, 45	CO2_emissions_vec_kg
$\sim$ Renewable	Combustion, 14
Renewable, 55	CO2_kg
~Resources	Emissions, 34
Resources, 59	CO_emissions_intensity_kgL
~Solar	Combustion, 14
Solar, 64	DieselInputs, 26
~Storage	CO_emissions_vec_kg
Storage, 69	Combustion, 14
∼Tidal	CO kg
Tidal, 72	Emissions, 34
~Wave	Combustion, 7
Wave, 79	~Combustion, 10
$\sim$ Wind	CH4_emissions_intensity_kgL, 13
Wind, 87	CH4_emissions_vec_kg, 14
	CO2_emissions_intensity_kgL, 14
addResource	CO2_emissions_vec_kg, 14
Model, 39	CO_emissions_intensity_kgL, 14
addResource1D	CO_emissions_vec_kg, 14
Resources, 59	Combustion, 9
addResource2D	commit, 10
Resources, 60	fuel_consumption_vec_L, 14
	fuel_cost_L, 15
capacity_kW	fuel cost vec, 15
Production, 47	getEmissionskg, 12
ProductionInputs, 51	getFuelConsumptionL, 13
capital_cost	linear_fuel_intercept_LkWh, 15
DieselInputs, 25	linear_fuel_slope_LkWh, 15
Production, 47	NOx emissions intensity kgL, 15
SolarInputs, 67	NOx_emissions_vec_kg, 15
TidalInputs, 76	PM_emissions_intensity_kgL, 16
WaveInputs, 83	PM_emissions_vec_kg, 16
WindInputs, 91	requestProductionkW, 13
capital_cost_vec	SOx emissions intensity kgL, 16
Production, 47	SOx_emissions_vec_kg, 16
CH4_emissions_intensity_kgL	type, 16
0 1 1 10	1, PO, 10

Combustion, 13

Combustion.h	Tidal, 74
CombustionType, 97	TidalInputs, 76
DIESEL, 97	Wind, 89
N_COMBUSTION_TYPES, 97	WindInputs, 91
combustion_inputs	DIESEL
DieselInputs, 26	Combustion.h, 97
combustion_ptr_vec	Diesel, 19
Model, 40	$\sim$ Diesel, 22
CombustionInputs, 17	commit, 22
production_inputs, 17	Diesel, 21
CombustionType	minimum_load_ratio, 23
Combustion.h, 97	minimum_runtime_hrs, 24
commit	requestProductionkW, 23
Combustion, 10	time_since_last_start_hrs, 24
Diesel, 22	DieselInputs, 24
Production, 46	capital_cost, 25
Renewable, 55	CH4_emissions_intensity_kgL, 26
Solar, 65	CO2_emissions_intensity_kgL, 26
Tidal, 73	CO_emissions_intensity_kgL, 26
Wave, 80	combustion_inputs, 26
Wind, 87	fuel_cost_L, 26
computeProductionkW	linear_fuel_intercept_LkWh, 26
Renewable, 56	linear_fuel_slope_LkWh, 27
Solar, 65	minimum_load_ratio, 27
Tidal, 73	minimum_runtime_hrs, 27
Wave, 80	NOx_emissions_intensity_kgL, 27
Wind, 88	operation_maintenance_cost_kWh, 27
control_mode	PM_emissions_intensity_kgL, 28
Controller, 19	replace_running_hrs, 28
ModelInputs, 42	SOx_emissions_intensity_kgL, 28
Controller, 18	dispatch_vec_kW
~Controller, 18	Production, 48
clear, 19	dt_vec_hrs
control_mode, 19	ElectricalLoad, 32
Controller, 18 controller	electrical load
	Model, 41
Model, 41 Controller.h	ElectricalLoad, 28
	∼ElectricalLoad, 30
ControlMode, 94 CYCLE CHARGING, 94	clear, 30
LOAD FOLLOWING, 94	dt_vec_hrs, 32
N CONTROL MODES, 94	ElectricalLoad, 29, 30
ControlMode	load_vec_kW, 32
Controller.h, 94	max_load_kW, 32
curtailment_vec_kW	mean_load_kW, 32
Production, 47	min load kW, 32
CYCLE CHARGING	n_points, 33
Controller.h, 94	n_years, 33
Controllerin, 54	path_2_electrical_load_time_series, 33
derating	readLoadData, 31
Solar, 66	time_vec_hrs, 33
SolarInputs, 68	Emissions, 33
design_energy_period_s	CH4_kg, 34
Wave, 82	CO2_kg, 34
WaveInputs, 83	CO_kg, <mark>34</mark>
design_significant_wave_height_m	NOx_kg, 34
Wave, 82	PM_kg, 35
WaveInputs, 83	SOx_kg, 35
design_speed_ms	expectedErrorNotDetected
· -	•

testing_utils.cpp, 157 testing_utils.h, 164	test_Controller.cpp, 148 test_Diesel.cpp, 121
