

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
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]	10
4.1.2.2 Combustion() [2/2]	10
4.1.2.3 ~Combustion()	11
4.1.3 Member Function Documentation	11
4.1.3.1 __checkInputs()	11
4.1.3.2 __writeSummary()	11
4.1.3.3 __writeTimeSeries()	12
4.1.3.4 commit()	12
4.1.3.5 computeEconomics()	13
4.1.3.6 computeFuelAndEmissions()	13
4.1.3.7 getEmissionskg()	14
4.1.3.8 getFuelConsumptionL()	14
4.1.3.9 requestProductionkW()	15
4.1.3.10 writeResults()	15
4.1.4 Member Data Documentation	16
4.1.4.1 CH4_emissions_intensity_kgL	16
4.1.4.2 CH4_emissions_vec_kg	16
4.1.4.3 CO2_emissions_intensity_kgL	16
4.1.4.4 CO2_emissions_vec_kg	16
4.1.4.5 CO_emissions_intensity_kgL	16
4.1.4.6 CO_emissions_vec_kg	17
4.1.4.7 fuel_consumption_vec_L	17
4.1.4.8 fuel_cost_L	17
4.1.4.9 fuel_cost_vec	17
4.1.4.10 linear_fuel_intercept_LkWh	17
4.1.4.11 linear_fuel_slope_LkWh	17
4.1.4.12 NOx_emissions_intensity_kgL	18
4.1.4.13 NOx_emissions_vec_kg	18
4.1.4.14 PM_emissions_intensity_kgL	18
4.1.4.15 PM_emissions_vec_kg	18

4.1.4.16 SOx_emissions_intensity_kgL	18
4.1.4.17 SOx_emissions_vec_kg	18
4.1.4.18 total_emissions	19
4.1.4.19 total_fuel_consumed_L	19
4.1.4.20 type	19
4.2 CombustionInputs Struct Reference	19
4.2.1 Detailed Description	20
4.2.2 Member Data Documentation	20
4.2.2.1 production_inputs	20
4.3 Controller Class Reference	20
4.3.1 Detailed Description	21
4.3.2 Constructor & Destructor Documentation	22
4.3.2.1 Controller()	22
4.3.2.2 ~Controller()	22
4.3.3 Member Function Documentation	22
4.3.3.1 __applyCycleChargingControl_CHARGING()	22
4.3.3.2 __applyCycleChargingControl_DISCHARGING()	23
4.3.3.3 __applyLoadFollowingControl_CHARGING()	24
4.3.3.4 __applyLoadFollowingControl_DISCHARGING()	25
4.3.3.5 __computeNetLoad()	25
4.3.3.6 __constructCombustionMap()	26
4.3.3.7 __getRenewableProduction()	28
4.3.3.8 __handleCombustionDispatch()	29
4.3.3.9 __handleStorageCharging() [1/2]	30
4.3.3.10 __handleStorageCharging() [2/2]	31
4.3.3.11 __handleStorageDischarging()	31
4.3.3.12 applyDispatchControl()	31
4.3.3.13 clear()	33
4.3.3.14 init()	33
4.3.3.15 setControlMode()	33
4.3.4 Member Data Documentation	34
4.3.4.1 combustion_map	34
4.3.4.2 control_mode	34
4.3.4.3 control_string	34
4.3.4.4 missed_load_vec_kW	35
4.3.4.5 net_load_vec_kW	35
4.4 Diesel Class Reference	35
4.4.1 Detailed Description	37
4.4.2 Constructor & Destructor Documentation	37
4.4.2.1 Diesel() [1/2]	37
4.4.2.2 Diesel() [2/2]	37
4.4.2.3 ~Diesel()	38

4.4.3 Member Function Documentation	39
4.4.3.1 __checkInputs()	39
4.4.3.2 __getGenericCapitalCost()	40
4.4.3.3 __getGenericFuelIntercept()	41
4.4.3.4 __getGenericFuelSlope()	41
4.4.3.5 __getGenericOpMaintCost()	42
4.4.3.6 __handleStartStop()	42
4.4.3.7 __writeSummary()	43
4.4.3.8 __writeTimeSeries()	44
4.4.3.9 commit()	45
4.4.3.10 requestProductionkW()	47
4.4.4 Member Data Documentation	48
4.4.4.1 minimum_load_ratio	48
4.4.4.2 minimum_runtime_hrs	48
4.4.4.3 time_since_last_start_hrs	48
4.5 DieselInputs Struct Reference	49
4.5.1 Detailed Description	50
4.5.2 Member Data Documentation	50
4.5.2.1 capital_cost	50
4.5.2.2 CH4_emissions_intensity_kgL	50
4.5.2.3 CO2_emissions_intensity_kgL	51
4.5.2.4 CO_emissions_intensity_kgL	51
4.5.2.5 combustion_inputs	51
4.5.2.6 fuel_cost_L	51
4.5.2.7 linear_fuel_intercept_LkWh	51
4.5.2.8 linear_fuel_slope_LkWh	51
4.5.2.9 minimum_load_ratio	52
4.5.2.10 minimum_runtime_hrs	52
4.5.2.11 NOx_emissions_intensity_kgL	52
4.5.2.12 operation_maintenance_cost_kWh	52
4.5.2.13 PM_emissions_intensity_kgL	52
4.5.2.14 replace_running_hrs	52
4.5.2.15 SOx_emissions_intensity_kgL	53
4.6 ElectricalLoad Class Reference	53
4.6.1 Detailed Description	54
4.6.2 Constructor & Destructor Documentation	54
4.6.2.1 ElectricalLoad() [1/2]	54
4.6.2.2 ElectricalLoad() [2/2]	54
4.6.2.3 ~ElectricalLoad()	54
4.6.3 Member Function Documentation	54
4.6.3.1 clear()	55
4.6.3.2 readLoadData()	55

4.6.4 Member Data Documentation	56
4.6.4.1 dt_vec_hrs	56
4.6.4.2 load_vec_kW	56
4.6.4.3 max_load_kW	56
4.6.4.4 mean_load_kW	57
4.6.4.5 min_load_kW	57
4.6.4.6 n_points	57
4.6.4.7 n_years	57
4.6.4.8 path_2_electrical_load_time_series	57
4.6.4.9 time_vec_hrs	57
4.7 Emissions Struct Reference	58
4.7.1 Detailed Description	58
4.7.2 Member Data Documentation	58
4.7.2.1 CH4_kg	58
4.7.2.2 CO2_kg	58
4.7.2.3 CO_kg	59
4.7.2.4 NOx_kg	59
4.7.2.5 PM_kg	59
4.7.2.6 SOx_kg	59
4.8 Lilon Class Reference	60
4.8.1 Detailed Description	60
4.8.2 Constructor & Destructor Documentation	61
4.8.2.1 Lilon()	61
4.8.2.2 ~Lilon()	61
4.9 Model Class Reference	61
4.9.1 Detailed Description	63
4.9.2 Constructor & Destructor Documentation	63
4.9.2.1 Model() [1/2]	63
4.9.2.2 Model() [2/2]	63
4.9.2.3 ~Model()	64
4.9.3 Member Function Documentation	64
4.9.3.1 __checkInputs()	64
4.9.3.2 __computeEconomics()	64
4.9.3.3 __computeFuelAndEmissions()	65
4.9.3.4 __computeLevellizedCostOfEnergy()	65
4.9.3.5 __computeNetPresentCost()	66
4.9.3.6 __writeSummary()	66
4.9.3.7 __writeTimeSeries()	69
4.9.3.8 addDiesel()	69
4.9.3.9 addResource()	70
4.9.3.10 addSolar()	70
4.9.3.11 addTidal()	71

4.9.3.12 addWave()	71
4.9.3.13 addWind()	71
4.9.3.14 clear()	72
4.9.3.15 reset()	72
4.9.3.16 run()	73
4.9.3.17 writeResults()	73
4.9.4 Member Data Documentation	74
4.9.4.1 combustion_ptr_vec	74
4.9.4.2 controller	75
4.9.4.3 electrical_load	75
4.9.4.4 levellized_cost_of_energy_kWh	75
4.9.4.5 net_present_cost	75
4.9.4.6 renewable_ptr_vec	75
4.9.4.7 resources	75
4.9.4.8 storage_ptr_vec	76
4.9.4.9 total_dispatch_discharge_kWh	76
4.9.4.10 total_emissions	76
4.9.4.11 total_fuel_consumed_L	76
4.10 ModelInputs Struct Reference	76
4.10.1 Detailed Description	77
4.10.2 Member Data Documentation	77
4.10.2.1 control_mode	77
4.10.2.2 path_2_electrical_load_time_series	77
4.11 Production Class Reference	77
4.11.1 Detailed Description	79
4.11.2 Constructor & Destructor Documentation	79
4.11.2.1 Production() [1/2]	80
4.11.2.2 Production() [2/2]	80
4.11.2.3 ~Production()	81
4.11.3 Member Function Documentation	81
4.11.3.1 __checkInputs()	81
4.11.3.2 __computeRealDiscountAnnual()	82
4.11.3.3 __handleReplacement()	82
4.11.3.4 commit()	83
4.11.3.5 computeEconomics()	84
4.11.4 Member Data Documentation	85
4.11.4.1 capacity_kW	85
4.11.4.2 capital_cost	85
4.11.4.3 capital_cost_vec	85
4.11.4.4 curtailment_vec_kW	85
4.11.4.5 dispatch_vec_kW	85
4.11.4.6 is_running	86

4.11.4.7 is_running_vec	86
4.11.4.8 is_sunk	86
4.11.4.9 levellized_cost_of_energy_kWh	86
4.11.4.10 n_points	86
4.11.4.11 n_replacements	86
4.11.4.12 n_starts	87
4.11.4.13 n_years	87
4.11.4.14 net_present_cost	87
4.11.4.15 nominal_discount_annual	87
4.11.4.16 nominal_inflation_annual	87
4.11.4.17 operation_maintenance_cost_kWh	87
4.11.4.18 operation_maintenance_cost_vec	88
4.11.4.19 print_flag	88
4.11.4.20 production_vec_kW	88
4.11.4.21 real_discount_annual	88
4.11.4.22 replace_running_hrs	88
4.11.4.23 running_hours	88
4.11.4.24 storage_vec_kW	89
4.11.4.25 total_dispatch_kWh	89
4.11.4.26 type_str	89
4.12 ProductionInputs Struct Reference	89
4.12.1 Detailed Description	90
4.12.2 Member Data Documentation	90
4.12.2.1 capacity_kW	90
4.12.2.2 is_sunk	90
4.12.2.3 nominal_discount_annual	90
4.12.2.4 nominal_inflation_annual	90
4.12.2.5 print_flag	90
4.12.2.6 replace_running_hrs	91
4.13 Renewable Class Reference	91
4.13.1 Detailed Description	92
4.13.2 Constructor & Destructor Documentation	92
4.13.2.1 Renewable() [1/2]	93
4.13.2.2 Renewable() [2/2]	93
4.13.2.3 ~Renewable()	93
4.13.3 Member Function Documentation	94
4.13.3.1 __checkInputs()	94
4.13.3.2 __handleStartStop()	94
4.13.3.3 __writeSummary()	94
4.13.3.4 __writeTimeSeries()	95
4.13.3.5 commit()	95
4.13.3.6 computeEconomics()	96

4.13.3.7 computeProductionkW() [1/2]	96
4.13.3.8 computeProductionkW() [2/2]	96
4.13.3.9 writeResults()	96
4.13.4 Member Data Documentation	97
4.13.4.1 resource_key	97
4.13.4.2 type	98
4.14 RenewableInputs Struct Reference	98
4.14.1 Detailed Description	98
4.14.2 Member Data Documentation	98
4.14.2.1 production_inputs	99
4.15 Resources Class Reference	99
4.15.1 Detailed Description	100
4.15.2 Constructor & Destructor Documentation	100
4.15.2.1 Resources()	100
4.15.2.2 ~Resources()	100
4.15.3 Member Function Documentation	101
4.15.3.1 __checkResourceKey1D()	101
4.15.3.2 __checkResourceKey2D()	101
4.15.3.3 __checkTimePoint()	102
4.15.3.4 __readSolarResource()	103
4.15.3.5 __readTidalResource()	104
4.15.3.6 __readWaveResource()	105
4.15.3.7 __readWindResource()	106
4.15.3.8 __throwLengthError()	106
4.15.3.9 addResource()	107
4.15.3.10 clear()	108
4.15.4 Member Data Documentation	108
4.15.4.1 path_map_1D	109
4.15.4.2 path_map_2D	109
4.15.4.3 resource_map_1D	109
4.15.4.4 resource_map_2D	109
4.15.4.5 string_map_1D	109
4.15.4.6 string_map_2D	109
4.16 Solar Class Reference	110
4.16.1 Detailed Description	111
4.16.2 Constructor & Destructor Documentation	111
4.16.2.1 Solar() [1/2]	111
4.16.2.2 Solar() [2/2]	112
4.16.2.3 ~Solar()	112
4.16.3 Member Function Documentation	113
4.16.3.1 __checkInputs()	113
4.16.3.2 __getGenericCapitalCost()	113

4.16.3.3	__getGenericOpMaintCost()	113
4.16.3.4	__writeSummary()	114
4.16.3.5	__writeTimeSeries()	115
4.16.3.6	commit()	116
4.16.3.7	computeProductionkW()	116
4.16.4	Member Data Documentation	117
4.16.4.1	derating	117
4.17	SolarInputs Struct Reference	117
4.17.1	Detailed Description	118
4.17.2	Member Data Documentation	118
4.17.2.1	capital_cost	119
4.17.2.2	derating	119
4.17.2.3	operation_maintenance_cost_kWh	119
4.17.2.4	renewable_inputs	119
4.17.2.5	resource_key	119
4.18	Storage Class Reference	120
4.18.1	Detailed Description	120
4.18.2	Constructor & Destructor Documentation	120
4.18.2.1	Storage()	120
4.18.2.2	~Storage()	121
4.19	Tidal Class Reference	121
4.19.1	Detailed Description	123
4.19.2	Constructor & Destructor Documentation	123
4.19.2.1	Tidal() [1/2]	123
4.19.2.2	Tidal() [2/2]	123
4.19.2.3	~Tidal()	124
4.19.3	Member Function Documentation	125
4.19.3.1	__checkInputs()	125
4.19.3.2	__computeCubicProductionkW()	125
4.19.3.3	__computeExponentialProductionkW()	126
4.19.3.4	__computeLookupProductionkW()	127
4.19.3.5	__getGenericCapitalCost()	127
4.19.3.6	__getGenericOpMaintCost()	128
4.19.3.7	__writeSummary()	128
4.19.3.8	__writeTimeSeries()	129
4.19.3.9	commit()	130
4.19.3.10	computeProductionkW()	131
4.19.4	Member Data Documentation	132
4.19.4.1	design_speed_ms	132
4.19.4.2	power_model	132
4.19.4.3	power_model_string	132
4.20	TidalInputs Struct Reference	133

4.20.1 Detailed Description	133
4.20.2 Member Data Documentation	134
4.20.2.1 capital_cost	134
4.20.2.2 design_speed_ms	134
4.20.2.3 operation_maintenance_cost_kWh	134
4.20.2.4 power_model	134
4.20.2.5 renewable_inputs	134
4.20.2.6 resource_key	135
4.21 Wave Class Reference	135
4.21.1 Detailed Description	136
4.21.2 Constructor & Destructor Documentation	137
4.21.2.1 Wave() [1/2]	137
4.21.2.2 Wave() [2/2]	137
4.21.2.3 ~Wave()	138
4.21.3 Member Function Documentation	138
4.21.3.1 __checkInputs()	138
4.21.3.2 __computeGaussianProductionkW()	139
4.21.3.3 __computeLookupProductionkW()	140
4.21.3.4 __computeParaboloidProductionkW()	140
4.21.3.5 __getGenericCapitalCost()	141
4.21.3.6 __getGenericOpMaintCost()	142
4.21.3.7 __writeSummary()	142
4.21.3.8 __writeTimeSeries()	143
4.21.3.9 commit()	144
4.21.3.10 computeProductionkW()	145
4.21.4 Member Data Documentation	146
4.21.4.1 design_energy_period_s	146
4.21.4.2 design_significant_wave_height_m	146
4.21.4.3 power_model	147
4.21.4.4 power_model_string	147
4.22 WaveInputs Struct Reference	147
4.22.1 Detailed Description	148
4.22.2 Member Data Documentation	148
4.22.2.1 capital_cost	148
4.22.2.2 design_energy_period_s	148
4.22.2.3 design_significant_wave_height_m	149
4.22.2.4 operation_maintenance_cost_kWh	149
4.22.2.5 power_model	149
4.22.2.6 renewable_inputs	149
4.22.2.7 resource_key	149
4.23 Wind Class Reference	150
4.23.1 Detailed Description	151

4.23.2 Constructor & Destructor Documentation	151
4.23.2.1 Wind() [1/2]	152
4.23.2.2 Wind() [2/2]	152
4.23.2.3 ~Wind()	153
4.23.3 Member Function Documentation	153
4.23.3.1 __checkInputs()	153
4.23.3.2 __computeExponentialProductionkW()	154
4.23.3.3 __computeLookupProductionkW()	154
4.23.3.4 __getGenericCapitalCost()	155
4.23.3.5 __getGenericOpMaintCost()	155
4.23.3.6 __writeSummary()	155
4.23.3.7 __writeTimeSeries()	157
4.23.3.8 commit()	158
4.23.3.9 computeProductionkW()	158
4.23.4 Member Data Documentation	159
4.23.4.1 design_speed_ms	159
4.23.4.2 power_model	159
4.23.4.3 power_model_string	160
4.24 WindInputs Struct Reference	160
4.24.1 Detailed Description	161
4.24.2 Member Data Documentation	161
4.24.2.1 capital_cost	161
4.24.2.2 design_speed_ms	161
4.24.2.3 operation_maintenance_cost_kWh	161
4.24.2.4 power_model	162
4.24.2.5 renewable_inputs	162
4.24.2.6 resource_key	162
5 File Documentation	163
5.1 header/Controller.h File Reference	163
5.1.1 Detailed Description	164
5.1.2 Enumeration Type Documentation	164
5.1.2.1 ControlMode	164
5.2 header/ElectricalLoad.h File Reference	164
5.2.1 Detailed Description	165
5.3 header/Model.h File Reference	165
5.3.1 Detailed Description	166
5.4 header/Production/Combustion/Combustion.h File Reference	166
5.4.1 Detailed Description	167
5.4.2 Enumeration Type Documentation	167
5.4.2.1 CombustionType	167
5.5 header/Production/Combustion/Diesel.h File Reference	168

5.5.1 Detailed Description	169
5.6 header/Production/Production.h File Reference	169
5.6.1 Detailed Description	169
5.7 header/Production/Renewable/Renewable.h File Reference	170
5.7.1 Detailed Description	170
5.7.2 Enumeration Type Documentation	170
5.7.2.1 RenewableType	170
5.8 header/Production/Renewable/Solar.h File Reference	171
5.8.1 Detailed Description	172
5.9 header/Production/Renewable/Tidal.h File Reference	172
5.9.1 Detailed Description	173
5.9.2 Enumeration Type Documentation	173
5.9.2.1 TidalPowerProductionModel	173
5.10 header/Production/Renewable/Wave.h File Reference	173
5.10.1 Detailed Description	174
5.10.2 Enumeration Type Documentation	174
5.10.2.1 WavePowerProductionModel	174
5.11 header/Production/Renewable/Wind.h File Reference	175
5.11.1 Detailed Description	176
5.11.2 Enumeration Type Documentation	176
5.11.2.1 WindPowerProductionModel	176
5.12 header/Resources.h File Reference	176
5.12.1 Detailed Description	177
5.13 header/std_includes.h File Reference	177
5.13.1 Detailed Description	178
5.14 header/Storage/Lilon.h File Reference	178
5.14.1 Detailed Description	178
5.15 header/Storage/Storage.h File Reference	179
5.15.1 Detailed Description	179
5.16 pybindings/PYBIND11_PGM.cpp File Reference	179
5.16.1 Detailed Description	180
5.16.2 Function Documentation	180
5.16.2.1 PYBIND11_MODULE()	180
5.17 source/Controller.cpp File Reference	181
5.17.1 Detailed Description	181
5.18 source/ElectricalLoad.cpp File Reference	182
5.18.1 Detailed Description	182
5.19 source/Model.cpp File Reference	182
5.19.1 Detailed Description	182
5.20 source/Production/Combustion/Combustion.cpp File Reference	183
5.20.1 Detailed Description	183
5.21 source/Production/Combustion/Diesel.cpp File Reference	183

5.21.1 Detailed Description	183
5.22 source/Production/Production.cpp File Reference	184
5.22.1 Detailed Description	184
5.23 source/Production/Renewable/Renewable.cpp File Reference	184
5.23.1 Detailed Description	184
5.24 source/Production/Renewable/Solar.cpp File Reference	185
5.24.1 Detailed Description	185
5.25 source/Production/Renewable/Tidal.cpp File Reference	185
5.25.1 Detailed Description	185
5.26 source/Production/Renewable/Wave.cpp File Reference	186
5.26.1 Detailed Description	186
5.27 source/Production/Renewable/Wind.cpp File Reference	186
5.27.1 Detailed Description	186
5.28 source/Resources.cpp File Reference	187
5.28.1 Detailed Description	187
5.29 source/Storage/Lilon.cpp File Reference	187
5.29.1 Detailed Description	187
5.30 source/Storage/Storage.cpp File Reference	188
5.30.1 Detailed Description	188
5.31 test/source/Production/Combustion/test_Combustion.cpp File Reference	188
5.31.1 Detailed Description	189
5.31.2 Function Documentation	189
5.31.2.1 main()	189
5.32 test/source/Production/Combustion/test_Diesel.cpp File Reference	190
5.32.1 Detailed Description	191
5.32.2 Function Documentation	191
5.32.2.1 main()	191
5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference	196
5.33.1 Detailed Description	196
5.33.2 Function Documentation	196
5.33.2.1 main()	196
5.34 test/source/Production/Renewable/test_Solar.cpp File Reference	197
5.34.1 Detailed Description	198
5.34.2 Function Documentation	198
5.34.2.1 main()	198
5.35 test/source/Production/Renewable/test_Tidal.cpp File Reference	201
5.35.1 Detailed Description	201
5.35.2 Function Documentation	201
5.35.2.1 main()	202
5.36 test/source/Production/Renewable/test_Wave.cpp File Reference	204
5.36.1 Detailed Description	205
5.36.2 Function Documentation	205

5.36.2.1 main()	205
5.37 test/source/Production/Renewable/test_Wind.cpp File Reference	208
5.37.1 Detailed Description	209
5.37.2 Function Documentation	209
5.37.2.1 main()	209
5.38 test/source/Production/test_Production.cpp File Reference	212
5.38.1 Detailed Description	212
5.38.2 Function Documentation	212
5.38.2.1 main()	213
5.39 test/source/Storage/test_Lilon.cpp File Reference	214
5.39.1 Detailed Description	215
5.39.2 Function Documentation	215
5.39.2.1 main()	215
5.40 test/source/Storage/test_Storage.cpp File Reference	216
5.40.1 Detailed Description	216
5.40.2 Function Documentation	216
5.40.2.1 main()	216
5.41 test/source/test_Controller.cpp File Reference	217
5.41.1 Detailed Description	217
5.41.2 Function Documentation	217
5.41.2.1 main()	217
5.42 test/source/test_ElectricalLoad.cpp File Reference	218
5.42.1 Detailed Description	218
5.42.2 Function Documentation	219
5.42.2.1 main()	219
5.43 test/source/test_Model.cpp File Reference	221
5.43.1 Detailed Description	221
5.43.2 Function Documentation	221
5.43.2.1 main()	222
5.44 test/source/test_Resources.cpp File Reference	228
5.44.1 Detailed Description	228
5.44.2 Function Documentation	228
5.44.2.1 main()	229
5.45 test/utls/testing_utils.cpp File Reference	234
5.45.1 Detailed Description	235
5.45.2 Function Documentation	235
5.45.2.1 expectedErrorNotDetected()	235
5.45.2.2 printGold()	236
5.45.2.3 printGreen()	236
5.45.2.4 printRed()	236
5.45.2.5 testFloatEquals()	237
5.45.2.6 testGreaterThan()	237

5.45.2.7 testGreaterThanOrEqualTo()	238
5.45.2.8 testLessThan()	239
5.45.2.9 testLessThanOrEqualTo()	239
5.45.2.10 testTruth()	240
5.46 test/utls/testing_utils.h File Reference	240
5.46.1 Detailed Description	241
5.46.2 Macro Definition Documentation	242
5.46.2.1 FLOAT_TOLERANCE	242
5.46.3 Function Documentation	242
5.46.3.1 expectedErrorNotDetected()	242
5.46.3.2 printGold()	242
5.46.3.3 printGreen()	243
5.46.3.4 printRed()	243
5.46.3.5 testFloatEquals()	243
5.46.3.6 testGreaterThan()	244
5.46.3.7 testGreaterThanOrEqualTo()	245
5.46.3.8 testLessThan()	245
5.46.3.9 testLessThanOrEqualTo()	246
5.46.3.10 testTruth()	246
Bibliography	249
Index	251