testing_utils.n, 104	test_ElectricalLoad.cpp, 149
FLOAT_TOLERANCE	test_Lilon.cpp, 146
testing utils.h, 164	test_Lilon.cpp, 146 test_Model.cpp, 152
fuel consumption vec L	test Production.cpp, 143
Combustion, 14	test_Renewable.cpp, 126
fuel cost L	test_Resources.cpp, 155
Combustion, 15	test_Solar.cpp, 128
DieselInputs, 26	test_Storage.cpp, 147
fuel cost vec	test_Tidal.cpp, 132
Combustion, 15	test Wave.cpp, 136
	test_Wind.cpp, 139
getEmissionskg	max_load_kW
Combustion, 12	ElectricalLoad, 32
getFuelConsumptionL	mean_load_kW
Combustion, 13	ElectricalLoad, 32
la a a da v/O a vatura lla vila 100	min_load_kW
header/Controller.h, 93	ElectricalLoad, 32
header/ElectricalLoad.h, 94	minimum load ratio
header/Model.h, 95	Diesel, 23
header/Production/Combustion/Combustion.h, 96	DieselInputs, 27
header/Production/Combustion/Diesel.h, 98	minimum_runtime_hrs
header/Production/Production.h, 99	Diesel, 24
header/Production/Renewable/Renewable.h, 99	DieselInputs, 27
header/Production/Renewable/Solar.h, 101	Model, 37
header/Production/Renewable/Tidal.h, 102	$\sim$ Model, 39
header/Production/Renewable/Wave.h, 103 header/Production/Renewable/Wind.h, 105	addResource, 39
header/Resources.h, 106	clear, 40
header/std_includes.h, 107	combustion_ptr_vec, 40
header/Storage/Lilon.h, 108	controller, 41
header/Storage/Storage.h, 109	electrical_load, 41
neader/Storage/Storage.n, 103	Model, 38
is_running	renewable_ptr_vec, 41
Production, 48	reset, 40
is_running_vec	resources, 41
Production, 48	storage_ptr_vec, 41
is sunk	ModelInputs, 42
Production, 48	control_mode, 42
ProductionInputs, 52	path_2_electrical_load_time_series, 42
·	N. COMPLICTION TYPES
levellized_cost_of_energy_kWh	N_COMBUSTION_TYPES
Production, 48	Combustion.h, 97
Lilon, 35	N_CONTROL_MODES
$\sim$ Lilon, 36	Controller.h, 94
Lilon, 36	n_points
linear_fuel_intercept_LkWh	ElectricalLoad, 33
Combustion, 15	Production, 48
Diesellnputs, 26	N_RENEWABLE_TYPES
linear_fuel_slope_LkWh	Renewable.h, 100
Combustion, 15	n_replacements Production, 49
Diesellnputs, 27	
LOAD_FOLLOWING	n_starts
Controller.h, 94	Production, 49 N_TIDAL_POWER_PRODUCTION_MODELS
load_vec_kW	Tidal.h, 103
ElectricalLoad, 32	N_WAVE_POWER_PRODUCTION_MODELS
main test_Combustion.cpp, 119	Wave.h, 104 N_WIND_POWER_PRODUCTION_MODELS
iesi_oombustion.opp, 113	IN_NAUND_I ONNEIT_I HODOGHON_MODELS

Wind.h, 106	testing_utils.h, 165
n_years	Production, 43
ElectricalLoad, 33	∼Production, 45
net_present_cost	capacity_kW, 47
Production, 49	capital_cost, 47
nominal_discount_annual	capital_cost_vec, 47
ProductionInputs, 52	commit, 46
nominal_inflation_annual	curtailment_vec_kW, 47
ProductionInputs, 52	dispatch_vec_kW, 48
NOx_emissions_intensity_kgL	is_running, 48
Combustion, 15	is_running_vec, 48
Diesellnputs, 27	is_sunk, 48
NOx_emissions_vec_kg	levellized_cost_of_energy_kWh, 48
Combustion, 15	n_points, 48
NOx_kg	n_replacements, 49
Emissions, 34	n_starts, 49
	net_present_cost, 49
operation_maintenance_cost_kWh	operation_maintenance_cost_kWh, 49
Diesellnputs, 27	operation_maintenance_cost_vec, 49
Production, 49	print_flag, 49
SolarInputs, 68	Production, 44, 45
TidalInputs, 76	production_vec_kW, 50
WaveInputs, 84	real_discount_annual, 50
WindInputs, 91	replace_running_hrs, 50
operation_maintenance_cost_vec	running_hours, 50
Production, 49	storage_vec_kW, 50
	type_str, 50
path_2_electrical_load_time_series	production_inputs