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

CombustionInputs	19
Controller	20
DieselInputs	49
ElectricalLoad	53
Emissions	58
Model	61
ModelInputs	76
Production	77
Combustion	7
Diesel	35
Renewable	91
Solar	110
Tidal	121
Wave	135
Wind	150
ProductionInputs	89
RenewableInputs	98
Resources	99
SolarInputs	117
Storage	120
Lilon	60
TidalInputs	133
WaveInputs	147
WindInputs	160

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 . . .	19
Controller	A class which contains a various dispatch control logic. Intended to serve as a component class of Model	20
Diesel	A derived class of the Combustion branch of Production which models production using a diesel generator	35
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 . . .	49
ElectricalLoad	A class which contains time and electrical load data. Intended to serve as a component class of Model	53
Emissions	A structure which bundles the emitted masses of various emissions chemistries	58
Lilon	A derived class of Storage which models energy storage by way of lithium-ion batteries	60
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	61
ModelInputs	A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except <code>path_2_electrical_load_time_series</code> , for which a valid input must be provided)	76
Production	The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise	77
ProductionInputs	A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input	89

Renewable	The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy	91
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 . . .	98
Resources	A class which contains renewable resource data. Intended to serve as a component class of Model	99
Solar	A derived class of the Renewable branch of Production which models solar production	110
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	117
Storage	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	120
Tidal	A derived class of the Renewable branch of Production which models tidal production	121
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	133
Wave	A derived class of the Renewable branch of Production which models wave production	135
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	147
Wind	A derived class of the Renewable branch of Production which models wind production	150
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	160

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	163
header/ ElectricalLoad.h	
Header file the ElectricalLoad class	164
header/ Model.h	
Header file the Model class	165
header/ Resources.h	
Header file the Resources class	176
header/ std_includes.h	
Header file which simply batches together the usual, standard includes	177
header/Production/ Production.h	
Header file the Production class	169
header/Production/Combustion/ Combustion.h	
Header file the Combustion class	166
header/Production/Combustion/ Diesel.h	
Header file the Diesel class	168
header/Production/Renewable/ Renewable.h	
Header file the Renewable class	170
header/Production/Renewable/ Solar.h	
Header file the Solar class	171
header/Production/Renewable/ Tidal.h	
Header file the Tidal class	172
header/Production/Renewable/ Wave.h	
Header file the Wave class	173
header/Production/Renewable/ Wind.h	
Header file the Wind class	175
header/Storage/ Lilon.h	
Header file the Lilon class	178
header/Storage/ Storage.h	
Header file the Storage class	179
pybindings/ PYBIND11_PGM.cpp	
Python 3 bindings file for PGMcpp	179
source/ Controller.cpp	
Implementation file for the Controller class	181
source/ ElectricalLoad.cpp	
Implementation file for the ElectricalLoad class	182

source/ Model.cpp	
Implementation file for the Model class	182
source/ Resources.cpp	
Implementation file for the Resources class	187
source/Production/ Production.cpp	
Implementation file for the Production class	184
source/Production/Combustion/ Combustion.cpp	
Implementation file for the Combustion class	183
source/Production/Combustion/ Diesel.cpp	
Implementation file for the Diesel class	183
source/Production/Renewable/ Renewable.cpp	
Implementation file for the Renewable class	184
source/Production/Renewable/ Solar.cpp	
Implementation file for the Solar class	185
source/Production/Renewable/ Tidal.cpp	
Implementation file for the Tidal class	185
source/Production/Renewable/ Wave.cpp	
Implementation file for the Wave class	186
source/Production/Renewable/ Wind.cpp	
Implementation file for the Wind class	186
source/Storage/ Lilon.cpp	
Implementation file for the Lilon class	187
source/Storage/ Storage.cpp	
Implementation file for the Storage class	188
test/source/ test_Controller.cpp	
Testing suite for Controller class	217
test/source/ test_ElectricalLoad.cpp	
Testing suite for ElectricalLoad class	218
test/source/ test_Model.cpp	
Testing suite for Model class	221
test/source/ test_Resources.cpp	
Testing suite for Resources class	228
test/source/Production/ test_Production.cpp	
Testing suite for Production class	212
test/source/Production/Combustion/ test_Combustion.cpp	
Testing suite for Combustion class	188
test/source/Production/Combustion/ test_Diesel.cpp	
Testing suite for Diesel class	190
test/source/Production/Renewable/ test_Renewable.cpp	
Testing suite for Renewable class	196
test/source/Production/Renewable/ test_Solar.cpp	
Testing suite for Solar class	197
test/source/Production/Renewable/ test_Tidal.cpp	
Testing suite for Tidal class	201
test/source/Production/Renewable/ test_Wave.cpp	
Testing suite for Wave class	204
test/source/Production/Renewable/ test_Wind.cpp	
Testing suite for Wind class	208
test/source/Storage/ test_Lilon.cpp	
Testing suite for Lilon class	214
test/source/Storage/ test_Storage.cpp	
Testing suite for Storage class	216
test/utills/ testing_utils.cpp	
Header file for various PGMcpp testing utilities	234
test/utills/ testing_utils.h	
Header file for various PGMcpp testing utilities	240

Chapter 4

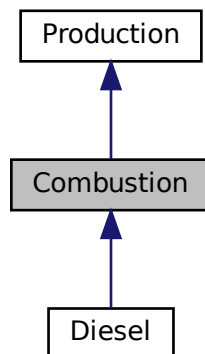
Class Documentation

4.1 Combustion Class Reference

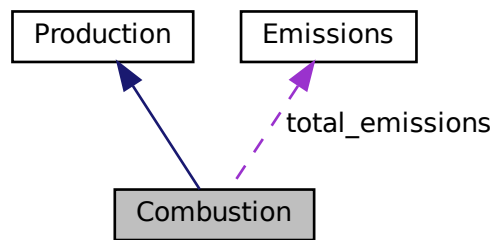
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

- [Combustion](#) (void)
Constructor (dummy) for the [Combustion](#) class.
- [Combustion](#) (int, double, [CombustionInputs](#))
Constructor (intended) for the [Combustion](#) class.
- void [computeFuelAndEmissions](#) (void)
Helper method to compute the total fuel consumption and emissions over the [Model](#) run.
- void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [requestProductionkW](#) (int, double, double)
- virtual double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- double [getFuelConsumptionL](#) (double, double)
Method which takes in production and returns volume of fuel burned over the given interval of time.
- [Emissions](#) [getEmissionskg](#) (double)
Method which takes in volume of fuel consumed and returns mass spectrum of resulting emissions.
- void [writeResults](#) (std::string, std::vector< double > *, int, int=-1)
Method which writes [Combustion](#) results to an output directory.
- virtual [~Combustion](#) (void)
Destructor for the [Combustion](#) class.

Public Attributes

- [CombustionType](#) type
The type ([CombustionType](#)) of the asset.
- double [fuel_cost_L](#)
The cost of fuel [1/L] (undefined currency).
- double [linear_fuel_slope_LkWh](#)
The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double [linear_fuel_intercept_LkWh](#)
The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.

- double [CO2_emissions_intensity_kgL](#)
Carbon dioxide (CO2) emissions intensity [kg/L].
- double [CO_emissions_intensity_kgL](#)
Carbon monoxide (CO) emissions intensity [kg/L].
- double [NOx_emissions_intensity_kgL](#)
Nitrogen oxide (NOx) emissions intensity [kg/L].
- double [SOx_emissions_intensity_kgL](#)
Sulfur oxide (SOx) emissions intensity [kg/L].
- double [CH4_emissions_intensity_kgL](#)
Methane (CH4) emissions intensity [kg/L].
- double [PM_emissions_intensity_kgL](#)
Particulate Matter (PM) emissions intensity [kg/L].
- double [total_fuel_consumed_L](#)
The total fuel consumed [L] over a model run.
- [Emissions total_emissions](#)
An [Emissions](#) structure for holding total emissions [kg].
- std::vector< double > [fuel_consumption_vec_L](#)
A vector of fuel consumed [L] over each modelling time step.
- std::vector< double > [fuel_cost_vec](#)
A vector of fuel costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are 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.

Private Member Functions

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

4.1.1 Detailed Description

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

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Combustion() [1/2]

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

Constructor (dummy) for the [Combustion](#) class.

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

4.1.2.2 Combustion() [2/2]

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

Constructor (intended) for the [Combustion](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>combustion_inputs</i>	A structure of Combustion constructor inputs.

```
93 :
94 Production(
95     n_points,
96     n_years,
97     combustion_inputs.production_inputs
98 )
99 {
100     // 1. check inputs
101     this->__checkInputs(combustion_inputs);
102
103     // 2. set attributes
104     this->fuel_cost_L = 0;
105
106     this->linear_fuel_slope_LkWh = 0;
107     this->linear_fuel_intercept_LkWh = 0;
108
109     this->CO2_emissions_intensity_kgL = 0;
110     this->CO_emissions_intensity_kgL = 0;
111     this->NOx_emissions_intensity_kgL = 0;
112     this->SOx_emissions_intensity_kgL = 0;
113     this->CH4_emissions_intensity_kgL = 0;
114     this->PM_emissions_intensity_kgL = 0;
115
116     this->total_fuel_consumed_L = 0;
117
118     this->fuel_consumption_vec_L.resize(this->n_points, 0);
119     this->fuel_cost_vec.resize(this->n_points, 0);
120
121     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
122     this->CO_emissions_vec_kg.resize(this->n_points, 0);
123     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
124     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
125     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
126     this->PM_emissions_vec_kg.resize(this->n_points, 0);
127
128     // 3. construction print
129     if (this->print_flag) {
130         std::cout << "Combustion object constructed at " << this << std::endl;
131     }
132
133     return;
134 } /* Combustion() */
```

4.1.2.3 ~Combustion()

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

Destructor for the [Combustion](#) class.

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

4.1.3 Member Function Documentation

4.1.3.1 __checkInputs()

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

Helper method to check inputs to the [Combustion](#) constructor.

Parameters

<i>combustion_inputs</i>	A structure of Combustion constructor inputs.
--------------------------	---

```
40 {
41     // ...
42
43     return;
44 } /* __checkInputs() */
```

4.1.3.2 __writeSummary()

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

Reimplemented in [Diesel](#).

```
85 {return;}
```

4.1.3.3 __writeTimeSeries()

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

Reimplemented in [Diesel](#).

```
90         {return;}

```

4.1.3.4 commit()

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

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

Parameters

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

```
236 {
237     // 1. invoke base class method
238     load_kW = Production::commit(
239         timestep,
240         dt_hrs,
241         production_kW,
242         load_kW
243     );
244
245
246     if (this->is_running) {
247         // 2. compute and record fuel consumption
248         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
249         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
250
251         // 3. compute and record emissions
252         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
253         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
254         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
255         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
256         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
257         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
258         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
259
260         // 4. incur fuel costs

```

```

261         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
262     }
263
264     return load_kW;
265 } /* commit() */

```

4.1.3.5 computeEconomics()

```

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

```

Helper method to compute key economic metrics for the [Model](#) run.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the <i>time_vec_hrs</i> attribute of the ElectricalLoad .
-------------------------	--

Reimplemented from [Production](#).

```

180 {
181     // 1. account for fuel costs in net present cost
182     double t_hrs = 0;
183     double real_discount_scalar = 0;
184
185     for (int i = 0; i < this->n_points; i++) {
186         t_hrs = time_vec_hrs_ptr->at(i);
187
188         real_discount_scalar = 1.0 / pow(
189             1 + this->real_discount_annual,
190             t_hrs / 8760
191         );
192
193         this->net_present_cost += real_discount_scalar * this->fuel_cost_vec[i];
194     }
195
196     // 2. invoke base class method
197     Production :: computeEconomics(time_vec_hrs_ptr);
198
199     return;
200 } /* computeEconomics() */

```

4.1.3.6 computeFuelAndEmissions()

```

void Combustion::computeFuelAndEmissions (
    void )

```

Helper method to compute the total fuel consumption and emissions over the [Model](#) run.

```

150 {
151     for (int i = 0; i < n_points; i++) {
152         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
153
154         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
155         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
156         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
157         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
158         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
159         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
160     }
161
162     return;
163 } /* computeFuelAndEmissions() */

```

4.1.3.7 getEmissionskg()

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

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

Parameters

<i>fuel_consumed_L</i>	The volume of fuel consumed [L].
------------------------	----------------------------------

Returns

A structure containing the mass spectrum of resulting emissions.

```
313                                     {
314     Emissions emissions;
315
316     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
317     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
318     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
319     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
320     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
321     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
322
323     return emissions;
324 } /* getEmissionskg() */
```

4.1.3.8 getFuelConsumptionL()

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

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

Parameters

<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
<i>production_kW</i>	The production [kW] of the asset in this timestep.

Returns

The volume of fuel consumed [L].

```
287 {
288     double fuel_consumed_L = (
289         this->linear_fuel_slope_LkWh * production_kW +
290         this->linear_fuel_intercept_LkWh * this->capacity_kW
291     ) * dt_hrs;
292
293     return fuel_consumed_L;
294 } /* getFuelConsumptionL() */
```

4.1.3.9 requestProductionkW()

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

Reimplemented in [Diesel](#).

```
130 {return 0;}
```

4.1.3.10 writeResults()

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

Method which writes [Combustion](#) results to an output directory.

Parameters

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

```
360 {
361     // 1. handle sentinel
362     if (max_lines < 0) {
363         max_lines = this->n_points;
364     }
365
366     // 2. create subdirectories
367     write_path += "Production/";
368     if (not std::filesystem::is_directory(write_path)) {
369         std::filesystem::create_directory(write_path);
370     }
371
372     write_path += "Combustion/";
373     if (not std::filesystem::is_directory(write_path)) {
374         std::filesystem::create_directory(write_path);
375     }
376
377     write_path += this->type_str;
378     write_path += "_";
379     write_path += std::to_string(int(ceil(this->capacity_kW)));
380     write_path += "kW_idx";
381     write_path += std::to_string(combustion_index);
382     write_path += "/";
383     std::filesystem::create_directory(write_path);
384
385     // 3. write summary
386     this->__writeSummary(write_path);
387
388     // 4. write time series
389     if (max_lines > this->n_points) {
390         max_lines = this->n_points;
391     }
392
393     if (max_lines > 0) {
394         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
395     }
```

```
396
397     return;
398 } /* writeResults() */
```

4.1.4 Member Data Documentation

4.1.4.1 CH4_emissions_intensity_kgL

```
double Combustion::CH4_emissions_intensity_kgL
```

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

4.1.4.2 CH4_emissions_vec_kg

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

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

4.1.4.3 CO2_emissions_intensity_kgL

```
double Combustion::CO2_emissions_intensity_kgL
```

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

4.1.4.4 CO2_emissions_vec_kg

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

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

4.1.4.5 CO_emissions_intensity_kgL

```
double Combustion::CO_emissions_intensity_kgL
```

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

4.1.4.6 CO_emissions_vec_kg

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

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

4.1.4.7 fuel_consumption_vec_L

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

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

4.1.4.8 fuel_cost_L

```
double Combustion::fuel_cost_L
```

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

4.1.4.9 fuel_cost_vec

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

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

4.1.4.10 linear_fuel_intercept_LkWh

```
double Combustion::linear_fuel_intercept_LkWh
```

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

4.1.4.11 linear_fuel_slope_LkWh

```
double Combustion::linear_fuel_slope_LkWh
```

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

4.1.4.12 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.13 NOx_emissions_vec_kg

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

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

4.1.4.14 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.15 PM_emissions_vec_kg

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

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

4.1.4.16 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.17 SOx_emissions_vec_kg

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

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

4.1.4.18 total_emissions

`Emissions` `Combustion::total_emissions`

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

4.1.4.19 total_fuel_consumed_L

`double` `Combustion::total_fuel_consumed_L`

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

4.1.4.20 type

`CombustionType` `Combustion::type`

The type (`CombustionType`) of the asset.

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

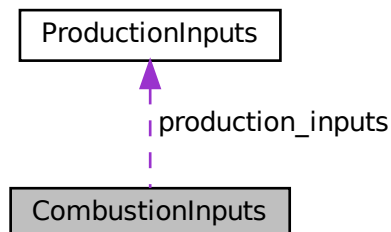
- `header/Production/Combustion/Combustion.h`
- `source/Production/Combustion/Combustion.cpp`

4.2 CombustionInputs Struct Reference

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

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

Collaboration diagram for `CombustionInputs`:



Public Attributes

- [ProductionInputs](#) `production_inputs`
An encapsulated [ProductionInputs](#) instance.

4.2.1 Detailed Description

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

4.2.2 Member Data Documentation

4.2.2.1 `production_inputs`

[ProductionInputs](#) `CombustionInputs::production_inputs`

An encapsulated [ProductionInputs](#) instance.

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

- `header/Production/Combustion/Combustion.h`

4.3 Controller Class Reference

A class which contains a various dispatch control logic. Intended to serve as a component class of [Model](#).

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

Public Member Functions

- [Controller](#) (void)
Constructor for the [Controller](#) class.
- void [setControlMode](#) ([ControlMode](#))
- void [init](#) ([ElectricalLoad](#) *, std::vector< [Renewable](#) * > *, [Resources](#) *, std::vector< [Combustion](#) * > *)
Method to initialize the [Controller](#) component of the [Model](#).
- void [applyDispatchControl](#) ([ElectricalLoad](#) *, std::vector< [Combustion](#) * > *, std::vector< [Renewable](#) * > *, std::vector< [Storage](#) * > *)
Method to apply dispatch control at every point in the modelling time series.
- void [clear](#) (void)
Method to clear all attributes of the [Controller](#) object.
- [~Controller](#) (void)
Destructor for the [Controller](#) class.

Public Attributes

- [ControlMode](#) `control_mode`
The *ControlMode* that is active in the *Model*.
- `std::string` `control_string`
A string describing the active *ControlMode*.
- `std::vector< double >` `net_load_vec_kW`
A vector of net load values [kW] at each point in the modelling time series. Net load is defined as load minus all available *Renewable* production.
- `std::vector< double >` `missed_load_vec_kW`
A vector of missed load values [kW] at each point in the modelling time series.
- `std::map< double, std::vector< bool > >` `combustion_map`
A map of all possible combustion states, for use in determining optimal dispatch.

Private Member Functions

- `void` `__computeNetLoad` (*ElectricalLoad* *, `std::vector< Renewable * >` *, *Resources* *)
Helper method to compute and populate the net load vector.
- `void` `__constructCombustionMap` (`std::vector< Combustion * >` *)
Helper method to construct a *Combustion* map, for use in determining.
- `void` `__applyLoadFollowingControl_CHARGING` (int, *ElectricalLoad* *, `std::vector< Combustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply load following control action for given timestep of the *Model* run when net load ≤ 0 .
- `void` `__applyLoadFollowingControl_DISCHARGING` (int, *ElectricalLoad* *, `std::vector< Combustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply load following control action for given timestep of the *Model* run when net load > 0 .
- `void` `__applyCycleChargingControl_CHARGING` (int, *ElectricalLoad* *, `std::vector< Combustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply cycle charging control action for given timestep of the *Model* run when net load ≤ 0 . Simply defaults to load following control.
- `void` `__applyCycleChargingControl_DISCHARGING` (int, *ElectricalLoad* *, `std::vector< Combustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply cycle charging control action for given timestep of the *Model* run when net load > 0 . Defaults to load following control if no depleted storage assets.
- `void` `__handleStorageCharging` (int, double, `std::list< Storage * >` *)
Helper method to handle the charging of the given *Storage* assets.
- `void` `__handleStorageCharging` (int, double, `std::vector< Storage * >` *)
Helper method to handle the charging of the given *Storage* assets.
- `double` `__getRenewableProduction` (int, double, *Renewable* *, *Resources* *)
Helper method to compute the production from the given *Renewable* asset at the given point in time.
- `double` `__handleCombustionDispatch` (int, double, double, `std::vector< Combustion * >` *, bool)
Helper method to handle the optimal dispatch of *Combustion* assets. Dispatches for 1.2x the received net load, so as to ensure a "20% spinning reserve". Dispatches a minimum number of *Combustion* assets, which then share the load proportional to their rated capacities.
- `double` `__handleStorageDischarging` (int, double, double, `std::list< Storage * >` *)
Helper method to handle the discharging of the given *Storage* assets.

4.3.1 Detailed Description

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

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Controller()

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

Constructor for the [Controller](#) class.

```
812 {
813     return;
814 } /* Controller() */
```

4.3.2.2 ~Controller()

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

Destructor for the [Controller](#) class.

```
1041 {
1042     this->clear();
1043
1044     return;
1045 } /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

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

Helper method to apply cycle charging control action for given timestep of the [Model](#) run when net load ≤ 0 . Simply defaults to load following control.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>storage_ptr_vec_ptr</i>	A pointer to the Storage pointer vector of the Model .

```

384 {
385     // 1. default to load following
386     this->__applyLoadFollowingControl_CHARGING(
387         timestep,
388         electrical_load_ptr,
389         combustion_ptr_vec_ptr,
390         renewable_ptr_vec_ptr,
391         storage_ptr_vec_ptr
392     );
393
394     return;
395 } /* __applyCycleChargingControl_CHARGING() */

```

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

```

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

```

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

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>storage_ptr_vec_ptr</i>	A pointer to the Storage pointer vector of the Model .

curtailment

```

434 {
435     // 1. get dt_hrs, net load
436     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
437     double net_load_kW = this->net_load_vec_kW[timestep];
438
439     // 2. partition Storage assets into depleted and non-depleted
440     std::list<Storage*> depleted_storage_ptr_list;
441     std::list<Storage*> nondepleted_storage_ptr_list;
442
443     Storage* storage_ptr;
444     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
445         storage_ptr = storage_ptr_vec_ptr->at(i);
446
447         //...
448     }
449
450     // 3. discharge non-depleted storage assets
451     net_load_kW = this->__handleStorageDischarging(
452         timestep,
453         dt_hrs,
454         net_load_kW,
455         nondepleted_storage_ptr_list
456     );
457
458     // 4. request optimal production from all Combustion assets
459     // default to load following if no depleted storage
460     if (depleted_storage_ptr_list.empty()) {
461         net_load_kW = this->__handleCombustionDispatch(
462             timestep,
463             dt_hrs,
464             net_load_kW,
465             combustion_ptr_vec_ptr,
466             false // is_cycle_charging

```

```

467     );
468 }
469
470 else {
471     net_load_kW = this->__handleCombustionDispatch(
472         timestep,
473         dt_hrs,
474         net_load_kW,
475         combustion_ptr_vec_ptr,
476         true // is_cycle_charging
477     );
478 }
479
480 // 5. attempt to charge depleted Storage assets using any and all available
481 // charge priority is Combustion, then Renewable
482 this->__handleStorageCharging(timestep, dt_hrs, depleted_storage_ptr_list);
483
484 // 6. record any missed load
485 if (net_load_kW > 1e-6) {
486     this->missed_load_vec_kW[timestep] = net_load_kW;
487 }
488
489 return;
490 }
491 /* __applyCycleChargingControl_DISCHARGING() */

```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```

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

```

Helper method to apply load following control action for given timestep of the [Model](#) run when net load ≤ 0 ;

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>storage_ptr_vec_ptr</i>	A pointer to the Storage pointer vector of the Model .

```

245 {
246     // 1. get dt_hrs, set net load
247     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
248     double net_load_kW = 0;
249
250     // 2. request zero production from all Combustion assets
251     this->__handleCombustionDispatch(
252         timestep,
253         dt_hrs,
254         net_load_kW,
255         combustion_ptr_vec_ptr,
256         false // is_cycle_charging
257     );
258
259     // 3. attempt to charge all Storage assets using any and all available curtailment
260     // charge priority is Combustion, then Renewable
261     this->__handleStorageCharging(timestep, dt_hrs, storage_ptr_vec_ptr);
262
263     return;
264 } /* __applyLoadFollowingControl_CHARGING() */

```


4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

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

Helper method to apply load following control action for given timestep of the [Model](#) run when net load > 0;.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>storage_ptr_vec_ptr</i>	A pointer to the Storage pointer vector of the Model .

curtailment

```
302 {
303     // 1. get dt_hrs, net load
304     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
305     double net_load_kW = this->net_load_vec_kW[timestep];
306
307     // 2. partition Storage assets into depleted and non-depleted
308     std::list<Storage*> depleted_storage_ptr_list;
309     std::list<Storage*> nondepleted_storage_ptr_list;
310
311     Storage* storage_ptr;
312     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
313         storage_ptr = storage_ptr_vec_ptr->at(i);
314
315         //...
316     }
317
318     // 3. discharge non-depleted storage assets
319     net_load_kW = this->__handleStorageDischarging(
320         timestep,
321         dt_hrs,
322         net_load_kW,
323         nondepleted_storage_ptr_list
324     );
325
326     // 4. request optimal production from all Combustion assets
327     net_load_kW = this->__handleCombustionDispatch(
328         timestep,
329         dt_hrs,
330         net_load_kW,
331         combustion_ptr_vec_ptr,
332         false // is_cycle_charging
333     );
334
335     // 5. attempt to charge depleted Storage assets using any and all available
336     // charge priority is Combustion, then Renewable
337     this->__handleStorageCharging(timestep, dt_hrs, depleted_storage_ptr_list);
338
339     // 6. record any missed load
340     if (net_load_kW > 1e-6) {
341         this->missed_load_vec_kW[timestep] = net_load_kW;
342     }
343
344     return;
345 }
346 } /* __applyLoadFollowingControl_DISCHARGING() */
```

4.3.3.5 __computeNetLoad()

```
void Controller::__computeNetLoad (
    ElectricalLoad * electrical_load_ptr,
```

```
std::vector< Renewable * > * renewable_ptr_vec_ptr,
Resources * resources_ptr ) [private]
```

Helper method to compute and populate the net load vector.

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

Parameters

<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>resources_ptr</i>	A pointer to the Resources component of the Model .

```
57 {
58     // 1. init
59     this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
60     this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
61
62     // 2. populate net load vector
63     double dt_hrs = 0;
64     double load_kW = 0;
65     double net_load_kW = 0;
66     double production_kW = 0;
67
68     Renewable* renewable_ptr;
69
70     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
71         dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
72         load_kW = electrical_load_ptr->load_vec_kW[i];
73         net_load_kW = load_kW;
74
75         for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {
76             renewable_ptr = renewable_ptr_vec_ptr->at(j);
77
78             production_kW = this->__getRenewableProduction(
79                 i,
80                 dt_hrs,
81                 renewable_ptr,
82                 resources_ptr
83             );
84
85             load_kW = renewable_ptr->commit(
86                 i,
87                 dt_hrs,
88                 production_kW,
89                 load_kW
90             );
91
92             net_load_kW -= production_kW;
93         }
94         this->net_load_vec_kW[i] = net_load_kW;
95     }
96
97     return;
98 }
99 /* __computeNetLoad() */
```

4.3.3.6 __constructCombustionMap()

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

Helper method to construct a [Combustion](#) map, for use in determining.

Parameters

<code>combustion_ptr_vec_ptr</code>	A pointer to the Combustion pointer vector of the Model .
-------------------------------------	---

```

121 {
122     // 1. get state table dimensions
123     int n_cols = combustion_ptr_vec_ptr->size();
124     int n_rows = pow(2, n_cols);
125
126     // 2. init state table (all possible on/off combinations)
127     std::vector<std::vector<bool>> state_table;
128     state_table.resize(n_rows, {});
129
130     int x = 0;
131     for (int i = 0; i < n_rows; i++) {
132         state_table[i].resize(n_cols, false);
133
134         x = i;
135         for (int j = 0; j < n_cols; j++) {
136             if (x % 2 == 0) {
137                 state_table[i][j] = true;
138             }
139             x /= 2;
140         }
141     }
142
143     // 3. construct combustion map (handle duplicates by keeping rows with minimum
144     //     trues)
145     double total_capacity_kW = 0;
146     int truth_count = 0;
147     int current_truth_count = 0;
148
149     for (int i = 0; i < n_rows; i++) {
150         total_capacity_kW = 0;
151         truth_count = 0;
152         current_truth_count = 0;
153
154         for (int j = 0; j < n_cols; j++) {
155             if (state_table[i][j]) {
156                 total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
157                 truth_count++;
158             }
159         }
160
161         if (this->combustion_map.count(total_capacity_kW) > 0) {
162             for (int j = 0; j < n_cols; j++) {
163                 if (this->combustion_map[total_capacity_kW][j]) {
164                     current_truth_count++;
165                 }
166             }
167
168             if (truth_count < current_truth_count) {
169                 this->combustion_map.erase(total_capacity_kW);
170             }
171         }
172
173         this->combustion_map.insert(
174             std::pair<double, std::vector<bool>> (
175                 total_capacity_kW,
176                 state_table[i]
177             )
178         );
179     }
180
181     // 4. test print
182     /*
183     std::cout << std::endl;
184
185     std::cout << "\t\t";
186     for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
187         std::cout << combustion_ptr_vec_ptr->at(i)->capacity_kW << "\t";
188     }
189     std::cout << std::endl;
190
191     std::map<double, std::vector<bool>>::iterator iter;
192     for (
193         iter = this->combustion_map.begin();
194         iter != this->combustion_map.end();
195         iter++
196     ) {
197         std::cout << iter->first << "\t\t";
198
199         for (size_t i = 0; i < iter->second.size(); i++) {
200             std::cout << iter->second[i] << "\t";
201         }
202         std::cout << "]" << std::endl;

```

```

203     }
204     */
205
206     return;
207 } /* __constructCombustionTable() */

```

4.3.3.7 __getRenewableProduction()

```

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

```

Helper method to compute the production from the given [Renewable](#) asset at the given point in time.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>renewable_ptr</i>	A pointer to the Renewable asset.
<i>resources_ptr</i>	A pointer to the Resources component of the Model .

Returns

The production [kW] of the [Renewable](#) asset.

```

595 {
596     double production_kW = 0;
597
598     switch (renewable_ptr->type) {
599         case (RenewableType :: SOLAR): {
600             production_kW = renewable_ptr->computeProductionkW(
601                 timestep,
602                 dt_hrs,
603                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
604             );
605             break;
606         }
607
608         case (RenewableType :: TIDAL): {
609             production_kW = renewable_ptr->computeProductionkW(
610                 timestep,
611                 dt_hrs,
612                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
613             );
614             break;
615         }
616
617         case (RenewableType :: WAVE): {
618             production_kW = renewable_ptr->computeProductionkW(
619                 timestep,
620                 dt_hrs,
621                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0],
622                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1]
623             );
624             break;
625         }
626
627         case (RenewableType :: WIND): {
628             production_kW = renewable_ptr->computeProductionkW(
629                 timestep,
630                 dt_hrs,
631                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
632             );
633         }
634     }
635 }

```

```

636
637         break;
638     }
639
640     default: {
641         std::string error_str = "ERROR: Controller::__getRenewableProduction(): ";
642         error_str += "renewable type ";
643         error_str += std::to_string(renewable_ptr->type);
644         error_str += " not recognized";
645
646         #ifdef _WIN32
647             std::cout << error_str << std::endl;
648         #endif
649
650         throw std::runtime_error(error_str);
651
652         break;
653     }
654 }
655
656 return production_kW;
657 } /* __getRenewableProduction() */

```

4.3.3.8 __handleCombustionDispatch()

```

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

```

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

bool is_cycle_charging)

Parameters

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

Returns

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

```

699 {
700     // 1. get minimal Combustion dispatch
701     double target_production_kW = 1.2 * net_load_kW;
702     double total_capacity_kW = 0;
703
704     std::map<double, std::vector<bool>::iterator> iter = this->combustion_map.begin();
705     while (iter != std::prev(this->combustion_map.end(), 1)) {
706         if (target_production_kW <= total_capacity_kW) {
707             break;
708         }
709
710         iter++;
711         total_capacity_kW = iter->first;
712     }

```

```

713
714 // 2. share load proportionally (by rated capacity) over active diesels
715 Combustion* combustion_ptr;
716 double production_kW = 0;
717 double request_kW = 0;
718 double _net_load_kW = net_load_kW;
719
720 for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
721     combustion_ptr = combustion_ptr_vec_ptr->at(i);
722
723     if (total_capacity_kW > 0) {
724         request_kW =
725             int(this->combustion_map[total_capacity_kW][i]) *
726             net_load_kW *
727             (combustion_ptr->capacity_kW / total_capacity_kW);
728     }
729
730     else {
731         request_kW = 0;
732     }
733
734     if (is_cycle_charging and request_kW > 0) {
735         if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
736             request_kW = 0.85 * combustion_ptr->capacity_kW;
737         }
738     }
739
740     production_kW = combustion_ptr->requestProductionkW(
741         timestep,
742         dt_hrs,
743         request_kW
744     );
745
746     _net_load_kW = combustion_ptr->commit(
747         timestep,
748         dt_hrs,
749         production_kW,
750         _net_load_kW
751     );
752 }
753
754 return _net_load_kW;
755 } /* __handleCombustionDispatch() */

```

4.3.3.9 __handleStorageCharging() [1/2]

```

void Controller::__handleStorageCharging (
    int timestep,
    double dt_hrs,
    std::list< Storage * > storage_ptr_list ) [private]

```

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

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>storage_ptr_list</i>	A list of pointers to the Storage assets that are to be charged.

```

521 {
522     //...
523
524     return;
525 } /* __handleStorageCharging() */

```

4.3.3.10 __handleStorageCharging() [2/2]

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

Helper method to handle the charging of the given [Storage](#) assets.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>storage_ptr_vec_ptr</i>	A pointer to a vector of pointers to the Storage assets that are to be charged.

```
555 {
556     ///...
557
558     return;
559 } /* __handleStorageCharging() */
```

4.3.3.11 __handleStorageDischarging()

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

Helper method to handle the discharging of the given [Storage](#) assets.

Parameters

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

Returns

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

```
789 {
790     ///...
791
792     return net_load_kW;
793 } /* __handleStorageDischarging() */
```

4.3.3.12 applyDispatchControl()

```
void Controller::applyDispatchControl (
    ElectricalLoad * electrical_load_ptr,
```

```

std::vector< Combustion * > * combustion_ptr_vec_ptr,
std::vector< Renewable * > * renewable_ptr_vec_ptr,
std::vector< Storage * > * storage_ptr_vec_ptr )

```

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

Parameters

<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>storage_ptr_vec_ptr</i>	A pointer to the Storage pointer vector of the Model .

```

936 {
937     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
938         switch (this->control_mode) {
939             case (ControlMode :: LOAD_FOLLOWING): {
940                 if (this->net_load_vec_kW[i] <= 0) {
941                     this->__applyLoadFollowingControl_CHARGING(
942                         i,
943                         electrical_load_ptr,
944                         combustion_ptr_vec_ptr,
945                         renewable_ptr_vec_ptr,
946                         storage_ptr_vec_ptr
947                     );
948                 }
949             }
950             else {
951                 this->__applyLoadFollowingControl_DISCHARGING(
952                     i,
953                     electrical_load_ptr,
954                     combustion_ptr_vec_ptr,
955                     renewable_ptr_vec_ptr,
956                     storage_ptr_vec_ptr
957                 );
958             }
959             break;
960         }
961     }
962
963     case (ControlMode :: CYCLE_CHARGING): {
964         if (this->net_load_vec_kW[i] <= 0) {
965             this->__applyCycleChargingControl_CHARGING(
966                 i,
967                 electrical_load_ptr,
968                 combustion_ptr_vec_ptr,
969                 renewable_ptr_vec_ptr,
970                 storage_ptr_vec_ptr
971             );
972         }
973     }
974     else {
975         this->__applyCycleChargingControl_DISCHARGING(
976             i,
977             electrical_load_ptr,
978             combustion_ptr_vec_ptr,
979             renewable_ptr_vec_ptr,
980             storage_ptr_vec_ptr
981         );
982     }
983     break;
984 }
985
986 default: {
987     std::string error_str = "ERROR: Controller :: applyDispatchControl(): ";
988     error_str += "control mode ";
989     error_str += std::to_string(this->control_mode);
990     error_str += " not recognized";
991
992     #ifdef _WIN32
993         std::cout << error_str << std::endl;
994     #endif
995
996     throw std::runtime_error(error_str);
997     break;
998 }
999 }
1000 }
1001 }
1002 }

```



```

1003
1004     return;
1005 }    /* applyDispatchControl() */

```

4.3.3.13 clear()

```

void Controller::clear (
    void )

```

Method to clear all attributes of the [Controller](#) object.

```

1020 {
1021     this->net_load_vec_kW.clear();
1022     this->missed_load_vec_kW.clear();
1023     this->combustion_map.clear();
1024
1025     return;
1026 }    /* clear() */

```

4.3.3.14 init()

```

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

```

Method to initialize the [Controller](#) component of the [Model](#).

Parameters

<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .
<i>resources_ptr</i>	A pointer to the Resources component of the Model .
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .

```

895 {
896     // 1. compute net load
897     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);
898
899     // 2. construct Combustion table
900     this->__constructCombustionMap(combustion_ptr_vec_ptr);
901
902     return;
903 }    /* init() */

```

4.3.3.15 setControlMode()

```

void Controller::setControlMode (
    ControlMode control_mode )

```

Parameters

<i>control_mode</i>	The ControlMode which is to be active in the Controller .
---------------------	---

```

829 {
830     this->control_mode = control_mode;
831
832     switch(control_mode) {
833         case (ControlMode :: LOAD_FOLLOWING): {
834             this->control_string = "LOAD_FOLLOWING";
835
836             break;
837         }
838
839         case (ControlMode :: CYCLE_CHARGING): {
840             this->control_string = "CYCLE_CHARGING";
841
842             break;
843         }
844
845         default: {
846             std::string error_str = "ERROR: Controller :: setControlMode(): ";
847             error_str += "control mode ";
848             error_str += std::to_string(control_mode);
849             error_str += " not recognized";
850
851             #ifdef _WIN32
852                 std::cout << error_str << std::endl;
853             #endif
854
855             throw std::runtime_error(error_str);
856
857             break;
858         }
859     }
860
861     return;
862 } /* setControlMode() */

```

4.3.4 Member Data Documentation

4.3.4.1 combustion_map

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

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

4.3.4.2 control_mode

```
ControlMode Controller::control_mode
```

The ControlMode that is active in the [Model](#).

4.3.4.3 control_string

```
std::string Controller::control_string
```

A string describing the active ControlMode.

4.3.4.4 missed_load_vec_kW

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

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

4.3.4.5 net_load_vec_kW

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

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

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

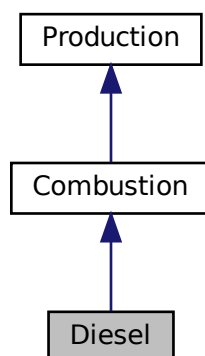
- header/[Controller.h](#)
- source/[Controller.cpp](#)

4.4 Diesel Class Reference

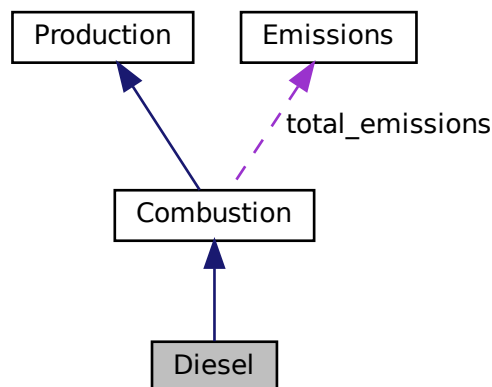
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

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

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

- [Diesel](#) (void)
Constructor (dummy) for the [Diesel](#) class.
- [Diesel](#) (int, double, [DieselInputs](#))
Constructor (intended) for the [Diesel](#) class.
- double [requestProductionkW](#) (int, double, double)
Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Diesel](#) (void)
Destructor for the [Diesel](#) class.

Public Attributes

- double [minimum_load_ratio](#)
The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.
- double [minimum_runtime_hrs](#)
The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.
- double [time_since_last_start_hrs](#)
The time that has elapsed [hrs] since the last start of the asset.

Private Member Functions

- void `__checkInputs` ([DieselInputs](#))
Helper method to check inputs to the [Diesel](#) constructor.
- void `__handleStartStop` (int, double, double)
Helper method (private) to handle the starting/stopping of the diesel generator. The minimum runtime constraint is enforced in this method.
- double `__getGenericFuelSlope` (void)
Helper method to generate a generic, linearized fuel consumption slope for a diesel generator.
- double `__getGenericFuelIntercept` (void)
Helper method to generate a generic, linearized fuel consumption intercept for a diesel generator.
- double `__getGenericCapitalCost` (void)
Helper method to generate a generic diesel generator capital cost.
- double `__getGenericOpMaintCost` (void)
Helper method (private) to generate a generic diesel generator operation and maintenance cost. This is a cost incurred per unit energy produced.
- void `__writeSummary` (std::string)
Helper method to write summary results for [Diesel](#).
- void `__writeTimeSeries` (std::string, std::vector< double > *, int=-1)
Helper method to write time series results for [Diesel](#).

4.4.1 Detailed Description

A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Diesel() [1/2]

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

Constructor (dummy) for the [Diesel](#) class.

```
571 {
572     return;
573 } /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

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

Constructor (intended) for the [Diesel](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>diesel_inputs</i>	A structure of Diesel constructor inputs.

```

601 :
602 Combustion(
603     n_points,
604     n_years,
605     diesel_inputs.combustion_inputs
606 )
607 {
608     // 1. check inputs
609     this->__checkInputs(diesel_inputs);
610
611     // 2. set attributes
612     this->type = CombustionType :: DIESEL;
613     this->type_str = "DIESEL";
614
615     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
616
617     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
618
619     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
620     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
621     this->time_since_last_start_hrs = 0;
622
623     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
624     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
625     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
626     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
627     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
628     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
629
630     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
631         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
632     }
633
634     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
635         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
636     }
637
638     if (diesel_inputs.capital_cost < 0) {
639         this->capital_cost = this->__getGenericCapitalCost();
640     }
641
642     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
643         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
644     }
645
646     if (not this->is_sunk) {
647         this->capital_cost_vec[0] = this->capital_cost;
648     }
649
650     // 3. construction print
651     if (this->print_flag) {
652         std::cout << "Diesel object constructed at " << this << std::endl;
653     }
654
655     return;
656 } /* Diesel() */

```

4.4.2.3 ~Diesel()

```

Diesel::~Diesel (
    void )

```

Destructor for the [Diesel](#) class.

```

785 {
786     // 1. destruction print
787     if (this->print_flag) {
788         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
789     }
790
791     return;
792 } /* ~Diesel() */

```

4.4.3 Member Function Documentation

4.4.3.1 `__checkInputs()`

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

Helper method to check inputs to the `Diesel` constructor.

Parameters

<code>diesel_inputs</code>	A structure of <code>Diesel</code> constructor inputs.
----------------------------	--

```
39 {
40     // 1. check fuel_cost_L
41     if (diesel_inputs.fuel_cost_L < 0) {
42         std::string error_str = "ERROR: Diesel(): ";
43         error_str += "DieselInputs::fuel_cost_L must be >= 0";
44
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check CO2_emissions_intensity_kgL
53     if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
54         std::string error_str = "ERROR: Diesel(): ";
55         error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";
56
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     // 3. check CO_emissions_intensity_kgL
65     if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
66         std::string error_str = "ERROR: Diesel(): ";
67         error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";
68
69         #ifdef _WIN32
70             std::cout << error_str << std::endl;
71         #endif
72
73         throw std::invalid_argument(error_str);
74     }
75
76     // 4. check NOx_emissions_intensity_kgL
77     if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
78         std::string error_str = "ERROR: Diesel(): ";
79         error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";
80
81         #ifdef _WIN32
82             std::cout << error_str << std::endl;
83         #endif
84
85         throw std::invalid_argument(error_str);
86     }
87
88     // 5. check SOx_emissions_intensity_kgL
89     if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
90         std::string error_str = "ERROR: Diesel(): ";
91         error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";
92
93         #ifdef _WIN32
94             std::cout << error_str << std::endl;
95         #endif
96
97         throw std::invalid_argument(error_str);
98     }
99 }
```

```

100 // 6. check CH4_emissions_intensity_kgL
101 if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
102     std::string error_str = "ERROR: Diesel(): ";
103     error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";
104
105     #ifdef _WIN32
106         std::cout << error_str << std::endl;
107     #endif
108
109     throw std::invalid_argument(error_str);
110 }
111
112 // 7. check PM_emissions_intensity_kgL
113 if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
114     std::string error_str = "ERROR: Diesel(): ";
115     error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";
116
117     #ifdef _WIN32
118         std::cout << error_str << std::endl;
119     #endif
120
121     throw std::invalid_argument(error_str);
122 }
123
124 // 8. check minimum_load_ratio
125 if (diesel_inputs.minimum_load_ratio < 0) {
126     std::string error_str = "ERROR: Diesel(): ";
127     error_str += "DieselInputs::minimum_load_ratio must be >= 0";
128
129     #ifdef _WIN32
130         std::cout << error_str << std::endl;
131     #endif
132
133     throw std::invalid_argument(error_str);
134 }
135
136 // 9. check minimum_runtime_hrs
137 if (diesel_inputs.minimum_runtime_hrs < 0) {
138     std::string error_str = "ERROR: Diesel(): ";
139     error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";
140
141     #ifdef _WIN32
142         std::cout << error_str << std::endl;
143     #endif
144
145     throw std::invalid_argument(error_str);
146 }
147
148 // 10. check replace_running_hrs
149 if (diesel_inputs.replace_running_hrs <= 0) {
150     std::string error_str = "ERROR: Diesel(): ";
151     error_str += "DieselInputs::replace_running_hrs must be > 0";
152
153     #ifdef _WIN32
154         std::cout << error_str << std::endl;
155     #endif
156
157     throw std::invalid_argument(error_str);
158 }
159
160 return;
161 } /* __checkInputs() */

```

4.4.3.2 __getGenericCapitalCost()

```

double Diesel::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic diesel generator capital cost.

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

Returns

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

```
238 {
239     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.425) + 800;
240
241     return capital_cost_per_kW * this->capacity_kW;
242 } /* __getGenericCapitalCost() */
```

4.4.3.3 __getGenericFuelIntercept()

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

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

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

Ref: [HOMER \[2023c\]](#)

Ref: [HOMER \[2023d\]](#)

Returns

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

```
213 {
214     double linear_fuel_intercept_LkWh = 0.0940 * pow(this->capacity_kW, -0.2735);
215
216     return linear_fuel_intercept_LkWh;
217 } /* __getGenericFuelIntercept() */
```

4.4.3.4 __getGenericFuelSlope()

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

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

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

Ref: [HOMER \[2023c\]](#)

Ref: [HOMER \[2023e\]](#)

Returns

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

```
185 {
186     double linear_fuel_slope_LkWh = 0.4234 * pow(this->capacity_kW, -0.1012);
187
188     return linear_fuel_slope_LkWh;
189 } /* __getGenericFuelSlope() */
```

4.4.3.5 `__getGenericOpMaintCost()`

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

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

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

Returns

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

```
266 {
267     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
268
269     return operation_maintenance_cost_kWh;
270 } /* __getGenericOpMaintCost() */
```

4.4.3.6 `__handleStartStop()`

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

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

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>production_kW</i>	The current rate of production [kW] of the generator.

```
300 {
301     /*
302     * Helper method (private) to handle the starting/stopping of the diesel
303     * generator. The minimum runtime constraint is enforced in this method.
304     */
305
306     if (this->is_running) {
307         // handle stopping
308         if (
309             production_kW <= 0 and
310             this->time_since_last_start_hrs >= this->minimum_runtime_hrs
311         ) {
312             this->is_running = false;
313         }
314     }
315
316     else {
317         // handle starting
318         if (production_kW > 0) {
319             this->is_running = true;
320             this->n_starts++;
321             this->time_since_last_start_hrs = 0;
322         }
323     }
324 }
```

```

325     return;
326 } /* __handleStartStop() */

```

4.4.3.7 __writeSummary()

```

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

```

Helper method to write summary results for [Diesel](#).

Parameters

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

Reimplemented from [Combustion](#).

```

345 {
346     // 1. create filestream
347     write_path += "summary_results.md";
348     std::ofstream ofs;
349     ofs.open(write_path, std::ofstream::out);
350
351     // 2. write to summary results (markdown)
352     ofs << "# ";
353     ofs << std::to_string(int(ceil(this->capacity_kW)));
354     ofs << " kW DIESEL Summary Results\n";
355     ofs << "\n-----\n\n";
356
357     // 2.1. Production attributes
358     ofs << "## Production Attributes\n";
359     ofs << "\n";
360
361     ofs << "Capacity: " << this->capacity_kW << "kW \n";
362     ofs << "\n";
363
364     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
365     ofs << "Capital Cost: " << this->capital_cost << " \n";
366     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
367         << " per kWh produced \n";
368     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
369         << " \n";
370     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
371         << " \n";
372     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
373     ofs << "\n";
374
375     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
376     ofs << "\n-----\n\n";
377
378     // 2.2. Combustion attributes
379     ofs << "## Combustion Attributes\n";
380     ofs << "\n";
381
382     ofs << "Fuel Cost: " << this->fuel_cost_L << " per L \n";
383     ofs << "\n";
384
385     ofs << "Linear Fuel Slope: " << this->linear_fuel_slope_LkWh << " L/kWh \n";
386     ofs << "Linear Fuel Intercept Coefficient: " << this->linear_fuel_intercept_LkWh
387         << " L/kWh \n";
388     ofs << "\n";
389
390     ofs << "Carbon Dioxide (CO2) Emissions Intensity: "
391         << this->CO2_emissions_intensity_kgL << " kg/L \n";
392
393     ofs << "Carbon Monoxide (CO) Emissions Intensity: "
394         << this->CO_emissions_intensity_kgL << " kg/L \n";
395
396     ofs << "Nitrogen Oxides (NOx) Emissions Intensity: "
397         << this->NOx_emissions_intensity_kgL << " kg/L \n";
398
399     ofs << "Sulfur Oxides (SOx) Emissions Intensity: "
400         << this->SOx_emissions_intensity_kgL << " kg/L \n";

```

```

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

```

4.4.3.8 __writeTimeSeries()

```

void Diesel::__writeTimeSeries (
    std::string write_path,

```

```
std::vector< double > * time_vec_hrs_ptr,
int max_lines = -1 ) [private], [virtual]
```

Helper method to write time series results for [Diesel](#).