ElectricalLoad, 33	CombustionInputs, 17
ModelInputs, 42	RenewableInputs, 57
path_map_1D	production_vec_kW
Resources, 60	Production, 50
path_map_2D	ProductionInputs, 51
Resources, 60	capacity_kW, 51
PM_emissions_intensity_kgL	is_sunk, 52
Combustion, 16	nominal_discount_annual, 52
Diesellnputs, 28	nominal_inflation_annual, 52
PM_emissions_vec_kg	print_flag, 52
Combustion, 16	replace_running_hrs, 52
PM_kg	PYBIND11_MODULE
Emissions, 35	PYBIND11_PGM.cpp, 110
power_model	PYBIND11_PGM.cpp
Tidal, 74	PYBIND11_MODULE, 110
TidalInputs, 76	pybindings/PYBIND11_PGM.cpp, 109
Wave, 82	
WaveInputs, 84	readLoadData
Wind, 89	ElectricalLoad, 31
WindInputs, 91	real_discount_annual
print_flag	Production, 50
Production, 49	Renewable, 53
ProductionInputs, 52	$\sim$ Renewable, 55
printGold	commit, 55
testing_utils.cpp, 158	computeProductionkW, 56
testing_utils.h, 165	Renewable, 54
printGreen	resource_key, 56
testing_utils.cpp, 158	type, 56
testing_utils.h, 165	Renewable.h
printRed	N_RENEWABLE_TYPES, 100
testing_utils.cpp, 159	RenewableType, 100

SOLAR, 100	derating, 68
TIDAL, 100	operation_maintenance_cost_kWh, 68
WAVE, 100	renewable_inputs, 68
WIND, 100	resource_key, 68
renewable_inputs	source/Controller.cpp, 111
SolarInputs, 68	source/ElectricalLoad.cpp, 112
TidalInputs, 77	source/Model.cpp, 112
WaveInputs, 84	source/Production/Combustion/Combustion.cpp, 113
WindInputs, 91	source/Production/Combustion/Diesel.cpp, 113
renewable_ptr_vec	source/Production/Production.cpp, 114
Model, 41	source/Production/Renewable/Renewable.cpp, 114
RenewableInputs, 57	source/Production/Renewable/Solar.cpp, 115
production_inputs, 57	source/Production/Renewable/Tidal.cpp, 115
RenewableType	source/Production/Renewable/Wave.cpp, 116
Renewable.h, 100	source/Production/Renewable/Wind.cpp, 116
replace_running_hrs	source/Resources.cpp, 117
DieselInputs, 28	source/Storage/Lilon.cpp, 117
Production, 50	source/Storage/Storage.cpp, 118
ProductionInputs, 52	SOx_emissions_intensity_kgL
requestProductionkW	Combustion, 16
Combustion, 13	DieselInputs, 28
Diesel, 23	SOx_emissions_vec_kg
reset	Combustion, 16
Model, 40	SOx_kg
resource_key	Emissions, 35
Renewable, 56	Storage, 69
SolarInputs, 68	$\sim$ Storage, 69
TidalInputs, 77	Storage, 69
WaveInputs, 84	storage_ptr_vec
WindInputs, 91	Model, 41
resource_map_1D	storage_vec_kW
Resources, 61	Production, 50
resource_map_2D	
Resources, 61	test/source/Production/Combustion/test_Combustion.cpp
Resources, 58	118
$\sim$ Resources, 59	test/source/Production/Combustion/test_Diesel.cpp,
addResource1D, 59	120
addResource2D, 60	test/source/Production/Renewable/test_Renewable.cpp,
clear, 60	126
path_map_1D, 60	test/source/Production/Renewable/test_Solar.cpp, 127
path_map_2D, 60	test/source/Production/Renewable/test_Tidal.cpp, 131
resource_map_1D, 61	test/source/Production/Renewable/test_Wave.cpp, 135
resource_map_2D, 61	test/source/Production/Renewable/test_Wind.cpp, 139
Resources, 58	test/source/Production/test_Production.cpp, 142
resources	test/source/Storage/test_Lilon.cpp, 145
Model, 41	test/source/Storage/test_Storage.cpp, 146
running_hours	test/source/test_Controller.cpp, 147
Production, 50	test/source/test_ElectricalLoad.cpp, 148
00115	test/source/test_Model.cpp, 151
SOLAR	test/source/test_Resources.cpp, 155
Renewable.h, 100	test/utils/testing_utils.cpp, 157
Solar, 61	test/utils/testing_utils.h, 163