Parameters

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

Reimplemented from [Combustion](#).

```
505 {
506     // 1. create filestream
507     write_path += "time_series_results.csv";
508     std::ofstream ofs;
509     ofs.open(write_path, std::ofstream::out);
510
511     // 2. write time series results (comma separated value)
512     ofs << "Time (since start of data) [hrs],";
513     ofs << "Production [kW],";
514     ofs << "Dispatch [kW],";
515     ofs << "Storage [kW],";
516     ofs << "Curtailement [kW],";
517     ofs << "Is Running (N = 0 / Y = 1),";
518     ofs << "Fuel Consumption [L],";
519     ofs << "Fuel Cost (actual),";
520     ofs << "Carbon Dioxide (CO2) Emissions [kg],";
521     ofs << "Carbon Monoxide (CO) Emissions [kg],";
522     ofs << "Nitrogen Oxides (NOx) Emissions [kg],";
523     ofs << "Sulfur Oxides (SOx) Emissions [kg],";
524     ofs << "Methane (CH4) Emissions [kg],";
525     ofs << "Particulate Matter (PM) Emissions [kg],";
526     ofs << "Capital Cost (actual),";
527     ofs << "Operation and Maintenance Cost (actual),";
528     ofs << "\n";
529
530     for (int i = 0; i < max_lines; i++) {
531         ofs << time_vec_hrs_ptr->at(i) << ",";
532         ofs << this->production_vec_kW[i] << ",";
533         ofs << this->dispatch_vec_kW[i] << ",";
534         ofs << this->storage_vec_kW[i] << ",";
535         ofs << this->curtailment_vec_kW[i] << ",";
536         ofs << this->is_running_vec[i] << ",";
537         ofs << this->fuel_consumption_vec_L[i] << ",";
538         ofs << this->fuel_cost_vec[i] << ",";
539         ofs << this->CO2_emissions_vec_kg[i] << ",";
540         ofs << this->CO_emissions_vec_kg[i] << ",";
541         ofs << this->NOx_emissions_vec_kg[i] << ",";
542         ofs << this->SOx_emissions_vec_kg[i] << ",";
543         ofs << this->CH4_emissions_vec_kg[i] << ",";
544         ofs << this->PM_emissions_vec_kg[i] << ",";
545         ofs << this->capital_cost_vec[i] << ",";
546         ofs << this->operation_maintenance_cost_vec[i] << ",";
547         ofs << "\n";
548     }
549
550     ofs.close();
551     return;
552 } /* __writeTimeSeries() */
```

4.4.3.9 commit()

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

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

Parameters

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

Returns

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

Reimplemented from [Combustion](#).

```

743 {
744     // 1. handle start/stop, enforce minimum runtime constraint
745     this->__handleStartStop(timestep, dt_hrs, production_kW);
746
747     // 2. invoke base class method
748     load_kW = Combustion::commit(
749         timestep,
750         dt_hrs,
751         production_kW,
752         load_kW
753     );
754
755     if (this->is_running) {
756         // 3. log time since last start
757         this->time_since_last_start_hrs += dt_hrs;
758
759         // 4. correct operation and maintenance costs (should be non-zero if idling)
760         if (production_kW <= 0) {
761             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
762
763             double operation_maintenance_cost =
764                 this->operation_maintenance_cost_kWh * produced_kWh;
765             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
766         }
767     }
768
769     return load_kW;
770 } /* commit() */

```

4.4.3.10 requestProductionkW()

```

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

```

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

Parameters

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

Returns

The production [kW] delivered by the diesel generator.

Reimplemented from [Combustion](#).

```
688 {
689     // 1. return on request of zero
690     if (request_kW <= 0) {
691         return 0;
692     }
693
694     double deliver_kW = request_kW;
695
696     // 2. enforce capacity constraint
697     if (deliver_kW > this->capacity_kW) {
698         deliver_kW = this->capacity_kW;
699     }
700
701     // 3. enforce minimum load ratio
702     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
703         deliver_kW = this->minimum_load_ratio * this->capacity_kW;
704     }
705
706     return deliver_kW;
707 } /* requestProductionkW() */
```

4.4.4 Member Data Documentation

4.4.4.1 minimum_load_ratio

```
double Diesel::minimum_load_ratio
```

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

4.4.4.2 minimum_runtime_hrs

```
double Diesel::minimum_runtime_hrs
```

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

4.4.4.3 time_since_last_start_hrs

```
double Diesel::time_since_last_start_hrs
```

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

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

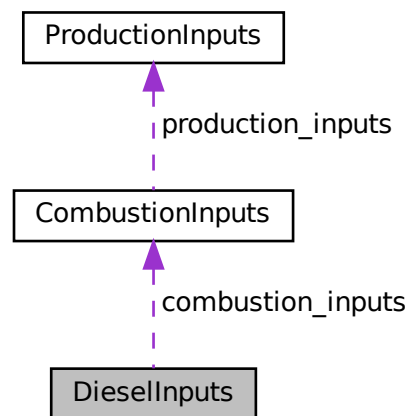
- header/Production/Combustion/[Diesel.h](#)
- source/Production/Combustion/[Diesel.cpp](#)

4.5 DieselInputs Struct Reference

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

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

Collaboration diagram for DieselInputs:



Public Attributes

- [CombustionInputs combustion_inputs](#)
An encapsulated [CombustionInputs](#) instance.
- double [replace_running_hrs](#) = 30000
The number of running hours after which the asset must be replaced. Overwrites the [ProductionInputs](#) attribute.
- double [capital_cost](#) = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double [operation_maintenance_cost_kWh](#) = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double [fuel_cost_L](#) = 1.70
The cost of fuel [1/L] (undefined currency).
- double [minimum_load_ratio](#) = 0.2
The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.
- double [minimum_runtime_hrs](#) = 4
The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.
- double [linear_fuel_slope_LkWh](#) = -1

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

- double [linear_fuel_intercept_LkWh](#) = -1

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

- double [CO2_emissions_intensity_kgL](#) = 2.7

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

- double [CO_emissions_intensity_kgL](#) = 0.0178

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

- double [NOx_emissions_intensity_kgL](#) = 0.0014

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

- double [SOx_emissions_intensity_kgL](#) = 0.0042

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

- double [CH4_emissions_intensity_kgL](#) = 0.0007

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

- double [PM_emissions_intensity_kgL](#) = 0.0001

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

4.5.1 Detailed Description

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

Ref: [HOMER \[2023c\]](#)

Ref: [HOMER \[2023d\]](#)

Ref: [HOMER \[2023e\]](#)

Ref: docs/refs/diesel_emissions_ref_1.pdf

Ref: docs/refs/diesel_emissions_ref_2.pdf

4.5.2 Member Data Documentation

4.5.2.1 capital_cost

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

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

4.5.2.2 CH4_emissions_intensity_kgL

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

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

4.5.2.3 CO2_emissions_intensity_kgL

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

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

4.5.2.4 CO_emissions_intensity_kgL

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

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

4.5.2.5 combustion_inputs

```
CombustionInputs DieselInputs::combustion_inputs
```

An encapsulated [CombustionInputs](#) instance.

4.5.2.6 fuel_cost_L

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

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

4.5.2.7 linear_fuel_intercept_LkWh

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

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

4.5.2.8 linear_fuel_slope_LkWh

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

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

4.5.2.9 minimum_load_ratio

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

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

4.5.2.10 minimum_runtime_hrs

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

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

4.5.2.11 NOx_emissions_intensity_kgL

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

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

4.5.2.12 operation_maintenance_cost_kWh

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

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

4.5.2.13 PM_emissions_intensity_kgL

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

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

4.5.2.14 replace_running_hrs

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

The number of running hours after which the asset must be replaced. Overwrites the [ProductionInputs](#) attribute.

4.5.2.15 SOx_emissions_intensity_kgL

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

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

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

- header/Production/Combustion/[Diesel.h](#)

4.6 ElectricalLoad Class Reference

A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

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

Public Member Functions

- [ElectricalLoad](#) (void)
Constructor (dummy) for the [ElectricalLoad](#) class.
- [ElectricalLoad](#) (std::string)
Constructor (intended) for the [ElectricalLoad](#) class.
- void [readLoadData](#) (std::string)
Method to read electrical load data into an already existing [ElectricalLoad](#) object. Clears and overwrites any existing attribute values.
- void [clear](#) (void)
Method to clear all attributes of the [ElectricalLoad](#) object.
- [~ElectricalLoad](#) (void)
Destructor for the [ElectricalLoad](#) class.

Public Attributes

- int [n_points](#)
The number of points in the modelling time series.
- double [n_years](#)
The number of years being modelled (inferred from [time_vec_hrs](#)).
- double [min_load_kW](#)
The minimum [kW] of the given electrical load time series.
- double [mean_load_kW](#)
The mean, or average, [kW] of the given electrical load time series.
- double [max_load_kW](#)
The maximum [kW] of the given electrical load time series.
- std::string [path_2_electrical_load_time_series](#)
A string defining the path (either relative or absolute) to the given electrical load time series.
- std::vector< double > [time_vec_hrs](#)
A vector to hold a given sequence of model times [hrs]. This defines the modelling time series.
- std::vector< double > [dt_vec_hrs](#)
A vector to hold a sequence of model time deltas [hrs].
- std::vector< double > [load_vec_kW](#)
A vector to hold a given sequence of electrical load values [kW].

4.6.1 Detailed Description

A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

4.6.2 Constructor & Destructor Documentation

4.6.2.1 ElectricalLoad() [1/2]

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

Constructor (dummy) for the [ElectricalLoad](#) class.

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

4.6.2.2 ElectricalLoad() [2/2]

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

Constructor (intended) for the [ElectricalLoad](#) class.

Parameters

<i>path_2_electrical_load_time_series</i>	A string defining the path (either relative or absolute) to the given electrical load time series.
---	--

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

4.6.2.3 ~ElectricalLoad()

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

Destructor for the [ElectricalLoad](#) class.

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

4.6.3 Member Function Documentation

4.6.3.1 clear()

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

Method to clear all attributes of the [ElectricalLoad](#) object.

```
157 {
158     this->n_points = 0;
159     this->n_years = 0;
160     this->min_load_kW = 0;
161     this->mean_load_kW = 0;
162     this->max_load_kW = 0;
163
164     this->path_2_electrical_load_time_series.clear();
165     this->time_vec_hrs.clear();
166     this->dt_vec_hrs.clear();
167     this->load_vec_kW.clear();
168
169     return;
170 } /* clear() */
```

4.6.3.2 readLoadData()

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

Method to read electrical load data into an already existing [ElectricalLoad](#) object. Clears and overwrites any existing attribute values.

Parameters

<i>path_2_electrical_load_time_series</i>	A string defining the path (either relative or absolute) to the given electrical load time series.
---	--

```
79 {
80     // 1. clear
81     this->clear();
82
83     // 2. init CSV reader, record path
84     io::CSVReader<2> CSV(path_2_electrical_load_time_series);
85
86     CSV.read_header(
87         io::ignore_extra_column,
88         "Time (since start of data) [hrs]",
89         "Electrical Load [kW]"
90     );
91
92     this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;
93
94     // 3. read in time and load data, increment n_points, track min and max load
95     double time_hrs = 0;
96     double load_kW = 0;
97     double load_sum_kW = 0;
98
99     this->n_points = 0;
100
101     this->min_load_kW = std::numeric_limits<double>::infinity();
102     this->max_load_kW = -1 * std::numeric_limits<double>::infinity();
103
104     while (CSV.read_row(time_hrs, load_kW)) {
105         this->time_vec_hrs.push_back(time_hrs);
106         this->load_vec_kW.push_back(load_kW);
107
108         load_sum_kW += load_kW;
109
110         this->n_points++;
111
112         if (this->min_load_kW > load_kW) {
113             this->min_load_kW = load_kW;
114         }
115     }
```

```

116         if (this->max_load_kW < load_kW) {
117             this->max_load_kW = load_kW;
118         }
119     }
120
121     // 4. compute mean load
122     this->mean_load_kW = load_sum_kW / this->n_points;
123
124     // 5. set number of years (assuming 8,760 hours per year)
125     this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;
126
127     // 6. populate dt_vec_hrs
128     this->dt_vec_hrs.resize(n_points, 0);
129
130     for (int i = 0; i < n_points; i++) {
131         if (i == n_points - 1) {
132             this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
133         }
134         else {
135             double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];
136             this->dt_vec_hrs[i] = dt_hrs;
137         }
138     }
139
140 }
141
142 return;
143 } /* readLoadData() */

```

4.6.4 Member Data Documentation

4.6.4.1 dt_vec_hrs

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

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

4.6.4.2 load_vec_kW

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

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

4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

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

4.6.4.4 mean_load_kW

```
double ElectricalLoad::mean_load_kW
```

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

4.6.4.5 min_load_kW

```
double ElectricalLoad::min_load_kW
```

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

4.6.4.6 n_points

```
int ElectricalLoad::n_points
```

The number of points in the modelling time series.

4.6.4.7 n_years

```
double ElectricalLoad::n_years
```

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

4.6.4.8 path_2_electrical_load_time_series

```
std::string ElectricalLoad::path_2_electrical_load_time_series
```

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

4.6.4.9 time_vec_hrs

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

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

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

- header/[ElectricalLoad.h](#)
- source/[ElectricalLoad.cpp](#)

4.7 Emissions Struct Reference

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

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

Public Attributes

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

4.7.1 Detailed Description

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

4.7.2 Member Data Documentation

4.7.2.1 CH4_kg

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

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

4.7.2.2 CO2_kg

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

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

4.7.2.3 CO_kg

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

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

4.7.2.4 NOx_kg

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

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

4.7.2.5 PM_kg

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

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

4.7.2.6 SOx_kg

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

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

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

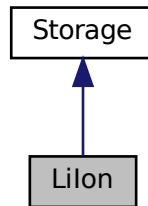
- [header/Production/Combustion/Combustion.h](#)

4.8 Lilon Class Reference

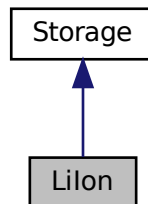
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

```
#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 [LiIon](#) class.

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

4.8.2.2 ~LiIon()

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

Destructor for the [LiIon](#) class.

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

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

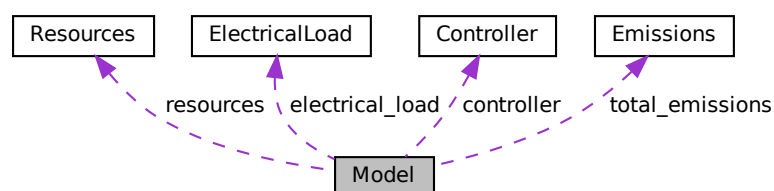
- header/Storage/[LiIon.h](#)
- source/Storage/[LiIon.cpp](#)

4.9 Model Class Reference

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

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

Collaboration diagram for Model:



Public Member Functions

- [Model](#) (void)
Constructor (dummy) for the [Model](#) class.
- [Model](#) ([ModelInputs](#))
Constructor (intended) for the [Model](#) class.
- void [addDiesel](#) ([DieselInputs](#))
Method to add a [Diesel](#) asset to the [Model](#).
- void [addResource](#) ([RenewableType](#), std::string, int)
A method to add a renewable resource time series to the [Model](#).
- void [addSolar](#) ([SolarInputs](#))
Method to add a [Solar](#) asset to the [Model](#).
- void [addTidal](#) ([TidalInputs](#))
Method to add a [Tidal](#) asset to the [Model](#).
- void [addWave](#) ([WaveInputs](#))
Method to add a [Wave](#) asset to the [Model](#).
- void [addWind](#) ([WindInputs](#))
Method to add a [Wind](#) asset to the [Model](#).
- void [run](#) (void)
A method to run the [Model](#).
- void [reset](#) (void)
Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors and resets select [Model](#) attributes. It leaves the [Controller](#), [ElectricalLoad](#), and [Resources](#) objects of the [Model](#) alone.
- void [clear](#) (void)
Method to clear all attributes of the [Model](#) object.
- void [writeResults](#) (std::string, int=-1)
Method which writes [Model](#) results to an output directory. Also calls out to [writeResults\(\)](#) for each contained asset.
- [~Model](#) (void)
Destructor for the [Model](#) class.

Public Attributes

- double [total_fuel_consumed_L](#)
The total fuel consumed [L] over a model run.
- [Emissions](#) [total_emissions](#)
An [Emissions](#) structure for holding total emissions [kg].
- double [net_present_cost](#)
The net present cost of the [Model](#) (undefined currency).
- double [total_dispatch_discharge_kWh](#)
The total energy dispatched/discharged [kWh] over the [Model](#) run.
- double [levellized_cost_of_energy_kWh](#)
The levellized cost of energy, per unit energy dispatched/discharged, of the [Model](#) [1/kWh] (undefined currency).
- [Controller](#) [controller](#)
[Controller](#) component of [Model](#).
- [ElectricalLoad](#) [electrical_load](#)
[ElectricalLoad](#) component of [Model](#).
- [Resources](#) [resources](#)
[Resources](#) component of [Model](#).
- std::vector< [Combustion](#) * > [combustion_ptr_vec](#)
A vector of pointers to the various [Combustion](#) assets in the [Model](#).
- std::vector< [Renewable](#) * > [renewable_ptr_vec](#)
A vector of pointers to the various [Renewable](#) assets in the [Model](#).
- std::vector< [Storage](#) * > [storage_ptr_vec](#)
A vector of pointers to the various [Storage](#) assets in the [Model](#).

Private Member Functions

- void `__checkInputs` (`ModelInputs`)
Helper method (private) to check inputs to the `Model` constructor.
- void `__computeFuelAndEmissions` (void)
Helper method to compute the total fuel consumption and emissions over the `Model` run.
- void `__computeNetPresentCost` (void)
Helper method to compute the overall net present cost, for the `Model` run, from the asset-wise net present costs.
- void `__computeLevellizedCostOfEnergy` (void)
Helper method to compute the overall levellized cost of energy, for the `Model` run, from the asset-wise levellized costs of energy.
- void `__computeEconomics` (void)
Helper method to compute key economic metrics for the `Model` run.
- void `__writeSummary` (std::string)
Helper method to write summary results for `Model`.
- void `__writeTimeSeries` (std::string, int=-1)
Helper method to write time series results for `Model`.

4.9.1 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.

```
468 {
469     return;
470 } /* Model() */
```

4.9.2.2 `Model()` [2/2]

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

Constructor (intended) for the `Model` class.

Parameters

<code>model_inputs</code>	A structure of <code>Model</code> constructor inputs.
---------------------------	---

```

487 {
488     // 1. check inputs
489     this->__checkInputs(model_inputs);
490
491     // 2. read in electrical load data
492     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
493
494     // 3. set control mode
495     this->controller.setControlMode(model_inputs.control_mode);
496
497     // 4. set public attributes
498     this->total_fuel_consumed_L = 0;
499     this->net_present_cost = 0;
500     this->total_dispatch_discharge_kWh = 0;
501     this->levellized_cost_of_energy_kWh = 0;
502
503     return;
504 } /* Model() */

```

4.9.2.3 ~Model()

```

Model::~~Model (
    void )

```

Destructor for the [Model](#) class.

```

906 {
907     this->clear();
908     return;
909 } /* ~Model() */

```

4.9.3 Member Function Documentation

4.9.3.1 __checkInputs()

```

void Model::__checkInputs (
    ModelInputs ) [private]

```

Helper method (private) to check inputs to the [Model](#) constructor.

Parameters

<i>model_inputs</i>	A structure of Model constructor inputs.
---------------------	--

```

40 {
41     //...
42
43     return;
44 } /* __checkInputs() */

```

4.9.3.2 __computeEconomics()

```

void Model::__computeEconomics (
    void ) [private]

```


Helper method to compute key economic metrics for the [Model](#) run.

```
208 {
209     this->__computeNetPresentCost();
210     this->__computeLevellizedCostOfEnergy();
211
212     return;
213 } /* __computeEconomics() */
```

4.9.3.3 __computeFuelAndEmissions()

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

Helper method to compute the total fuel consumption and emissions over the [Model](#) run.

```
60 {
61     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
62         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
63
64         this->total_fuel_consumed_L +=
65             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
66
67         this->total_emissions.CO2_kg +=
68             this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
69
70         this->total_emissions.CO_kg +=
71             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
72
73         this->total_emissions.NOx_kg +=
74             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
75
76         this->total_emissions.SOx_kg +=
77             this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
78
79         this->total_emissions.CH4_kg +=
80             this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
81
82         this->total_emissions.PM_kg +=
83             this->combustion_ptr_vec[i]->total_emissions.PM_kg;
84     }
85
86     return;
87 } /* __computeFuelAndEmissions() */
```

4.9.3.4 __computeLevellizedCostOfEnergy()

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

Helper method to compute the overall levellized cost of energy, for the [Model](#) run, from the asset-wise levellized costs of energy.

```
162 {
163     // 1. account for Combustion economics in levellized cost of energy
164     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
165         this->levellized_cost_of_energy_kWh +=
166             (
167                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
168                 this->combustion_ptr_vec[i]->total_dispatch_kWh
169             ) / this->total_dispatch_discharge_kWh;
170     }
171
172     // 2. account for Renewable economics in levellized cost of energy
173     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
174         this->levellized_cost_of_energy_kWh +=
175             (
176                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
177                 this->renewable_ptr_vec[i]->total_dispatch_kWh
178             ) / this->total_dispatch_discharge_kWh;
179     }
180 }
```

```

181 // 3. account for Storage economics in levellized cost of energy
182 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
183     /*
184     this->levellized_cost_of_energy_kWh +=
185     (
186         this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
187         this->storage_ptr_vec[i]->total_discharge_kWh
188     ) / this->total_dispatch_discharge_kWh;
189     */
190 }
191
192 return;
193 } /* __computeLevellizedCostOfEnergy() */

```

4.9.3.5 __computeNetPresentCost()

```

void Model::__computeNetPresentCost (
    void ) [private]

```

Helper method to compute the overall net present cost, for the [Model](#) run, from the asset-wise net present costs.

```

103 {
104     // 1. account for Combustion economics in net present cost
105     // increment total dispatch
106     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
107         this->combustion_ptr_vec[i]->computeEconomics(
108             &(this->electrical_load.time_vec_hrs)
109         );
110
111         this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
112
113         this->total_dispatch_discharge_kWh +=
114             this->combustion_ptr_vec[i]->total_dispatch_kWh;
115     }
116
117     // 2. account for Renewable economics in net present cost,
118     // increment total dispatch
119     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
120         this->renewable_ptr_vec[i]->computeEconomics(
121             &(this->electrical_load.time_vec_hrs)
122         );
123
124         this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
125
126         this->total_dispatch_discharge_kWh +=
127             this->renewable_ptr_vec[i]->total_dispatch_kWh;
128     }
129
130     // 3. account for Storage economics in net present cost
131     // increment total dispatch
132     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
133         /*
134         this->storage_ptr_vec[i]->computeEconomics(
135             &(this->electrical_load.time_vec_hrs)
136         );
137
138         this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
139
140         this->total_dispatch_discharge_kWh +=
141             this->storage_ptr_vec[i]->total_discharge_kWh;
142         */
143     }
144
145     return;
146 } /* __computeNetPresentCost() */

```

4.9.3.6 __writeSummary()

```

void Model::__writeSummary (
    std::string write_path ) [private]

```

Helper method to write summary results for [Model](#).

Parameters

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

```

231 {
232     // 1. create subdirectory
233     write_path += "Model/";
234     std::filesystem::create_directory(write_path);
235
236     // 2. create filestream
237     write_path += "summary_results.md";
238     std::ofstream ofs;
239     ofs.open(write_path, std::ofstream::out);
240
241     // 3. write summary results (markdown)
242     ofs << "# Model Summary Results\n";
243     ofs << "\n-----\n\n";
244
245     // 3.1. ElectricalLoad
246     ofs << "## Electrical Load\n";
247     ofs << "\n";
248     ofs << "Path: " <<
249         this->electrical_load.path_2_electrical_load_time_series << " \n";
250     ofs << "Data Points: " << this->electrical_load.n_points << " \n";
251     ofs << "Years: " << this->electrical_load.n_years << " \n";
252     ofs << "Min: " << this->electrical_load.min_load_kW << " kW \n";
253     ofs << "Mean: " << this->electrical_load.mean_load_kW << " kW \n";
254     ofs << "Max: " << this->electrical_load.max_load_kW << " kW \n";
255     ofs << "\n-----\n\n";
256
257     // 3.2. Controller
258     ofs << "## Controller\n";
259     ofs << "\n";
260     ofs << "Control Mode: " << this->controller.control_string << " \n";
261     ofs << "\n-----\n\n";
262
263     // 3.3. Resources (1D)
264     ofs << "## 1D Renewable Resources\n";
265     ofs << "\n";
266
267     std::map<int, std::string>::iterator string_map_1D_iter =
268         this->resources.string_map_1D.begin();
269     std::map<int, std::string>::iterator path_map_1D_iter =
270         this->resources.path_map_1D.begin();
271
272     while (
273         string_map_1D_iter != this->resources.string_map_1D.end() and
274         path_map_1D_iter != this->resources.path_map_1D.end()
275     ) {
276         ofs << "Resource Key: " << string_map_1D_iter->first << " \n";
277         ofs << "Type: " << string_map_1D_iter->second << " \n";
278         ofs << "Path: " << path_map_1D_iter->second << " \n";
279         ofs << "\n";
280
281         string_map_1D_iter++;
282         path_map_1D_iter++;
283     }
284
285     ofs << "\n-----\n\n";
286
287     // 3.4. Resources (2D)
288     ofs << "## 2D Renewable Resources\n";
289     ofs << "\n";
290
291     std::map<int, std::string>::iterator string_map_2D_iter =
292         this->resources.string_map_2D.begin();
293     std::map<int, std::string>::iterator path_map_2D_iter =
294         this->resources.path_map_2D.begin();
295
296     while (
297         string_map_2D_iter != this->resources.string_map_2D.end() and
298         path_map_2D_iter != this->resources.path_map_2D.end()
299     ) {
300         ofs << "Resource Key: " << string_map_2D_iter->first << " \n";
301         ofs << "Type: " << string_map_2D_iter->second << " \n";
302         ofs << "Path: " << path_map_2D_iter->second << " \n";
303         ofs << "\n";
304
305         string_map_2D_iter++;
306         path_map_2D_iter++;
307     }
308
309     ofs << "\n-----\n\n";
310

```

```

311 // 3.5. Combustion
312 ofs << "## Combustion Assets\n";
313 ofs << "\n";
314
315 for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
316     ofs << "Asset Index: " << i << " \n";
317     ofs << "Type: " << this->combustion_ptr_vec[i]->type_str << " \n";
318     ofs << "Capacity: " << this->combustion_ptr_vec[i]->capacity_kW << " kW \n";
319     ofs << "\n";
320 }
321
322 ofs << "\n-----\n\n";
323
324 // 3.6. Renewable
325 ofs << "## Renewable Assets\n";
326 ofs << "\n";
327
328 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
329     ofs << "Asset Index: " << i << " \n";
330     ofs << "Type: " << this->renewable_ptr_vec[i]->type_str << " \n";
331     ofs << "Capacity: " << this->renewable_ptr_vec[i]->capacity_kW << " kW \n";
332     ofs << "\n";
333 }
334
335 ofs << "\n-----\n\n";
336
337 // 3.7. Storage
338 ofs << "## Storage Assets\n";
339 ofs << "\n";
340
341 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
342     //...
343 }
344
345 ofs << "\n-----\n\n";
346
347 // 3.8. Model Results
348 ofs << "## Results\n";
349 ofs << "\n";
350
351 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
352 ofs << "\n";
353
354 ofs << "Total Dispatch + Discharge: " << this->total_dispatch_discharge_kWh
355     << " kWh \n";
356
357 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
358     << " per kWh dispatched/discharged \n";
359 ofs << "\n";
360
361 ofs << "Total Fuel Consumed: " << this->total_fuel_consumed_L << " L "
362     << "(Annual Average: " <<
363         this->total_fuel_consumed_L / this->electrical_load.n_years
364         << " L/yr) \n";
365 ofs << "\n";
366
367 ofs << "Total Carbon Dioxide (CO2) Emissions: " <<
368     this->total_emissions.CO2_kg << " kg "
369     << "(Annual Average: " <<
370         this->total_emissions.CO2_kg / this->electrical_load.n_years
371         << " kg/yr) \n";
372
373 ofs << "Total Carbon Monoxide (CO) Emissions: " <<
374     this->total_emissions.CO_kg << " kg "
375     << "(Annual Average: " <<
376         this->total_emissions.CO_kg / this->electrical_load.n_years
377         << " kg/yr) \n";
378
379 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
380     this->total_emissions.NOx_kg << " kg "
381     << "(Annual Average: " <<
382         this->total_emissions.NOx_kg / this->electrical_load.n_years
383         << " kg/yr) \n";
384
385 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
386     this->total_emissions.SOx_kg << " kg "
387     << "(Annual Average: " <<
388         this->total_emissions.SOx_kg / this->electrical_load.n_years
389         << " kg/yr) \n";
390
391 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
392     << "(Annual Average: " <<
393         this->total_emissions.CH4_kg / this->electrical_load.n_years
394         << " kg/yr) \n";
395
396 ofs << "Total Particulate Matter (PM) Emissions: " <<
397     this->total_emissions.PM_kg << " kg "

```

```

398         « "(Annual Average: " «
399         this->total_emissions.PM_kg / this->electrical_load.n_years
400         « " kg/yr)  \n";
401
402     ofs « "\n-----\n\n";
403
404     ofs.close();
405     return;
406 } /* __writeSummary() */

```

4.9.3.7 __writeTimeSeries()

```

void Model::__writeTimeSeries (
    std::string write_path,
    int max_lines = -1 ) [private]

```

Helper method to write time series results for [Model](#).

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>max_lines</i>	The maximum number of lines of output to write.

```

426 {
427     // 1. create filestream
428     write_path += "Model/time_series_results.csv";
429     std::ofstream ofs;
430     ofs.open(write_path, std::ofstream::out);
431
432     // 2. write time series results (comma separated value)
433     ofs « "Time (since start of data) [hrs],";
434     ofs « "Electrical Load [kW],";
435     ofs « "Net Load [kW],";
436     ofs « "Missed Load [kW],";
437     ofs « "\n";
438
439     for (int i = 0; i < max_lines; i++) {
440         ofs « this->electrical_load.time_vec_hrs[i] « ",";
441         ofs « this->electrical_load.load_vec_kW[i] « ",";
442         ofs « this->controller.net_load_vec_kW[i] « ",";
443         ofs « this->controller.missed_load_vec_kW[i] « ",";
444         ofs « "\n";
445     }
446
447     ofs.close();
448     return;
449 } /* __writeTimeSeries() */

```

4.9.3.8 addDiesel()

```

void Model::addDiesel (
    DieselInputs diesel_inputs )

```

Method to add a [Diesel](#) asset to the [Model](#).

Parameters

<i>diesel_inputs</i>	A structure of Diesel constructor inputs.
----------------------	---

```

521 {
522     Combustion* diesel_ptr = new Diesel(
523         this->electrical_load.n_points,
524         this->electrical_load.n_years,
525         diesel_inputs
526     );
527
528     this->combustion_ptr_vec.push_back(diesel_ptr);
529
530     return;
531 } /* addDiesel() */

```

4.9.3.9 addResource()

```

void Model::addResource (
    RenewableType renewable_type,
    std::string path_2_resource_data,
    int resource_key )

```

A method to add a renewable resource time series to the [Model](#).

Parameters

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

```

560 {
561     resources.addResource(
562         renewable_type,
563         path_2_resource_data,
564         resource_key,
565         &(this->electrical_load)
566     );
567
568     return;
569 } /* addResource() */

```

4.9.3.10 addSolar()

```

void Model::addSolar (
    SolarInputs solar_inputs )

```

Method to add a [Solar](#) asset to the [Model](#).

Parameters

<i>solar_inputs</i>	A structure of Solar constructor inputs.
---------------------	--

```

586 {
587     Renewable* solar_ptr = new Solar(
588         this->electrical_load.n_points,
589         this->electrical_load.n_years,
590         solar_inputs
591     );
592
593     this->renewable_ptr_vec.push_back(solar_ptr);

```

```

594
595     return;
596 } /* addSolar() */

```

4.9.3.11 addTidal()

```

void Model::addTidal (
    TidalInputs tidal_inputs )

```

Method to add a [Tidal](#) asset to the [Model](#).

Parameters

<i>tidal_inputs</i>	A structure of Tidal constructor inputs.
---------------------	--

```

613 {
614     Renewable* tidal_ptr = new Tidal(
615         this->electrical_load.n_points,
616         this->electrical_load.n_years,
617         tidal_inputs
618     );
619
620     this->renewable_ptr_vec.push_back(tidal_ptr);
621
622     return;
623 } /* addTidal() */

```

4.9.3.12 addWave()

```

void Model::addWave (
    WaveInputs wave_inputs )

```

Method to add a [Wave](#) asset to the [Model](#).

Parameters

<i>wave_inputs</i>	A structure of Wave constructor inputs.
--------------------	---

```

640 {
641     Renewable* wave_ptr = new Wave(
642         this->electrical_load.n_points,
643         this->electrical_load.n_years,
644         wave_inputs
645     );
646
647     this->renewable_ptr_vec.push_back(wave_ptr);
648
649     return;
650 } /* addWave() */

```

4.9.3.13 addWind()

```

void Model::addWind (
    WindInputs wind_inputs )

```

Method to add a [Wind](#) asset to the [Model](#).

Parameters

<code>wind_inputs</code>	A structure of Wind constructor inputs.
--------------------------	---

```

667 {
668     Renewable* wind_ptr = new Wind(
669         this->electrical_load.n_points,
670         this->electrical_load.n_years,
671         wind_inputs
672     );
673
674     this->renewable_ptr_vec.push_back(wind_ptr);
675
676     return;
677 } /* addWind() */

```

4.9.3.14 clear()

```

void Model::clear (
    void )

```

Method to clear all attributes of the [Model](#) object.

```

783 {
784     // 1. reset
785     this->reset();
786
787     // 2. clear components
788     controller.clear();
789     electrical_load.clear();
790     resources.clear();
791
792     return;
793 } /* clear() */

```

4.9.3.15 reset()

```

void Model::reset (
    void )

```

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

```

734 {
735     // 1. clear combustion_ptr_vec
736     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
737         delete this->combustion_ptr_vec[i];
738     }
739     this->combustion_ptr_vec.clear();
740
741     // 2. clear renewable_ptr_vec
742     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
743         delete this->renewable_ptr_vec[i];
744     }
745     this->renewable_ptr_vec.clear();
746
747     // 3. clear storage_ptr_vec
748     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
749         delete this->storage_ptr_vec[i];
750     }
751     this->storage_ptr_vec.clear();
752
753     // 4. reset attributes
754     this->total_fuel_consumed_L = 0;
755
756     this->total_emissions.CO2_kg = 0;
757     this->total_emissions.CO_kg = 0;

```



```

758     this->total_emissions.NOx_kg = 0;
759     this->total_emissions.SOx_kg = 0;
760     this->total_emissions.CH4_kg = 0;
761     this->total_emissions.PM_kg = 0;
762
763     this->net_present_cost = 0;
764     this->total_dispatch_discharge_kWh = 0;
765     this->levellized_cost_of_energy_kWh = 0;
766
767     return;
768 } /* reset() */

```

4.9.3.16 run()

```

void Model::run (
    void )

```

A method to run the [Model](#).

```

692 {
693     // 1. init Controller
694     this->controller.init (
695         &(this->electrical_load),
696         &(this->renewable_ptr_vec),
697         &(this->resources),
698         &(this->combustion_ptr_vec)
699     );
700
701     // 2. apply dispatch control
702     this->controller.applyDispatchControl (
703         &(this->electrical_load),
704         &(this->combustion_ptr_vec),
705         &(this->renewable_ptr_vec),
706         &(this->storage_ptr_vec)
707     );
708
709     // 3. compute total fuel consumption and emissions
710     this->__computeFuelAndEmissions();
711
712     // 4. compute key economic metrics
713     this->__computeEconomics();
714
715     return;
716 } /* run() */

```

4.9.3.17 writeResults()

```

void Model::writeResults (
    std::string write_path,
    int max_lines = -1 )

```

Method which writes [Model](#) results to an output directory. Also calls out to [writeResults\(\)](#) for each contained asset.

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>max_lines</i>	The maximum number of lines of output to write. If <0, then all available lines are written. If =0, then only summary results are written.

```

821 {
822     // 1. handle sentinel
823     if (max_lines < 0) {
824         max_lines = this->electrical_load.n_points;

```

```

825     }
826
827     // 2. check for pre-existing, warn (and remove), then create
828     if (write_path.back() != '/') {
829         write_path += '/';
830     }
831
832     if (std::filesystem::is_directory(write_path)) {
833         std::string warning_str = "WARNING: Model::writeResults(): ";
834         warning_str += write_path;
835         warning_str += " already exists, contents will be overwritten!";
836
837         std::cout << warning_str << std::endl;
838
839         std::filesystem::remove_all(write_path);
840     }
841
842     std::filesystem::create_directory(write_path);
843
844     // 3. write summary
845     this->__writeSummary(write_path);
846
847     // 4. write time series
848     if (max_lines > this->electrical_load.n_points) {
849         max_lines = this->electrical_load.n_points;
850     }
851
852     if (max_lines > 0) {
853         this->__writeTimeSeries(write_path, max_lines);
854     }
855
856     // 5. call out to Combustion :: writeResults()
857     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
858         this->combustion_ptr_vec[i]->writeResults(
859             write_path,
860             &(this->electrical_load.time_vec_hrs),
861             i,
862             max_lines
863         );
864     }
865
866     // 6. call out to Renewable :: writeResults()
867     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
868         this->renewable_ptr_vec[i]->writeResults(
869             write_path,
870             &(this->electrical_load.time_vec_hrs),
871             &(this->resources.resource_map_1D),
872             &(this->resources.resource_map_2D),
873             i,
874             max_lines
875         );
876     }
877
878     // 7. call out to Storage :: writeResults()
879     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
880         /*
881         this->storage_ptr_vec[i]->writeResults(
882             write_path,
883             &(this->electrical_load.time_vec_hrs),
884             i,
885             max_lines
886         );
887         */
888     }
889
890     return;
891 } /* writeResults() */

```

4.9.4 Member Data Documentation

4.9.4.1 combustion_ptr_vec

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

A vector of pointers to the various [Combustion](#) assets in the [Model](#).

4.9.4.2 controller

`Controller` `Model::controller`

`Controller` component of `Model`.

4.9.4.3 electrical_load

`ElectricalLoad` `Model::electrical_load`

`ElectricalLoad` component of `Model`.

4.9.4.4 levellized_cost_of_energy_kWh

`double` `Model::levellized_cost_of_energy_kWh`

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

4.9.4.5 net_present_cost

`double` `Model::net_present_cost`

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

4.9.4.6 renewable_ptr_vec

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

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

4.9.4.7 resources

`Resources` `Model::resources`

`Resources` component of `Model`.

4.9.4.8 storage_ptr_vec

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

A vector of pointers to the various [Storage](#) assets in the [Model](#).

4.9.4.9 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

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

4.9.4.10 total_emissions

```
Emissions Model::total_emissions
```

An [Emissions](#) structure for holding total emissions [kg].

4.9.4.11 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

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

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

- header/[Model.h](#)
- source/[Model.cpp](#)

4.10 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:

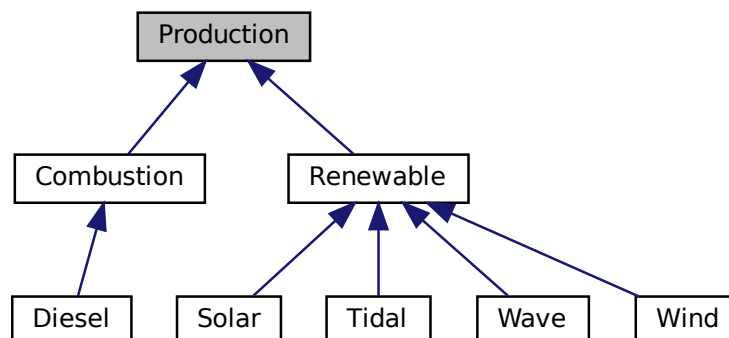
- [header/Model.h](#)

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, double, [ProductionInputs](#))
Constructor (intended) for the [Production](#) class.
- virtual void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual [~Production](#) (void)
Destructor for the [Production](#) class.

Public Attributes

- bool [print_flag](#)
A flag which indicates whether or not object construct/destruction should be verbose.
- bool [is_running](#)
A boolean which indicates whether or not the asset is running.
- bool [is_sunk](#)
A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- int [n_points](#)
The number of points in the modelling time series.
- int [n_starts](#)
The number of times the asset has been started.
- int [n_replacements](#)
The number of times the asset has been replaced.
- double [n_years](#)
The number of years being modelled.
- double [running_hours](#)
The number of hours for which the asset has been operating.
- double [replace_running_hrs](#)
The number of running hours after which the asset must be replaced.
- double [capacity_kW](#)
The rated production capacity [kW] of the asset.
- double [nominal_inflation_annual](#)
The nominal, annual inflation rate to use in computing model economics.
- double [nominal_discount_annual](#)
The nominal, annual discount rate to use in computing model economics.
- double [real_discount_annual](#)
The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double [capital_cost](#)
The capital cost of the asset (undefined currency).
- double [operation_maintenance_cost_kWh](#)
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.
- double [net_present_cost](#)
The net present cost of this asset.

- double [total_dispatch_kWh](#)
The total energy dispatched [kWh] over the [Model](#) run.
- double [levellized_cost_of_energy_kWh](#)
The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only 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).

Private Member Functions

- void [__checkInputs](#) (int, [ProductionInputs](#))
Helper method to check inputs to the [Production](#) constructor.
- void [__handleReplacement](#) (int)
Helper method to handle asset replacement and capital cost incursion, if applicable.
- double [__computeRealDiscountAnnual](#) (double, double)
Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

4.11.1 Detailed Description

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

4.11.2 Constructor & Destructor Documentation

4.11.2.1 Production() [1/2]

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

Constructor (dummy) for the [Production](#) class.

```
165 {
166     return;
167 } /* Production() */
```

4.11.2.2 Production() [2/2]

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

Constructor (intended) for the [Production](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>production_inputs</i>	A structure of Production constructor inputs.

```
196 {
197     // 1. check inputs
198     this->__checkInputs(n_points, production_inputs);
199
200     // 2. set attributes
201     this->print_flag = production_inputs.print_flag;
202     this->is_running = false;
203     this->is_sunk = production_inputs.is_sunk;
204
205     this->n_points = n_points;
206     this->n_starts = 0;
207     this->n_replacements = 0;
208
209     this->n_years = n_years;
210
211     this->running_hours = 0;
212     this->replace_running_hrs = production_inputs.replace_running_hrs;
213
214     this->capacity_kW = production_inputs.capacity_kW;
215
216     this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
217     this->nominal_discount_annual = production_inputs.nominal_discount_annual;
218     this->real_discount_annual = this->__computeRealDiscountAnnual(
219         production_inputs.nominal_inflation_annual,
220         production_inputs.nominal_discount_annual
221     );
222     this->capital_cost = 0;
223     this->operation_maintenance_cost_kWh = 0;
224     this->net_present_cost = 0;
225     this->total_dispatch_kWh = 0;
226     this->levelized_cost_of_energy_kWh = 0;
227
228     this->is_running_vec.resize(this->n_points, 0);
229
230     this->production_vec_kW.resize(this->n_points, 0);
231     this->dispatch_vec_kW.resize(this->n_points, 0);
232     this->storage_vec_kW.resize(this->n_points, 0);
233     this->curtailment_vec_kW.resize(this->n_points, 0);
234
235     this->capital_cost_vec.resize(this->n_points, 0);
236     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
237
238     // 3. construction print
239     if (this->print_flag) {
```



```

240         std::cout << "Production object constructed at " << this << std::endl;
241     }
242
243     return;
244 } /* Production() */

```

4.11.2.3 ~Production()

```

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

```

Destructor for the [Production](#) class.

```

394 {
395     // 1. destruction print
396     if (this->print_flag) {
397         std::cout << "Production object at " << this << " destroyed" << std::endl;
398     }
399
400     return;
401 } /* ~Production() */

```

4.11.3 Member Function Documentation

4.11.3.1 __checkInputs()

```

void Production::__checkInputs (
    int n_points,
    ProductionInputs production_inputs ) [private]

```

Helper method to check inputs to the [Production](#) constructor.

Parameters

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

```

41 {
42     // 1. check n_points
43     if (n_points <= 0) {
44         std::string error_str = "ERROR: Production(): n_points must be > 0";
45
46         #ifdef _WIN32
47             std::cout << error_str << std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     // 2. check capacity_kW
54     if (production_inputs.capacity_kW <= 0) {
55         std::string error_str = "ERROR: Production(): ";
56         error_str += "ProductionInputs::capacity_kW must be > 0";
57
58         #ifdef _WIN32
59             std::cout << error_str << std::endl;
60         #endif
61
62         throw std::invalid_argument(error_str);
63     }
64 }

```

```

65 // 3. check replace_running_hrs
66 if (production_inputs.replace_running_hrs <= 0) {
67     std::string error_str = "ERROR: Production(): ";
68     error_str += "ProductionInputs::replace_running_hrs must be > 0";
69
70     #ifdef _WIN32
71         std::cout << error_str << std::endl;
72     #endif
73
74     throw std::invalid_argument(error_str);
75 }
76
77 return;
78 } /* __checkInputs() */

```

4.11.3.2 __computeRealDiscountAnnual()

```

double Production::__computeRealDiscountAnnual (
    double nominal_inflation_annual,
    double nominal_discount_annual ) [private]

```

Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.

Ref: [HOMER \[2023h\]](#)

Ref: [HOMER \[2023b\]](#)

Parameters

<i>nominal_inflation_annual</i>	The nominal, annual inflation rate to use in computing model economics.
<i>nominal_discount_annual</i>	The nominal, annual discount rate to use in computing model economics.

Returns

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

```

110 {
111     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
112     real_discount_annual /= 1 + nominal_inflation_annual;
113
114     return real_discount_annual;
115 } /* __computeRealDiscountAnnual() */

```

4.11.3.3 __handleReplacement()

```

void Production::__handleReplacement (
    int timestep ) [private]

```

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

Parameters

<i>timestep</i>	The current time step of the Model run.
-----------------	---

```

133 {
134     if (
135         this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs
136     ) {
137         // 1. log replacement
138         this->n_replacements++;
139
140         // 2. incur capital cost in timestep
141         this->capital_cost_vec[timestep] = this->capital_cost;
142     }
143
144     return;
145 } /* __handleReplacement() */

```

4.11.3.4 commit()

```

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

```

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

Parameters

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

```

337 {
338     // 1. record production
339     this->production_vec_kW[timestep] = production_kW;
340
341     // 2. compute and record dispatch and curtailment
342     double dispatch_kW = 0;
343     double curtailment_kW = 0;
344
345     if (production_kW > load_kW) {
346         dispatch_kW = load_kW;
347         curtailment_kW = production_kW - dispatch_kW;
348     }
349
350     else {
351         dispatch_kW = production_kW;
352     }
353
354     this->dispatch_vec_kW[timestep] = dispatch_kW;
355     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
356     this->curtailment_vec_kW[timestep] = curtailment_kW;
357
358     // 3. update load
359     load_kW -= dispatch_kW;
360
361     if (this->is_running) {
362         // 4. log running state, running hours
363         this->is_running_vec[timestep] = this->is_running;
364         this->running_hours += dt_hrs;

```

```

365
366     // 5. incur operation and maintenance costs
367     double produced_kWh = production_kW * dt_hrs;
368
369     double operation_maintenance_cost =
370         this->operation_maintenance_cost_kWh * produced_kWh;
371     this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
372
373     // 6. incur capital costs (i.e., handle replacement)
374     this->__handleReplacement(timestep);
375 }
376
377
378     return load_kW;
379 } /* commit() */

```

4.11.3.5 computeEconomics()

```

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

```

Helper method to compute key economic metrics for the [Model](#) run.

Ref: [HOMER \[2023b\]](#)

Ref: [HOMER \[2023g\]](#)

Ref: [HOMER \[2023i\]](#)

Ref: [HOMER \[2023a\]](#)

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the <i>time_vec_hrs</i> attribute of the ElectricalLoad .
-------------------------	--

1. compute levlized cost of energy (per unit dispatched)

Reimplemented in [Renewable](#), and [Combustion](#).