~Solar, 64	test_Combustion.cpp
commit, 65	main, 119
computeProductionkW, 65	test_Controller.cpp
derating, 66	main, 148
Solar, 62, 64	test_Diesel.cpp
SolarInputs, 66	main, 121
capital_cost, 67	test_ElectricalLoad.cpp

main, 149	testing_utils.h, 168
test_Lilon.cpp	testTruth
main, 146	testing_utils.cpp, 162
test_Model.cpp	testing_utils.h, 169
main, 152	TIDAL
test_Production.cpp	Renewable.h, 100
main, 143	Tidal, 70
test_Renewable.cpp	$\sim$ Tidal, 72
main, 126	commit, 73
test_Resources.cpp	computeProductionkW, 73
main, 155	design_speed_ms, 74
test_Solar.cpp	power_model, 74
main, 128	Tidal, 71, 72 Tidal.h
test_Storage.cpp main, 147	N TIDAL POWER PRODUCTION MODELS,
test_Tidal.cpp	103
main, 132	TIDAL POWER CUBIC, 103
test_Wave.cpp	TIDAL POWER EXPONENTIAL, 103
main, 136	TIDAL POWER LOOKUP, 103
test Wind.cpp	TidalPowerProductionModel, 103
main, 139	TIDAL POWER CUBIC
testFloatEquals	Tidal.h, 103
testing_utils.cpp, 159	TIDAL_POWER_EXPONENTIAL
testing utils.h, 166	Tidal.h, 103
testGreaterThan	TIDAL POWER LOOKUP
testing_utils.cpp, 160	Tidal.h, 103
testing_utils.h, 166	TidalInputs, 75
testGreaterThanOrEqualTo	capital_cost, 76
testing_utils.cpp, 160	design_speed_ms, 76
testing_utils.h, 167	operation_maintenance_cost_kWh, 76
testing_utils.cpp	power_model, 76
expectedErrorNotDetected, 157	renewable_inputs, 77
printGold, 158	resource_key, 77
printGreen, 158	TidalPowerProductionModel
printRed, 159	Tidal.h, 103
testFloatEquals, 159	time_since_last_start_hrs
testGreaterThan, 160	Diesel, 24
testGreaterThanOrEqualTo, 160	time_vec_hrs
testLessThan, 161	ElectricalLoad, 33
testLessThanOrEqualTo, 162	type
testTruth, 162	Combustion, 16
testing_utils.h expectedErrorNotDetected, 164	Renewable, 56
FLOAT_TOLERANCE, 164	type_str Production, 50
printGold, 165	1 Toddellott, 30
printGreen, 165	WAVE
printRed, 165	Renewable.h, 100
testFloatEquals, 166	Wave, 77
testGreaterThan, 166	$\sim$ Wave, 79
testGreaterThanOrEqualTo, 167	commit, 80
testLessThan, 168	computeProductionkW, 80
testLessThanOrEqualTo, 168	design_energy_period_s, 82
testTruth, 169	design_significant_wave_height_m, 82
testLessThan	power_model, 82
testing_utils.cpp, 161	Wave, 78, 79
testing_utils.h, 168	Wave.h
testLessThanOrEqualTo	N_WAVE_POWER_PRODUCTION_MODELS,
testing_utils.cpp, 162	104
	WAVE_POWER_GAUSSIAN, 104

```
WAVE_POWER_LOOKUP, 104
    WAVE_POWER_PARABOLOID, 104
    WavePowerProductionModel, 104
WAVE_POWER_GAUSSIAN
    Wave.h, 104
WAVE POWER LOOKUP
    Wave.h, 104
WAVE_POWER_PARABOLOID
    Wave.h, 104
WaveInputs, 82
    capital_cost, 83
    design_energy_period_s, 83
    design_significant_wave_height_m, 83
    operation_maintenance_cost_kWh, 84
    power_model, 84
    renewable_inputs, 84
    resource key, 84
WavePowerProductionModel
    Wave.h, 104
WIND
    Renewable.h, 100
Wind, 85
    \simWind, 87
    commit, 87
    computeProductionkW, 88
    design_speed_ms, 89
    power_model, 89
    Wind, 86
Wind.h
    N WIND POWER PRODUCTION MODELS, 106
    WIND_POWER_EXPONENTIAL, 106
    WIND_POWER_LOOKUP, 106
    WindPowerProductionModel, 106
WIND_POWER_EXPONENTIAL
    Wind.h, 106
WIND_POWER_LOOKUP
    Wind.h, 106
WindInputs, 90
    capital_cost, 91
    design speed ms, 91
    operation maintenance cost kWh, 91
    power_model, 91
    renewable_inputs, 91
    resource key, 91
WindPowerProductionModel
    Wind.h, 106
```