```

266 {
267     // 1. compute net present cost
268     double t_hrs = 0;
269     double real_discount_scalar = 0;
270
271     for (int i = 0; i < this->n_points; i++) {
272         t_hrs = time_vec_hrs_ptr->at(i);
273
274         real_discount_scalar = 1.0 / pow(
275             1 + this->real_discount_annual,
276             t_hrs / 8760
277         );
278
279         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
280
281         this->net_present_cost +=
282             real_discount_scalar * this->operation_maintenance_cost_vec[i];
283     }
284
285     // assuming 8,760 hours per year
286     double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
287
288     double capital_recovery_factor =
289         (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
290         (pow(1 + this->real_discount_annual, n_years) - 1);
291
292     double total_annualized_cost = capital_recovery_factor *
293         this->net_present_cost;
294
295     this->levellized_cost_of_energy_kWh =

```

```
297         (n_years * total_annualized_cost) /  
298         total_dispatch_kWh;  
299  
300     return;  
301 } /* computeEconomics() */
```

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 levlized_cost_of_energy_kWh

```
double Production::levellized_cost_of_energy_kWh
```

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

4.11.4.10 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.11.4.11 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.11.4.12 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.11.4.13 n_years

```
double Production::n_years
```

The number of years being modelled.

4.11.4.14 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.11.4.15 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

4.11.4.16 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

4.11.4.17 operation_maintenance_cost_kWh

```
double Production::operation_maintenance_cost_kWh
```

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

4.11.4.18 operation_maintenance_cost_vec

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

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

4.11.4.19 print_flag

```
bool Production::print_flag
```

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

4.11.4.20 production_vec_kW

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

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

4.11.4.21 real_discount_annual

```
double Production::real_discount_annual
```

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

4.11.4.22 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.11.4.23 running_hours

```
double Production::running_hours
```

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

4.11.4.24 storage_vec_kW

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

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

4.11.4.25 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

4.11.4.26 type_str

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

A string describing the type of the asset.

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

- header/Production/[Production.h](#)
- source/Production/[Production.cpp](#)

4.12 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:

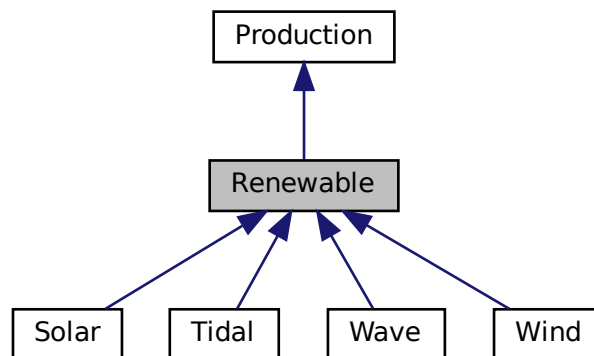
- header/Production/[Production.h](#)

4.13 Renewable Class Reference

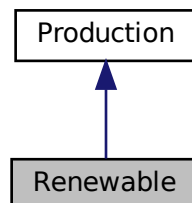
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#include <Renewable.h>
```

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



Public Member Functions

- [Renewable](#) (void)
Constructor (dummy) for the [Renewable](#) class.
- [Renewable](#) (int, double, [RenewableInputs](#))
Constructor (intended) for the [Renewable](#) class.
- void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [computeProductionkW](#) (int, double, double)
- virtual double [computeProductionkW](#) (int, double, double, double)
- virtual double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- void [writeResults](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int, int=-1)
Method which writes [Renewable](#) results to an output directory.
- virtual [~Renewable](#) (void)
Destructor for the [Renewable](#) class.

Public Attributes

- [RenewableType](#) type
The type ([RenewableType](#)) of the asset.
- int [resource_key](#)
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.

Private Member Functions

- void [__checkInputs](#) ([RenewableInputs](#))
Helper method to check inputs to the [Renewable](#) constructor.
- void [__handleStartStop](#) (int, double, double)
Helper method to handle the starting/stopping of the renewable asset.
- virtual void [__writeSummary](#) (std::string)
- virtual void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

4.13.1 Detailed Description

The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 Renewable() [1/2]

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

Constructor (dummy) for the [Renewable](#) class.

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

4.13.2.2 Renewable() [2/2]

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

Constructor (intended) for the [Renewable](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>renewable_inputs</i>	A structure of Renewable constructor inputs.

```
124 :
125 Production(
126     n_points,
127     n_years,
128     renewable_inputs.production_inputs
129 )
130 {
131     // 1. check inputs
132     this->__checkInputs(renewable_inputs);
133
134     // 2. set attributes
135     //...
136
137     // 3. construction print
138     if (this->print_flag) {
139         std::cout << "Renewable object constructed at " << this << std::endl;
140     }
141
142     return;
143 } /* Renewable() */
```

4.13.2.3 ~Renewable()

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

Destructor for the [Renewable](#) class.

```
320 {
321     // 1. destruction print
322     if (this->print_flag) {
323         std::cout << "Renewable object at " << this << " destroyed" << std::endl;
324     }
325
326     return;
327 } /* ~Renewable() */
```

4.13.3 Member Function Documentation

4.13.3.1 `__checkInputs()`

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

Helper method to check inputs to the [Renewable](#) constructor.

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

4.13.3.2 `__handleStartStop()`

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

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

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

4.13.3.3 `__writeSummary()`

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

Reimplemented in [Wind](#), [Wave](#), [Tidal](#), and [Solar](#).

```
72 {return;}
```

4.13.3.4 `__writeTimeSeries()`

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

Reimplemented in [Wind](#), [Wave](#), [Tidal](#), and [Solar](#).

```
79 {return;}
```

4.13.3.5 `commit()`

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

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

Parameters

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

```
201 {
202     // 1. handle start/stop
203     this->__handleStartStop(timestep, dt_hrs, production_kW);
204
205     // 2. invoke base class method
206     load_kW = Production::commit(
207         timestep,
208         dt_hrs,
209         production_kW,
210         load_kW
211     );
212
213
214     //...
215
216     return load_kW;
217 } /* commit() */
```

4.13.3.6 computeEconomics()

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

Helper method to compute key economic metrics for the [Model](#) run.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the <i>time_vec_hrs</i> attribute of the ElectricalLoad .
-------------------------	--

Reimplemented from [Production](#).

```
160 {
161     // 1. invoke base class method
162     Production::computeEconomics(time_vec_hrs_ptr);
163
164     return;
165 } /* computeEconomics() */
```

4.13.3.7 computeProductionkW() [1/2]

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

Reimplemented in [Wind](#), [Tidal](#), and [Solar](#).

```
95 {return 0;}
```

4.13.3.8 computeProductionkW() [2/2]

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

Reimplemented in [Wave](#).

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

4.13.3.9 writeResults()

```
void Renewable::writeResults (
    std::string write_path,
    std::vector< double > * time_vec_hrs_ptr,
    std::map< int, std::vector< double >> * resource_map_1D_ptr,
    std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
    int renewable_index,
    int max_lines = -1 )
```

Method which writes [Renewable](#) results to an output directory.

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>time_vec_hrs_ptr</i>	A pointer to the <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>renewable_index</i>	An integer which corresponds to the index of the Renewable asset in the Model .
<i>max_lines</i>	The maximum number of lines of output to write. If <0 , then all available lines are written. If $=0$, then only summary results are written.

```

261 {
262     // 1. handle sentinel
263     if (max_lines < 0) {
264         max_lines = this->n_points;
265     }
266
267     // 2. create subdirectories
268     write_path += "Production/";
269     if (not std::filesystem::is_directory(write_path)) {
270         std::filesystem::create_directory(write_path);
271     }
272
273     write_path += "Renewable/";
274     if (not std::filesystem::is_directory(write_path)) {
275         std::filesystem::create_directory(write_path);
276     }
277
278     write_path += this->type_str;
279     write_path += "_";
280     write_path += std::to_string(int(ceil(this->capacity_kw)));
281     write_path += "kW_idx";
282     write_path += std::to_string(renewable_index);
283     write_path += "/";
284     std::filesystem::create_directory(write_path);
285
286     // 3. write summary
287     this->__writeSummary(write_path);
288
289     // 4. write time series
290     if (max_lines > this->n_points) {
291         max_lines = this->n_points;
292     }
293
294     if (max_lines > 0) {
295         this->__writeTimeSeries(
296             write_path,
297             time_vec_hrs_ptr,
298             resource_map_1D_ptr,
299             resource_map_2D_ptr,
300             max_lines
301         );
302     }
303
304     return;
305 } /* writeResults() */

```

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:

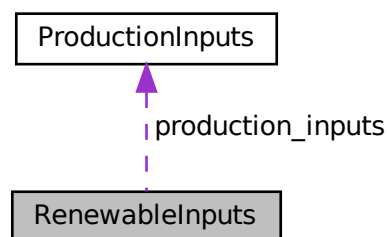
- `header/Production/Renewable/Renewable.h`
- `source/Production/Renewable/Renewable.cpp`

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 `addResource` (`RenewableType`, `std::string`, `int`, `ElectricalLoad *`)
A method to add a renewable resource time series to `Resources`. Checks if given resource key is already in use. The associated helper methods also check against `ElectricalLoad` to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).
- void `clear` (void)
Method to clear all attributes of the `Resources` object.
- `~Resources` (void)
Destructor for the `Resources` class.

Public Attributes

- `std::map< int, std::vector< double > >` `resource_map_1D`
A map `<int, vector>` of given 1D renewable resource time series.
- `std::map< int, std::string >` `string_map_1D`
A map `<int, string>` of descriptors for the type of the given 1D renewable resource time series.
- `std::map< int, std::string >` `path_map_1D`
A map `<int, string>` of the paths (either relative or absolute) to given 1D renewable resource time series.
- `std::map< int, std::vector< std::vector< double > > >` `resource_map_2D`
A map `<int, vector>` of given 2D renewable resource time series.
- `std::map< int, std::string >` `string_map_2D`
A map `<int, string>` of descriptors for the type of the given 2D renewable resource time series.
- `std::map< int, std::string >` `path_map_2D`
A map `<int, string>` of the paths (either relative or absolute) to given 2D renewable resource time series.

Private Member Functions

- void `__checkResourceKey1D` (int, [RenewableType](#))
Helper method to check if given resource key (1D) is already in use.
- void `__checkResourceKey2D` (int, [RenewableType](#))
Helper method to check if given resource key (2D) is already in use.
- void `__checkTimePoint` (double, double, std::string, [ElectricalLoad](#) *)
Helper method to check received time point against expected time point.
- void `__throwLengthError` (std::string, [ElectricalLoad](#) *)
Helper method to throw data length error.
- void `__readSolarResource` (std::string, int, [ElectricalLoad](#) *)
Helper method to handle reading a solar resource time series into [Resources](#).
- void `__readTidalResource` (std::string, int, [ElectricalLoad](#) *)
Helper method to handle reading a tidal resource time series into [Resources](#).
- void `__readWaveResource` (std::string, int, [ElectricalLoad](#) *)
Helper method to handle reading a wave resource time series into [Resources](#).
- void `__readWindResource` (std::string, int, [ElectricalLoad](#) *)
Helper method to handle reading a wind resource time series into [Resources](#).

4.15.1 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.

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

4.15.2.2 ~Resources()

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

Destructor for the [Resources](#) class.

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

4.15.3 Member Function Documentation

4.15.3.1 `__checkResourceKey1D()`

```
void Resources::__checkResourceKey1D (
    int resource_key,
    RenewableType renewable_type ) [private]
```

Helper method to check if given resource key (1D) is already in use.

Parameters

<i>resource_key</i>	The key associated with the given renewable resource.
---------------------	---

```
45 {
46     if (this->resource_map_1D.count(resource_key) > 0) {
47         std::string error_str = "ERROR: Resources::addResource(";
48
49         switch (renewable_type) {
50             case (RenewableType :: SOLAR): {
51                 error_str += "SOLAR): ";
52
53                 break;
54             }
55
56             case (RenewableType :: TIDAL): {
57                 error_str += "TIDAL): ";
58
59                 break;
60             }
61
62             case (RenewableType :: WIND): {
63                 error_str += "WIND): ";
64
65                 break;
66             }
67
68             default: {
69                 error_str += "UNDEFINED_TYPE): ";
70
71                 break;
72             }
73         }
74
75         error_str += "resource key (1D) ";
76         error_str += std::to_string(resource_key);
77         error_str += " is already in use";
78
79         #ifdef _WIN32
80             std::cout << error_str << std::endl;
81         #endif
82
83         throw std::invalid_argument(error_str);
84     }
85
86     return;
87 } /* __checkResourceKey1D() */
```

4.15.3.2 `__checkResourceKey2D()`

```
void Resources::__checkResourceKey2D (
    int resource_key,
    RenewableType renewable_type ) [private]
```

Helper method to check if given resource key (2D) is already in use.

Parameters

<i>resource_key</i>	The key associated with the given renewable resource.
---------------------	---

```

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

```

4.15.3.3 __checkTimePoint()

```

void Resources::__checkTimePoint (
    double time_received_hrs,
    double time_expected_hrs,
    std::string path_2_resource_data,
    ElectricalLoad * electrical_load_ptr ) [private]

```

Helper method to check received time point against expected time point.

Parameters

<i>time_received_hrs</i>	The point in time received from the given data.
<i>time_expected_hrs</i>	The point in time expected (this comes from the electrical load time series).
<i>path_2_resource_data</i>	The path (either relative or absolute) to the given resource time series.
<i>electrical_load_ptr</i>	A pointer to the Model's ElectricalLoad object.

```

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

```

```

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

```

4.15.3.4 __readSolarResource()

```

void Resources::__readSolarResource (
    std::string path_2_resource_data,
    int resource_key,
    ElectricalLoad * electrical_load_ptr ) [private]

```

Helper method to handle reading a solar resource time series into [Resources](#).

Parameters

<i>path_2_resource_data</i>	The path (either relative or absolute) to the given resource time series.
<i>resource_key</i>	The key associated with the given renewable resource.
<i>electrical_load_ptr</i>	A pointer to the Model's ElectricalLoad object.

```

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

```

4.15.3.5 __readTidalResource()

```
void Resources::__readTidalResource (
    std::string path_2_resource_data,
    int resource_key,
    ElectricalLoad * electrical_load_ptr ) [private]
```

Helper method to handle reading a tidal resource time series into [Resources](#).

Parameters

<i>path_2_resource_data</i>	The path (either relative or absolute) to the given resource time series.
<i>resource_key</i>	The key associated with the given renewable resource.
<i>electrical_load_ptr</i>	A pointer to the Model's ElectricalLoad object.

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


4.15.3.6 `__readWaveResource()`

```
void Resources::__readWaveResource (
    std::string path_2_resource_data,
    int resource_key,
    ElectricalLoad * electrical_load_ptr ) [private]
```

Helper method to handle reading a wave resource time series into [Resources](#).

Parameters

<code>path_2_resource_data</code>	The path (either relative or absolute) to the given resource time series.
<code>resource_key</code>	The key associated with the given renewable resource.
<code>electrical_load_ptr</code>	A pointer to the Model's ElectricalLoad object.

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

4.15.3.7 __readWindResource()

```
void Resources::__readWindResource (
    std::string path_2_resource_data,
    int resource_key,
    ElectricalLoad * electrical_load_ptr ) [private]
```

Helper method to handle reading a wind resource time series into [Resources](#).

Parameters

<i>path_2_resource_data</i>	The path (either relative or absolute) to the given resource time series.
<i>resource_key</i>	The key associated with the given renewable resource.
<i>electrical_load_ptr</i>	A pointer to the Model's ElectricalLoad object.

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

4.15.3.8 __throwLengthError()

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

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

Helper method to throw data length error.

Parameters

<i>path_2_resource_data</i>	The path (either relative or absolute) to the given resource time series.
<i>electrical_load_ptr</i>	A pointer to the Model's ElectricalLoad object.

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

4.15.3.9 addResource()

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

A method to add a renewable resource time series to [Resources](#). Checks if given resource key is already in use. The associated helper methods also check against [ElectricalLoad](#) to ensure that all added time series align with the electrical load time series (both in terms of length and which points in time are included).

Parameters

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

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

```

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

```

4.15.3.10 clear()

```

void Resources::clear (
    void )

```

Method to clear all attributes of the [Resources](#) object.

```

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

```

4.15.4 Member Data Documentation

4.15.4.1 path_map_1D

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

A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.

4.15.4.2 path_map_2D

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

A map <int, string> of the paths (either relative or absolute) to given 2D renewable resource time series.

4.15.4.3 resource_map_1D

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

A map <int, vector> of given 1D renewable resource time series.

4.15.4.4 resource_map_2D

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

A map <int, vector> of given 2D renewable resource time series.

4.15.4.5 string_map_1D

```
std::map<int, std::string> Resources::string_map_1D
```

A map <int, string> of descriptors for the type of the given 1D renewable resource time series.

4.15.4.6 string_map_2D

```
std::map<int, std::string> Resources::string_map_2D
```

A map <int, string> of descriptors for the type of the given 2D renewable resource time series.

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

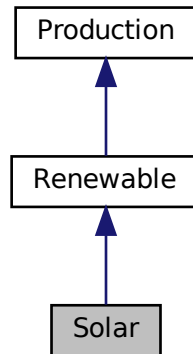
- header/[Resources.h](#)
- source/[Resources.cpp](#)

4.16 Solar Class Reference

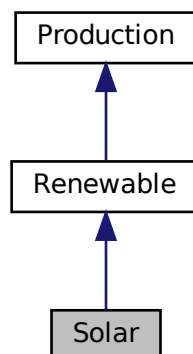
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

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

Inheritance diagram for Solar:



Collaboration diagram for Solar:



Public Member Functions

- [Solar](#) (void)
Constructor (dummy) for the [Solar](#) class.
- [Solar](#) (int, double, [SolarInputs](#))

Constructor (intended) for the [Solar](#) class.

- double [computeProductionkW](#) (int, double, double)

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

- double [commit](#) (int, double, double, double)

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

- [~Solar](#) (void)

Destructor for the [Solar](#) class.

Public Attributes

- double [derating](#)

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

Private Member Functions

- void [__checkInputs](#) ([SolarInputs](#))

Helper method to check inputs to the [Solar](#) constructor.

- double [__getGenericCapitalCost](#) (void)

Helper method to generate a generic solar PV array capital cost.

- double [__getGenericOpMaintCost](#) (void)

Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.

- void [__writeSummary](#) (std::string)

Helper method to write summary results for [Solar](#).

- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

Helper method to write time series results for [Solar](#).

4.16.1 Detailed Description

A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

4.16.2 Constructor & Destructor Documentation

4.16.2.1 [Solar\(\)](#) [1/2]

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

Constructor (dummy) for the [Solar](#) class.

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

4.16.2.2 Solar() [2/2]

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

Constructor (intended) for the [Solar](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>solar_inputs</i>	A structure of Solar constructor inputs.

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

4.16.2.3 ~Solar()

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

Destructor for the [Solar](#) class.

```
462 {
463     // 1. destruction print
464     if (this->print_flag) {
465         std::cout << "Solar object at " << this << " destroyed" << std::endl;
466     }
467
468     return;
469 } /* ~Solar() */
```


4.16.3 Member Function Documentation

4.16.3.1 `__checkInputs()`

```
void Solar::__checkInputs (
    SolarInputs solar_inputs ) [private]
```

Helper method to check inputs to the `Solar` constructor.

```
37 {
38     // 1. check derating
39     if (
40         solar_inputs.derating < 0 or
41         solar_inputs.derating > 1
42     ) {
43         std::string error_str = "ERROR: Solar(): ";
44         error_str += "SolarInputs::derating must be in the closed interval [0, 1]";
45
46         #ifdef _WIN32
47             std::cout << error_str << std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     return;
54 } /* __checkInputs() */
```

4.16.3.2 `__getGenericCapitalCost()`

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

Helper method to generate a generic solar PV array capital cost.

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

Returns

A generic capital cost for the solar PV array [CAD].

```
76 {
77     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;
78
79     return capital_cost_per_kW * this->capacity_kW;
80 } /* __getGenericCapitalCost() */
```

4.16.3.3 `__getGenericOpMaintCost()`

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

Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.

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

Returns

A generic operation and maintenance cost, per unit energy produced, for the solar PV array [CAD/kWh].

```
103 {
104     return 0.01;
105 } /* __getGenericOpMaintCost() */
```

4.16.3.4 __writeSummary()

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

Helper method to write summary results for [Solar](#).

Parameters

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

Reimplemented from [Renewable](#).

```
123 {
124     // 1. create filestream
125     write_path += "summary_results.md";
126     std::ofstream ofs;
127     ofs.open(write_path, std::ofstream::out);
128
129     // 2. write summary results (markdown)
130     ofs << "# ";
131     ofs << std::to_string(int(ceil(this->capacity_kW)));
132     ofs << " kW SOLAR Summary Results\n";
133     ofs << "\n-----\n\n";
134
135     // 2.1. Production attributes
136     ofs << "## Production Attributes\n";
137     ofs << "\n";
138
139     ofs << "Capacity: " << this->capacity_kW << "kW \n";
140     ofs << "\n";
141
142     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
143     ofs << "Capital Cost: " << this->capital_cost << " \n";
144     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
145         << " per kWh produced \n";
146     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
147         << " \n";
148     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
149         << " \n";
150     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
151     ofs << "\n";
152
153     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
154     ofs << "\n-----\n\n";
155
156     // 2.2. Renewable attributes
157     ofs << "## Renewable Attributes\n";
158     ofs << "\n";
159
160     ofs << "Resource Key (1D): " << this->resource_key << " \n";
161     ofs << "\n-----\n\n";
162
163     // 2.3. Solar attributes
164     ofs << "## Solar Attributes\n";
165     ofs << "\n";
166
167     ofs << "Derating Factor: " << this->derating << " \n";
168     ofs << "\n-----\n\n";
169
170     // 2.4. Solar Results
171     ofs << "## Results\n";
172     ofs << "\n";
173
174     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
175     ofs << "\n";
176
177     ofs << "Total Dispatch: " << this->total_dispatch_kWh
178         << " kWh \n";
179
180     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
181         << " per kWh dispatched \n";
182     ofs << "\n";
183
184     ofs << "Running Hours: " << this->running_hours << " \n";
185     ofs << "Replacements: " << this->n_replacements << " \n";
```

```

188
189     ofs << "\n-----\n\n";
190
191     ofs.close();
192     return;
193 } /* __writeSummary() */

```

4.16.3.5 __writeTimeSeries()

```

void Solar::__writeTimeSeries (
    std::string write_path,
    std::vector< double > * time_vec_hrs_ptr,
    std::map< int, std::vector< double >> * resource_map_1D_ptr,
    std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
    int max_lines = -1 ) [private], [virtual]

```

Helper method to write time series results for [Solar](#).

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>time_vec_hrs_ptr</i>	A pointer to the <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Renewable](#).

```

231 {
232     // 1. create filestream
233     write_path += "time_series_results.csv";
234     std::ofstream ofs;
235     ofs.open(write_path, std::ofstream::out);
236
237     // 2. write time series results (comma separated value)
238     ofs << "Time (since start of data) [hrs],";
239     ofs << "Solar Resource [kW/m2],";
240     ofs << "Production [kW],";
241     ofs << "Dispatch [kW],";
242     ofs << "Storage [kW],";
243     ofs << "Curtailement [kW],";
244     ofs << "Capital Cost (actual),";
245     ofs << "Operation and Maintenance Cost (actual),";
246     ofs << "\n";
247
248     for (int i = 0; i < max_lines; i++) {
249         ofs << time_vec_hrs_ptr->at(i) << ",";
250         ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ",";
251         ofs << this->production_vec_kW[i] << ",";
252         ofs << this->dispatch_vec_kW[i] << ",";
253         ofs << this->storage_vec_kW[i] << ",";
254         ofs << this->curtailment_vec_kW[i] << ",";
255         ofs << this->capital_cost_vec[i] << ",";
256         ofs << this->operation_maintenance_cost_vec[i] << ",";
257         ofs << "\n";
258     }
259
260     ofs.close();
261     return;
262 } /* __writeTimeSeries() */

```

4.16.3.6 commit()

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

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

Parameters

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

```
434 {
435     // 1. invoke base class method
436     load_kW = Renewable::commit(
437         timestep,
438         dt_hrs,
439         production_kW,
440         load_kW
441     );
442
443
444     //...
445
446     return load_kW;
447 } /* commit() */
```

4.16.3.7 computeProductionkW()

```
double Solar::computeProductionkW (
    int timestep,
    double dt_hrs,
    double solar_resource_kWm2 ) [virtual]
```

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

Ref: [HOMER \[2023f\]](#)

Parameters

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

Returns

The production [kW] of the solar PV array.

Reimplemented from [Renewable](#).

```
383 {
384     // check if no resource
385     if (solar_resource_kWm2 <= 0) {
386         return 0;
387     }
388
389     // compute production
390     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
391
392     // cap production at capacity
393     if (production_kW > this->capacity_kW) {
394         production_kW = this->capacity_kW;
395     }
396
397     return production_kW;
398 } /* computeProductionkW() */
```

4.16.4 Member Data Documentation

4.16.4.1 derating

```
double Solar::derating
```

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

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

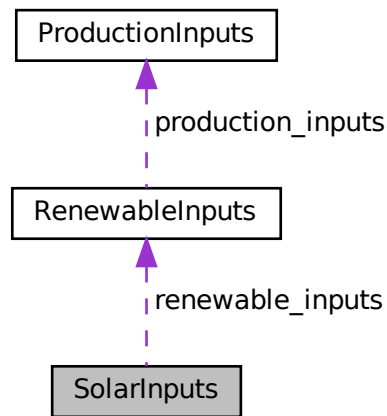
- header/Production/Renewable/[Solar.h](#)
- source/Production/Renewable/[Solar.cpp](#)

4.17 SolarInputs Struct Reference

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

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

Collaboration diagram for SolarInputs:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- `int` [resource_key](#) = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- `double` [capital_cost](#) = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- `double` [operation_maintenance_cost_kWh](#) = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- `double` [derating](#) = 0.8
The derating of the solar PV array (i.e., shadowing, soiling, etc.).

4.17.1 Detailed Description

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

4.17.2 Member Data Documentation

4.17.2.1 capital_cost

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

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

4.17.2.2 derating

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

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

4.17.2.3 operation_maintenance_cost_kWh

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

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

4.17.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.17.2.5 resource_key

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

A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.

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

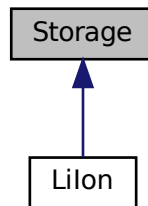
- [header/Production/Renewable/Solar.h](#)

4.18 Storage Class Reference

The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

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

Inheritance diagram for Storage:



Public Member Functions

- [Storage](#) (void)
Constructor for the [Storage](#) class.
- virtual [~Storage](#) (void)
Destructor for the [Storage](#) class.

4.18.1 Detailed Description

The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

4.18.2 Constructor & Destructor Documentation

4.18.2.1 Storage()

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

Constructor for the [Storage](#) class.

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


4.18.2.2 ~Storage()

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

Destructor for the [Storage](#) class.

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

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

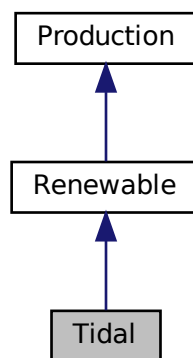
- [header/Storage/Storage.h](#)
- [source/Storage/Storage.cpp](#)

4.19 Tidal Class Reference

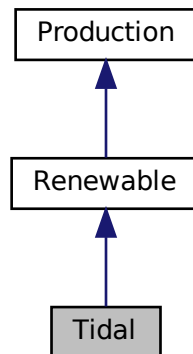
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

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

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- [Tidal](#) (void)
Constructor (dummy) for the [Tidal](#) class.
- [Tidal](#) (int, double, [TidalInputs](#))
Constructor (intended) for the [Tidal](#) class.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Tidal](#) (void)
Destructor for the [Tidal](#) class.

Public Attributes

- double [design_speed_ms](#)
The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.
- [TidalPowerProductionModel](#) [power_model](#)
The tidal power production model to be applied.
- std::string [power_model_string](#)
A string describing the active power production model.

Private Member Functions

- void [__checkInputs](#) ([TidalInputs](#))
Helper method to check inputs to the [Tidal](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic tidal turbine capital cost.

- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__computeCubicProductionkW](#) (int, double, double)
Helper method to compute tidal turbine production under a cubic production model.
- double [__computeExponentialProductionkW](#) (int, double, double)
Helper method to compute tidal turbine production under an exponential production model.
- double [__computeLookupProductionkW](#) (int, double, double)
Helper method to compute tidal turbine production by way of looking up using given power curve data.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Tidal](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
Helper method to write time series results for [Tidal](#).

4.19.1 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.

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

4.19.2.2 [Tidal\(\)](#) [2/2]

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

Constructor (intended) for the [Tidal](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>tidal_inputs</i>	A structure of Tidal constructor inputs.

```

457 :
458 Renewable(
459     n_points,
460     n_years,
461     tidal_inputs.renewable_inputs
462 )
463 {
464     // 1. check inputs
465     this->__checkInputs(tidal_inputs);
466
467     // 2. set attributes
468     this->type = RenewableType :: TIDAL;
469     this->type_str = "TIDAL";
470
471     this->resource_key = tidal_inputs.resource_key;
472
473     this->design_speed_ms = tidal_inputs.design_speed_ms;
474
475     this->power_model = tidal_inputs.power_model;
476
477     switch (this->power_model) {
478         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
479             this->power_model_string = "CUBIC";
480
481             break;
482         }
483
484         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
485             this->power_model_string = "EXPONENTIAL";
486
487             break;
488         }
489
490         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
491             this->power_model_string = "LOOKUP";
492
493             break;
494         }
495
496         default: {
497             std::string error_str = "ERROR: Tidal(): ";
498             error_str += "power production model ";
499             error_str += std::to_string(this->power_model);
500             error_str += " not recognized";
501
502             #ifdef _WIN32
503                 std::cout << error_str << std::endl;
504             #endif
505
506             throw std::runtime_error(error_str);
507
508             break;
509         }
510     }
511
512     if (tidal_inputs.capital_cost < 0) {
513         this->capital_cost = this->__getGenericCapitalCost();
514     }
515
516     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
517         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
518     }
519
520     if (not this->is_sunk) {
521         this->capital_cost_vec[0] = this->capital_cost;
522     }
523
524     // 3. construction print
525     if (this->print_flag) {
526         std::cout << "Tidal object constructed at " << this << std::endl;
527     }
528
529     return;
530 } /* Renewable() */

```

4.19.2.3 ~Tidal()

```

Tidal::~Tidal (
    void )

```

Destructor for the [Tidal](#) class.

```

684 {
685     // 1. destruction print
686     if (this->print_flag) {
687         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
688     }
689     return;
690 }
691 } /* ~Tidal() */

```

4.19.3 Member Function Documentation

4.19.3.1 __checkInputs()

```

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

```

Helper method to check inputs to the [Tidal](#) constructor.

```

37 {
38     // 1. check design_speed_ms
39     if (tidal_inputs.design_speed_ms <= 0) {
40         std::string error_str = "ERROR: Tidal(): ";
41         error_str += "TidalInputs::design_speed_ms must be > 0";
42
43         #ifdef _WIN32
44             std::cout << error_str << std::endl;
45         #endif
46
47         throw std::invalid_argument(error_str);
48     }
49
50     return;
51 } /* __checkInputs() */

```

4.19.3.2 __computeCubicProductionkW()

```

double Tidal::__computeCubicProductionkW (
    int timestep,
    double dt_hrs,
    double tidal_resource_ms ) [private]

```

Helper method to compute tidal turbine production under a cubic production model.

Ref: [Buckham et al. \[2023\]](#)

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>tidal_resource_ms</i>	The available tidal stream resource [m/s].

Returns

The production [kW] of the tidal turbine, under a cubic model.

```

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

```

4.19.3.3 __computeExponentialProductionkW()

```

double Tidal::__computeExponentialProductionkW (
    int timestep,
    double dt_hrs,
    double tidal_resource_ms ) [private]

```

Helper method to compute tidal turbine production under an exponential production model.

Ref: docs/refs/wind_tidal_wave.pdf

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>tidal_resource_ms</i>	The available tidal stream resource [m/s].

Returns

The production [kW] of the tidal turbine, under an exponential model.

```

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

```

```

212
213     return production * this->capacity_kW;
214 } /* __computeExponentialProductionkW() */

```

4.19.3.4 __computeLookupProductionkW()

```

double Tidal::__computeLookupProductionkW (
    int timestep,
    double dt_hrs,
    double tidal_resource_ms ) [private]

```

Helper method to compute tidal turbine production by way of looking up using given power curve data.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>tidal_resource_ms</i>	The available tidal stream resource [m/s].

Returns

The interpolated production [kW] of the tidal tubrine.

```

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

```

4.19.3.5 __getGenericCapitalCost()

```

double Tidal::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic tidal turbine capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: [MacDougall \[2019\]](#)

Returns

A generic capital cost for the tidal turbine [CAD].

```

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

```

4.19.3.6 `__getGenericOpMaintCost()`

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

Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: [MacDougall \[2019\]](#)

Returns

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

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

4.19.3.7 `__writeSummary()`

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

Helper method to write summary results for [Tidal](#).

Parameters

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

Reimplemented from [Renewable](#).

```
268 {
269     // 1. create filestream
270     write_path += "summary_results.md";
271     std::ofstream ofs;
272     ofs.open(write_path, std::ofstream::out);
273
274     // 2. write summary results (markdown)
275     ofs << "# ";
276     ofs << std::to_string(int(ceil(this->capacity_kW)));
277     ofs << " kW TIDAL Summary Results\n";
278     ofs << "\n-----\n\n";
279
280     // 2.1. Production attributes
281     ofs << "## Production Attributes\n";
282     ofs << "\n";
283
284     ofs << "Capacity: " << this->capacity_kW << "kW \n";
285     ofs << "\n";
286
287     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
288     ofs << "Capital Cost: " << this->capital_cost << " \n";
289     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
290         << " per kWh produced \n";
291     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
292         << " \n";
293     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
294         << " \n";
```



```

295     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
296     ofs << "\n";
297
298     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
299     ofs << "\n-----\n\n";
300
301     // 2.2. Renewable attributes
302     ofs << "## Renewable Attributes\n";
303     ofs << "\n";
304
305     ofs << "Resource Key (1D): " << this->resource_key << " \n";
306
307     ofs << "\n-----\n\n";
308
309     // 2.3. Tidal attributes
310     ofs << "## Tidal Attributes\n";
311     ofs << "\n";
312
313     ofs << "Power Production Model: " << this->power_model_string << " \n";
314     ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
315
316     ofs << "\n-----\n\n";
317
318     // 2.4. Tidal Results
319     ofs << "## Results\n";
320     ofs << "\n";
321
322     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
323     ofs << "\n";
324
325     ofs << "Total Dispatch: " << this->total_dispatch_kWh
326         << " kWh \n";
327
328     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
329         << " per kWh dispatched \n";
330     ofs << "\n";
331
332     ofs << "Running Hours: " << this->running_hours << " \n";
333     ofs << "Replacements: " << this->n_replacements << " \n";
334
335     ofs << "\n-----\n\n";
336
337     ofs.close();
338
339     return;
340 } /* __writeSummary() */

```

4.19.3.8 __writeTimeSeries()

```

void Tidal::__writeTimeSeries (
    std::string write_path,
    std::vector< double > * time_vec_hrs_ptr,
    std::map< int, std::vector< double >> * resource_map_1D_ptr,
    std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
    int max_lines = -1 ) [private], [virtual]

```

Helper method to write time series results for [Tidal](#).

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>time_vec_hrs_ptr</i>	A pointer to the <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Renewable](#).

```

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

```

4.19.3.9 commit()

```

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

```

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

Parameters

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

```

656 {
657     // 1. invoke base class method
658     load_kW = Renewable::commit(
659         timestep,
660         dt_hrs,
661         production_kW,
662         load_kW
663     );
664
665

```

```

666     //...
667
668     return load_kW;
669 } /* commit() */

```

4.19.3.10 computeProductionkW()

```

double Tidal::computeProductionkW (
    int timestep,
    double dt_hrs,
    double tidal_resource_ms ) [virtual]

```

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

Parameters

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

Returns

The production [kW] of the tidal turbine.

Reimplemented from [Renewable](#).

```

562 {
563     // check if no resource
564     if (tidal_resource_ms <= 0) {
565         return 0;
566     }
567
568     // compute production
569     double production_kW = 0;
570
571     switch (this->power_model) {
572         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
573             production_kW = this->__computeCubicProductionkW(
574                 timestep,
575                 dt_hrs,
576                 tidal_resource_ms
577             );
578
579             break;
580         }
581
582
583         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
584             production_kW = this->__computeExponentialProductionkW(
585                 timestep,
586                 dt_hrs,
587                 tidal_resource_ms
588             );
589
590             break;
591         }
592
593         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
594             production_kW = this->__computeLookupProductionkW(
595                 timestep,
596                 dt_hrs,
597                 tidal_resource_ms
598             );
599
600             break;
601         }

```

```

602
603     default: {
604         std::string error_str = "ERROR: Tidal::computeProductionkW(): ";
605         error_str += "power model ";
606         error_str += std::to_string(this->power_model);
607         error_str += " not recognized";
608
609         #ifdef _WIN32
610             std::cout << error_str << std::endl;
611         #endif
612
613         throw std::runtime_error(error_str);
614
615         break;
616     }
617 }
618
619 return production_kW;
620 } /* 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.

4.19.4.3 power_model_string

```
std::string Tidal::power_model_string
```

A string describing the active power production model.

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

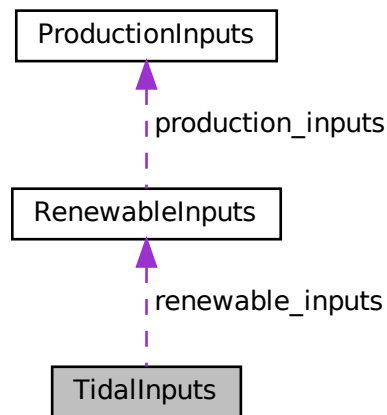
- [header/Production/Renewable/Tidal.h](#)
- [source/Production/Renewable/Tidal.cpp](#)

4.20 TidalInputs Struct Reference

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

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

Collaboration diagram for TidalInputs:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- int `resource_key` = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- double `capital_cost` = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double `operation_maintenance_cost_kWh` = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double `design_speed_ms` = 3
The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.
- [TidalPowerProductionModel](#) `power_model` = [TidalPowerProductionModel](#) :: `TIDAL_POWER_CUBIC`
The tidal power production model to be applied.

4.20.1 Detailed Description

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

4.20.2 Member Data Documentation

4.20.2.1 capital_cost

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

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

4.20.2.2 design_speed_ms

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

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

4.20.2.3 operation_maintenance_cost_kWh

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

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

4.20.2.4 power_model

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

The tidal power production model to be applied.

4.20.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.20.2.6 resource_key

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

A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.

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

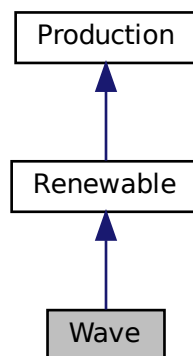
- [header/Production/Renewable/Tidal.h](#)

4.21 Wave Class Reference

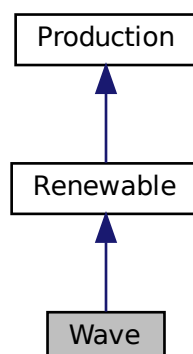
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

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

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

- [Wave](#) (void)
Constructor (dummy) for the [Wave](#) class.
- [Wave](#) (int, double, [WaveInputs](#))
Constructor (intended) for the [Wave](#) class.
- double [computeProductionkW](#) (int, double, double, double)
Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Wave](#) (void)
Destructor for the [Wave](#) class.

Public Attributes

- double [design_significant_wave_height_m](#)
The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- double [design_energy_period_s](#)
The energy period [s] at which the wave energy converter achieves its rated capacity.
- [WavePowerProductionModel](#) [power_model](#)
The wave power production model to be applied.
- std::string [power_model_string](#)
A string describing the active power production model.

Private Member Functions

- void [__checkInputs](#) ([WaveInputs](#))
Helper method to check inputs to the [Wave](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic wave energy converter capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__computeGaussianProductionkW](#) (int, double, double, double)
Helper method to compute wave energy converter production under a Gaussian production model.
- double [__computeParaboloidProductionkW](#) (int, double, double, double)
Helper method to compute wave energy converter production under a paraboloid production model.
- double [__computeLookupProductionkW](#) (int, double, double, double)
Helper method to compute wave energy converter production by way of looking up using given performance matrix.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Wave](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
Helper method to write time series results for [Wave](#).

4.21.1 Detailed Description

A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Wave() [1/2]

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

Constructor (dummy) for the [Wave](#) class.

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

4.21.2.2 Wave() [2/2]

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

Constructor (intended) for the [Wave](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>wave_inputs</i>	A structure of Wave constructor inputs.

```
510 :
511 Renewable(
512     n_points,
513     n_years,
514     wave_inputs.renewable_inputs
515 )
516 {
517     // 1. check inputs
518     this->__checkInputs(wave_inputs);
519
520     // 2. set attributes
521     this->type = RenewableType :: WAVE;
522     this->type_str = "WAVE";
523
524     this->resource_key = wave_inputs.resource_key;
525
526     this->design_significant_wave_height_m =
527         wave_inputs.design_significant_wave_height_m;
528     this->design_energy_period_s = wave_inputs.design_energy_period_s;
529
530     this->power_model = wave_inputs.power_model;
531
532     switch (this->power_model) {
533         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
534             this->power_model_string = "GAUSSIAN";
535
536             break;
537         }
538
539         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
540             this->power_model_string = "PARABOLOID";
541
542             break;
543         }
544     }
```

```

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

```

4.21.2.3 ~Wave()

```

Wave::~Wave (
    void )

```

Destructor for the [Wave](#) class.

```

745 {
746     // 1. destruction print
747     if (this->print_flag) {
748         std::cout << "Wave object at " << this << " destroyed" << std::endl;
749     }
750
751     return;
752 } /* ~Wave() */

```

4.21.3 Member Function Documentation

4.21.3.1 __checkInputs()

```

void Wave::__checkInputs (
    WaveInputs wave_inputs ) [private]

```

Helper method to check inputs to the [Wave](#) constructor.

Parameters

<i>wave_inputs</i>	A structure of Wave constructor inputs.
--------------------	---

```

39 {
40     // 1. check design_significant_wave_height_m
41     if (wave_inputs.design_significant_wave_height_m <= 0) {
42         std::string error_str = "ERROR: Wave(): ";
43         error_str += "WaveInputs::design_significant_wave_height_m must be > 0";
44
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check design_energy_period_s
53     if (wave_inputs.design_energy_period_s <= 0) {
54         std::string error_str = "ERROR: Wave(): ";
55         error_str += "WaveInputs::design_energy_period_s must be > 0";
56
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     return;
65 } /* __checkInputs() */

```

4.21.3.2 __computeGaussianProductionkW()

```

double Wave::__computeGaussianProductionkW (
    int timestep,
    double dt_hrs,
    double significant_wave_height_m,
    double energy_period_s ) [private]

```

Helper method to compute wave energy converter production under a Gaussian production model.

Ref: docs/refs/wind_tidal_wave.pdf

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under an exponential model.

```

160 {
161     double H_s_nondim =
162         (significant_wave_height_m - this->design_significant_wave_height_m) /
163         this->design_significant_wave_height_m;
164
165     double T_e_nondim =
166         (energy_period_s - this->design_energy_period_s) /

```

```

167         this->design_energy_period_s;
168
169     double production = exp(
170         -2.25119 * pow(T_e_nondim, 2) +
171         3.44570 * T_e_nondim * H_s_nondim -
172         4.01508 * pow(H_s_nondim, 2)
173     );
174
175     return production * this->capacity_kW;
176 } /* __computeGaussianProductionkW() */

```

4.21.3.3 __computeLookupProductionkW()

```

double Wave::__computeLookupProductionkW (
    int timestep,
    double dt_hrs,
    double significant_wave_height_m,
    double energy_period_s ) [private]

```

Helper method to compute wave energy converter production by way of looking up using given performance matrix.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The interpolated production [kW] of the wave energy converter.

```

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

```

4.21.3.4 __computeParaboloidProductionkW()

```

double Wave::__computeParaboloidProductionkW (
    int timestep,
    double dt_hrs,
    double significant_wave_height_m,
    double energy_period_s ) [private]

```

Helper method to compute wave energy converter production under a paraboloid production model.

Ref: [Robertson et al. \[2021\]](#)

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under a paraboloid model.

```

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

```

4.21.3.5 __getGenericCapitalCost()

```

double Wave::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic wave energy converter capital cost.

Note that this model expresses cost in terms of Canadian dollars [CAD].

Ref: [MacDougall \[2019\]](#)

Returns

A generic capital cost for the wave energy converter [CAD].

```

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

```

4.21.3.6 `__getGenericOpMaintCost()`

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

Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.

Note that this model expresses cost in terms of Canadian dollars [CAD/kWh].

Ref: [MacDougall \[2019\]](#)

Returns

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

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

4.21.3.7 `__writeSummary()`

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

Helper method to write summary results for [Wave](#).

Parameters

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

Reimplemented from [Renewable](#).

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

```

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

```

4.21.3.8 __writeTimeSeries()

```

void Wave::__writeTimeSeries (
    std::string write_path,
    std::vector< double > * time_vec_hrs_ptr,
    std::map< int, std::vector< double >> * resource_map_1D_ptr,
    std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
    int max_lines = -1 ) [private], [virtual]

```

Helper method to write time series results for [Wave](#).

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>time_vec_hrs_ptr</i>	A pointer to the <i>time_vec_hrs</i> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Renewable](#).

```

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

```

4.21.3.9 commit()

```

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

```

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

Parameters

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

```

717 {
718     // 1. invoke base class method
719     load_kW = Renewable::commit(
720         timestep,
721         dt_hrs,
722         production_kW,
723         load_kW
724     );
725
726
727     //...
728
729     return load_kW;
730 } /* commit() */

```

4.21.3.10 computeProductionkW()

```

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

```

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

Parameters

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

Returns

The production [kW] of the wave turbine.

Reimplemented from [Renewable](#).

```

621 {
622     // check if no resource
623     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
624         return 0;
625     }
626
627     // compute production
628     double production_kW = 0;
629
630     switch (this->power_model) {
631         case (WavePowerProductionModel::WAVE_POWER_PARABOLOID): {
632             production_kW = this->__computeParaboloidProductionkW(
633                 timestep,
634                 dt_hrs,
635                 significant_wave_height_m,
636                 energy_period_s
637             );
638         }

```

```

639         break;
640     }
641
642     case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
643         production_kW = this->__computeGaussianProductionkW(
644             timestep,
645             dt_hrs,
646             significant_wave_height_m,
647             energy_period_s
648         );
649
650         break;
651     }
652
653     case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
654         production_kW = this->__computeLookupProductionkW(
655             timestep,
656             dt_hrs,
657             significant_wave_height_m,
658             energy_period_s
659         );
660
661         break;
662     }
663
664     default: {
665         std::string error_str = "ERROR: Wave::computeProductionkW(): ";
666         error_str += "power model ";
667         error_str += std::to_string(this->power_model);
668         error_str += " not recognized";
669
670         #ifdef _WIN32
671             std::cout << error_str << std::endl;
672         #endif
673
674         throw std::runtime_error(error_str);
675
676         break;
677     }
678 }
679
680 return production_kW;
681 } /* 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.

4.21.4.4 power_model_string

`std::string` `Wave::power_model_string`

A string describing the active power production model.

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

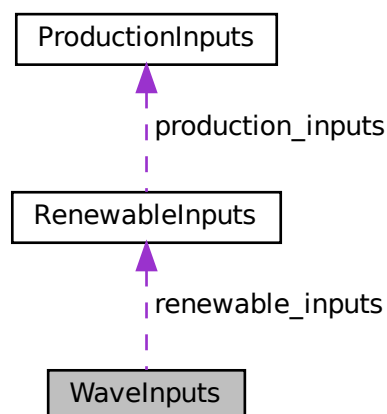
- `header/Production/Renewable/Wave.h`
- `source/Production/Renewable/Wave.cpp`

4.22 WaveInputs Struct Reference

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

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

Collaboration diagram for `WaveInputs`:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- `int resource_key = 0`
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- `double capital_cost = -1`
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- `double operation_maintenance_cost_kWh = -1`
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- `double design_significant_wave_height_m = 3`
The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- `double design_energy_period_s = 10`
The energy period [s] at which the wave energy converter achieves its rated capacity.
- [WavePowerProductionModel](#) `power_model = WavePowerProductionModel :: WAVE_POWER_PARABOLOID`
The wave power production model to be applied.

4.22.1 Detailed Description

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

4.22.2 Member Data Documentation

4.22.2.1 capital_cost

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

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

4.22.2.2 design_energy_period_s

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

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

4.22.2.3 design_significant_wave_height_m

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

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

4.22.2.4 operation_maintenance_cost_kWh

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

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

4.22.2.5 power_model

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

The wave power production model to be applied.

4.22.2.6 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.22.2.7 resource_key

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

A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.

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

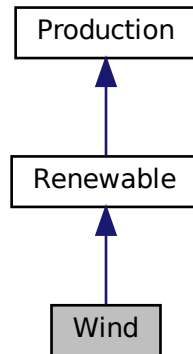
- [header/Production/Renewable/Wave.h](#)

4.23 Wind Class Reference

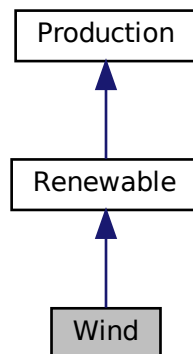
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

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

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- [Wind](#) (void)
Constructor (dummy) for the [Wind](#) class.
- [Wind](#) (int, double, [WindInputs](#))

Constructor (intended) for the [Wind](#) class.

- double [computeProductionkW](#) (int, double, double)
Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Wind](#) (void)
Destructor for the [Wind](#) class.

Public Attributes

- double [design_speed_ms](#)
The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- [WindPowerProductionModel](#) [power_model](#)
The wind power production model to be applied.
- std::string [power_model_string](#)
A string describing the active power production model.

Private Member Functions

- void [__checkInputs](#) ([WindInputs](#))
Helper method to check inputs to the [Wind](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic wind turbine capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__computeExponentialProductionkW](#) (int, double, double)
Helper method to compute wind turbine production under an exponential production model.
- double [__computeLookupProductionkW](#) (int, double, double)
Helper method to compute wind turbine production by way of looking up using given power curve data.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Wind](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
Helper method to write time series results for [Wind](#).

4.23.1 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.

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

4.23.2.2 Wind() [2/2]

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

Constructor (intended) for the [Wind](#) class.

Parameters

<i>n_points</i>	The number of points in the modelling time series.
<i>n_years</i>	The number of years being modelled.
<i>wind_inputs</i>	A structure of Wind constructor inputs.

```
420 :
421 Renewable(
422     n_points,
423     n_years,
424     wind_inputs.renewable_inputs
425 )
426 {
427     // 1. check inputs
428     this->__checkInputs(wind_inputs);
429
430     // 2. set attributes
431     this->type = RenewableType :: WIND;
432     this->type_str = "WIND";
433
434     this->resource_key = wind_inputs.resource_key;
435
436     this->design_speed_ms = wind_inputs.design_speed_ms;
437
438     this->power_model = wind_inputs.power_model;
439
440     switch (this->power_model) {
441         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
442             this->power_model_string = "EXPONENTIAL";
443
444             break;
445         }
446
447         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
448             this->power_model_string = "LOOKUP";
449
450             break;
451         }
452
453         default: {
454             std::string error_str = "ERROR: Wind(): ";
455             error_str += "power production model ";
456             error_str += std::to_string(this->power_model);
457             error_str += " not recognized";
458
459             #ifdef _WIN32
460                 std::cout << error_str << std::endl;
461             #endif
462
463             throw std::runtime_error(error_str);
```



```

464
465         break;
466     }
467 }
468
469 if (wind_inputs.capital_cost < 0) {
470     this->capital_cost = this->__getGenericCapitalCost();
471 }
472
473 if (wind_inputs.operation_maintenance_cost_kWh < 0) {
474     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
475 }
476
477 if (not this->is_sunk) {
478     this->capital_cost_vec[0] = this->capital_cost;
479 }
480
481 // 3. construction print
482 if (this->print_flag) {
483     std::cout << "Wind object constructed at " << this << std::endl;
484 }
485
486 return;
487 } /* Renewable() */

```

4.23.2.3 ~Wind()

```

Wind::~Wind (
    void )

```

Destructor for the [Wind](#) class.

```

630 {
631     // 1. destruction print
632     if (this->print_flag) {
633         std::cout << "Wind object at " << this << " destroyed" << std::endl;
634     }
635
636     return;
637 } /* ~Wind() */

```

4.23.3 Member Function Documentation

4.23.3.1 __checkInputs()

```

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

```

Helper method to check inputs to the [Wind](#) constructor.

Parameters

<i>wind_inputs</i>	A structure of Wind constructor inputs.
--------------------	---

```

39 {
40     // 1. check design_speed_ms
41     if (wind_inputs.design_speed_ms <= 0) {
42         std::string error_str = "ERROR: Wind(): ";
43         error_str += "WindInputs::design_speed_ms must be > 0";
44
45         #ifdef _WIN32
46         std::cout << error_str << std::endl;

```

```

47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     return;
53 } /* __checkInputs() */

```

4.23.3.2 __computeExponentialProductionkW()

```

double Wind::__computeExponentialProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]

```

Helper method to compute wind turbine production under an exponential production model.

Ref: docs/refs/wind_tidal_wave.pdf

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>wind_resource_ms</i>	The available wind resource [m/s].

Returns

The production [kW] of the wind turbine, under an exponential model.

```

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

```

4.23.3.3 __computeLookupProductionkW()

```

double Wind::__computeLookupProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]

```

Helper method to compute wind turbine production by way of looking up using given power curve data.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>wind_resource_ms</i>	The available wind resource [m/s].

Returns

The interpolated production [kW] of the wind turbine.

```

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

```

4.23.3.4 __getGenericCapitalCost()

```

double Wind::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic wind turbine capital cost.

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

Returns

A generic capital cost for the wind turbine [CAD].

```

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

```

4.23.3.5 __getGenericOpMaintCost()

```

double Wind::__getGenericOpMaintCost (
    void ) [private]

```

Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.

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

Returns

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

```

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

```

4.23.3.6 __writeSummary()

```

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

```

Helper method to write summary results for [Wind](#).

Parameters

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

Reimplemented from [Renewable](#).

```

213 {
214     // 1. create filestream
215     write_path += "summary_results.md";
216     std::ofstream ofs;
217     ofs.open(write_path, std::ofstream::out);
218
219     // 2. write summary results (markdown)
220     ofs << "# ";
221     ofs << std::to_string(int(ceil(this->capacity_kW)));
222     ofs << " kW WIND Summary Results\n";
223     ofs << "\n-----\n\n";
224
225     // 2.1. Production attributes
226     ofs << "## Production Attributes\n";
227     ofs << "\n";
228
229     ofs << "Capacity: " << this->capacity_kW << "kW \n";
230     ofs << "\n";
231
232     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
233     ofs << "Capital Cost: " << this->capital_cost << " \n";
234     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
235         << " per kWh produced \n";
236     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
237         << " \n";
238     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
239         << " \n";
240     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
241     ofs << "\n";
242
243     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
244     ofs << "\n-----\n\n";
245
246     // 2.2. Renewable attributes
247     ofs << "## Renewable Attributes\n";
248     ofs << "\n";
249
250     ofs << "Resource Key (ID): " << this->resource_key << " \n";
251     ofs << "\n-----\n\n";
252
253     // 2.3. Wind attributes
254     ofs << "## Wind Attributes\n";
255     ofs << "\n";
256
257     ofs << "Power Production Model: " << this->power_model_string << " \n";
258     switch (this->power_model) {
259     case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
260         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
261         break;
262     }
263     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
264         //...
265         break;
266     }
267     default: {
268         // write nothing!
269         break;
270     }
271     }
272     ofs << "\n-----\n\n";
273
274     // 2.4. Wind Results
275     ofs << "## Results\n";
276     ofs << "\n";
277
278     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
279     ofs << "\n";
280
281     ofs << "Total Dispatch: " << this->total_dispatch_kWh

```

```

290         « " kWh  \n";
291
292     ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
293         « " per kWh dispatched  \n";
294     ofs « "\n";
295
296     ofs « "Running Hours: " « this->running_hours « "  \n";
297     ofs « "Replacements: " « this->n_replacements « "  \n";
298
299     ofs « "\n-----\n\n";
300
301     ofs.close();
302
303     return;
304 } /* __writeSummary() */

```

4.23.3.7 __writeTimeSeries()

```

void Wind::__writeTimeSeries (
    std::string write_path,
    std::vector< double > * time_vec_hrs_ptr,
    std::map< int, std::vector< double >> * resource_map_1D_ptr,
    std::map< int, std::vector< std::vector< double >>> * resource_map_2D_ptr,
    int max_lines = -1 ) [private], [virtual]

```

Helper method to write time series results for [Wind](#).

Parameters

<i>write_path</i>	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
<i>time_vec_hrs_ptr</i>	A pointer to the <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Renewable](#).

```

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

```

```
372 } /* __writeTimeSeries() */
```

4.23.3.8 commit()

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

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

Parameters

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

```
602 {
603     // 1. invoke base class method
604     load_kW = Renewable::commit(
605         timestep,
606         dt_hrs,
607         production_kW,
608         load_kW
609     );
610
611
612     //...
613
614     return load_kW;
615 } /* commit() */
```

4.23.3.9 computeProductionkW()

```
double Wind::computeProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [virtual]
```

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

Parameters

<i>timestep</i>	The timestep (i.e., time series index) for the request.
<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
<i>wind_resource_ms</i>	Wind resource (i.e. wind speed) [m/s].

Returns

The production [kW] of the wind turbine.

Reimplemented from [Renewable](#).

```

519 {
520     // check if no resource
521     if (wind_resource_ms <= 0) {
522         return 0;
523     }
524
525     // compute production
526     double production_kW = 0;
527
528     switch (this->power_model) {
529         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
530             production_kW = this->__computeExponentialProductionkW(
531                 timestep,
532                 dt_hrs,
533                 wind_resource_ms
534             );
535
536             break;
537         }
538
539         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
540             production_kW = this->__computeLookupProductionkW(
541                 timestep,
542                 dt_hrs,
543                 wind_resource_ms
544             );
545
546             break;
547         }
548
549         default: {
550             std::string error_str = "ERROR: Wind::computeProductionkW(): ";
551             error_str += "power model ";
552             error_str += std::to_string(this->power_model);
553             error_str += " not recognized";
554
555             #ifdef _WIN32
556                 std::cout << error_str << std::endl;
557             #endif
558
559             throw std::runtime_error(error_str);
560
561             break;
562         }
563     }
564
565     return production_kW;
566 } /* 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.

4.23.4.3 power_model_string

```
std::string Wind::power_model_string
```

A string describing the active power production model.

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

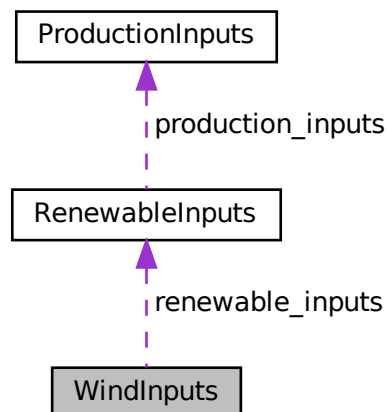
- [header/Production/Renewable/Wind.h](#)
- [source/Production/Renewable/Wind.cpp](#)

4.24 WindInputs Struct Reference

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

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

Collaboration diagram for WindInputs:



Public Attributes

- [RenewableInputs renewable_inputs](#)
An encapsulated [RenewableInputs](#) instance.
- int [resource_key](#) = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- double [capital_cost](#) = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double [operation_maintenance_cost_kWh](#) = -1

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

- double `design_speed_ms` = 8

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

- `WindPowerProductionModel power_model` = `WindPowerProductionModel :: WIND_POWER_EXPONENTIAL`

The wind power production model to be applied.

4.24.1 Detailed Description

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

4.24.2 Member Data Documentation

4.24.2.1 capital_cost

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

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

4.24.2.2 design_speed_ms

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

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

4.24.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

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

4.24.2.4 power_model

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

The wind power production model to be applied.

4.24.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.24.2.6 resource_key

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

A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.

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

- [header/Production/Renewable/Wind.h](#)

Chapter 5

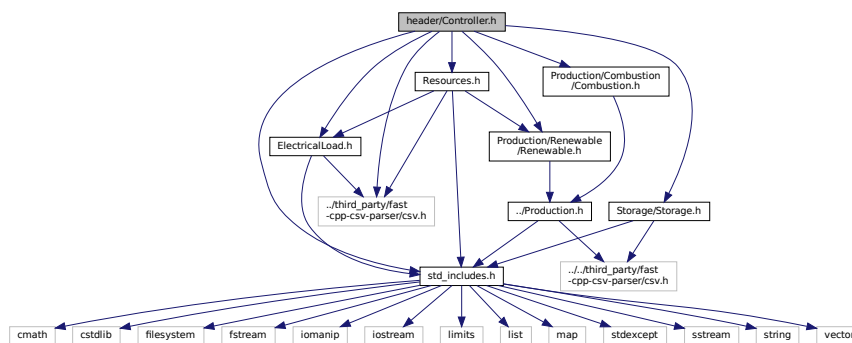
File Documentation

5.1 header/Controller.h File Reference

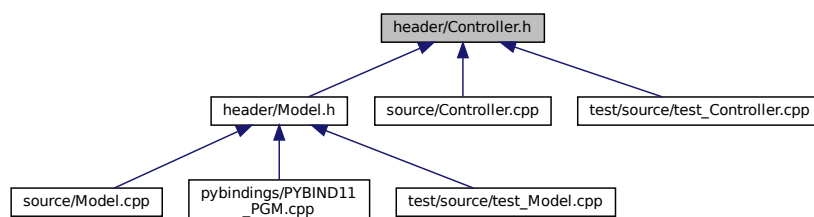
Header file the [Controller](#) class.

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

Include dependency graph for Controller.h:



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



Classes

- class [Controller](#)

A class which contains a various dispatch control logic. Intended to serve as a component class of [Model](#).

Enumerations

- enum [ControlMode](#) { [LOAD_FOLLOWING](#) , [CYCLE_CHARGING](#) , [N_CONTROL_MODES](#) }

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

5.1.1 Detailed Description

Header file the [Controller](#) class.

5.1.2 Enumeration Type Documentation

5.1.2.1 ControlMode

enum [ControlMode](#)

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

Enumerator

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

```

43         {
44     LOAD\_FOLLOWING,
45     CYCLE\_CHARGING,
46     N\_CONTROL\_MODES
47 };

```

5.2 header/ElectricalLoad.h File Reference

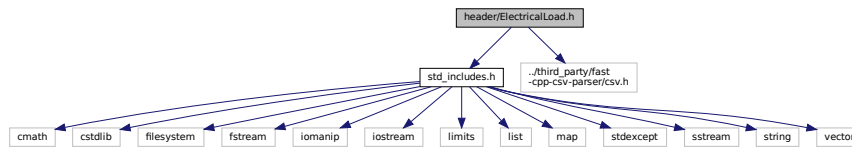
Header file the [ElectricalLoad](#) class.

```

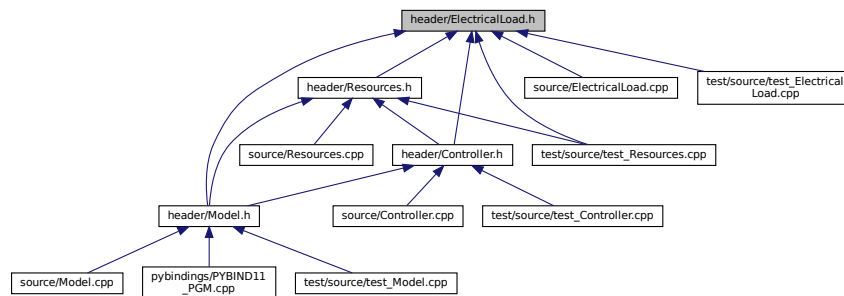
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"

```

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.

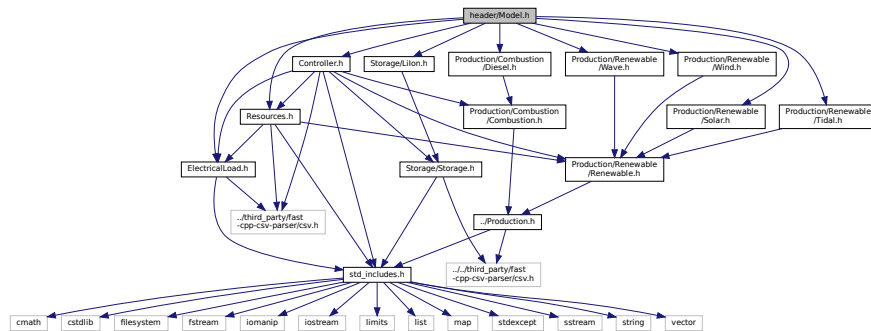
```

#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"

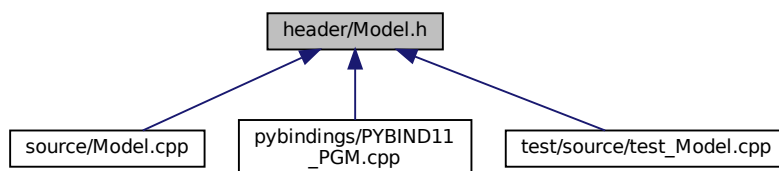
```

```
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:



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



Classes

- struct [ModellInputs](#)

A structure which bundles the necessary inputs for the [Model](#) constructor. Provides default values for every necessary input (except `path_2_electrical_load_time_series`, for which a valid input must be provided).

- class [Model](#)

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

5.3.1 Detailed Description

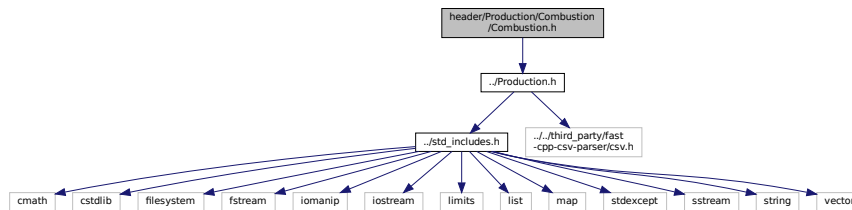
Header file the [Model](#) class.

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

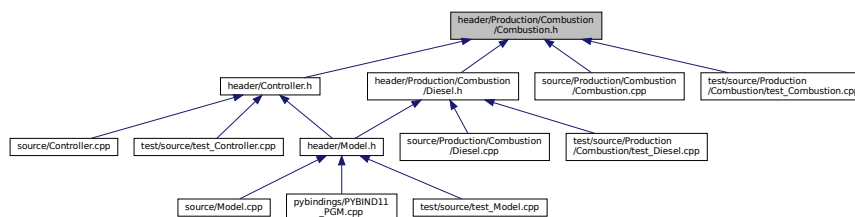
Header file the [Combustion](#) class.

```
#include "../Production.h"
```

Include dependency graph for Combustion.h:



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



Classes

- struct [CombustionInputs](#)
A structure which bundles the necessary inputs for the [Combustion](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [ProductionInputs](#).
- struct [Emissions](#)
A structure which bundles the emitted masses of various emissions chemistries.
- class [Combustion](#)
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

Enumerations

- enum [CombustionType](#) { DIESEL , N_COMBUSTION_TYPES }
An enumeration of the types of [Combustion](#) asset supported by PGMcpp.

5.4.1 Detailed Description

Header file the [Combustion](#) class.

5.4.2 Enumeration Type Documentation

5.4.2.1 CombustionType

```
enum CombustionType
```

An enumeration of the types of [Combustion](#) asset supported by PGMcpp.

Enumerator

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

```

33     {
34     DIESEL,
35     N_COMBUSTION_TYPES
36 };

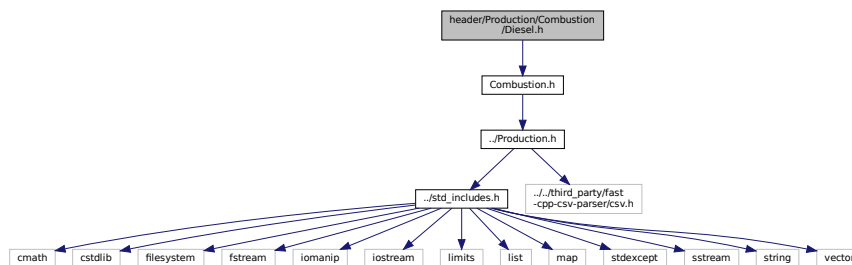
```

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

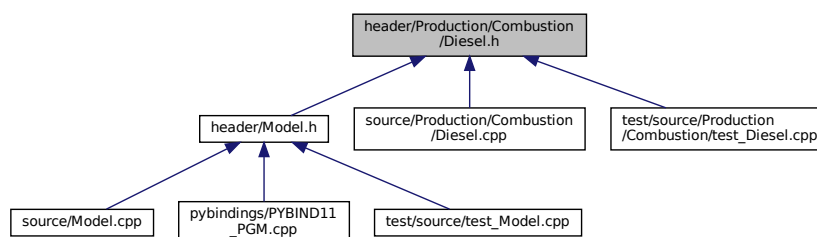
Header file the [Diesel](#) class.

```
#include "Combustion.h"
```

Include dependency graph for Diesel.h:



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



Classes

- struct [DieselInputs](#)

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

- class [Diesel](#)

A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

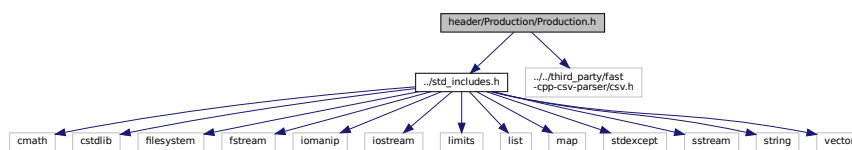
5.5.1 Detailed Description

Header file the [Diesel](#) class.

5.6 header/Production/Production.h File Reference

Header file the [Production](#) class.

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



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



Classes

- struct [ProductionInputs](#)
A structure which bundles the necessary inputs for the [Production](#) constructor. Provides default values for every necessary input.
- class [Production](#)
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.6.1 Detailed Description

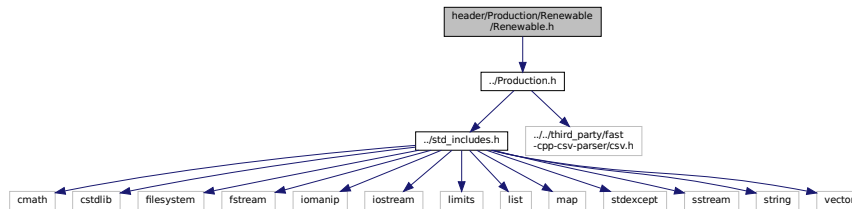
Header file the [Production](#) class.

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

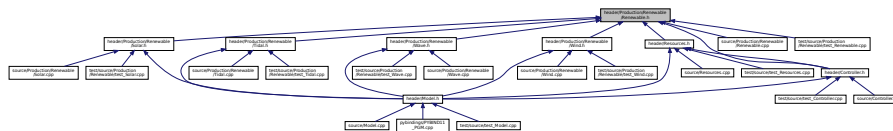
Header file the [Renewable](#) class.

```
#include "../Production.h"
```

Include dependency graph for Renewable.h:



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



Classes

- struct [RenewableInputs](#)

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

- class [Renewable](#)

The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

Enumerations

- enum [RenewableType](#) {
[SOLAR](#) , [TIDAL](#) , [WAVE](#) , [WIND](#) ,
[N_RENEWABLE_TYPES](#) }

An enumeration of the types of [Renewable](#) asset supported by PGMcpp.

5.7.1 Detailed Description

Header file the [Renewable](#) class.

5.7.2 Enumeration Type Documentation

5.7.2.1 RenewableType

```
enum RenewableType
```

An enumeration of the types of [Renewable](#) asset supported by PGMcpp.

Enumerator

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

```

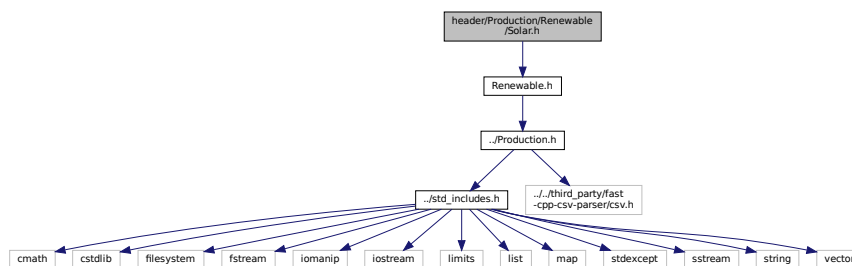
33     {
34     SOLAR,
35     TIDAL,
36     WAVE,
37     WIND,
38     N_RENEWABLE_TYPES
39 };

```

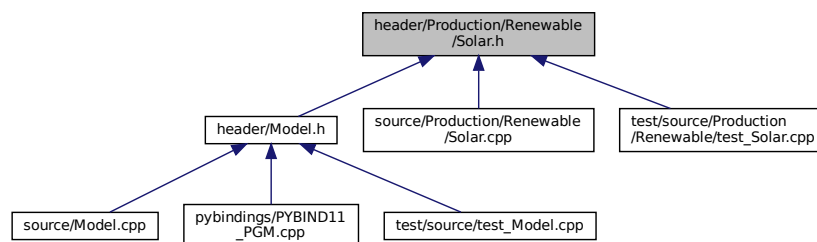
5.8 header/Production/Renewable/Solar.h File Reference

Header file the [Solar](#) class.

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



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



Classes

- struct [SolarInputs](#)

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

- class [Solar](#)

A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

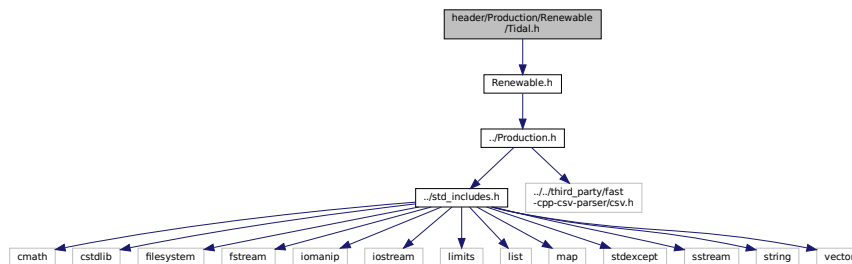
5.8.1 Detailed Description

Header file the [Solar](#) class.

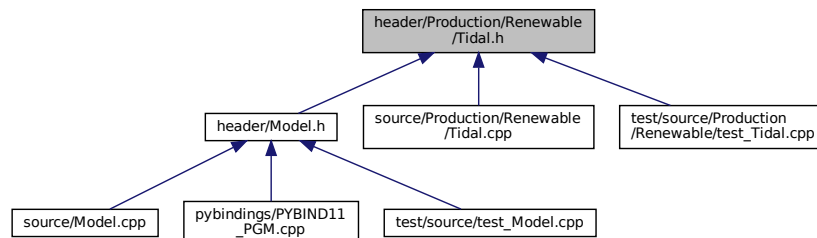
5.9 header/Production/Renewable/Tidal.h File Reference

Header file the [Tidal](#) class.

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



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



Classes

- struct [TidalInputs](#)

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

- class [Tidal](#)

A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

Enumerations

- enum [TidalPowerProductionModel](#) { [TIDAL_POWER_CUBIC](#) , [TIDAL_POWER_EXPONENTIAL](#) , [TIDAL_POWER_LOOKUP](#) , [N_TIDAL_POWER_PRODUCTION_MODELS](#) }

5.9.1 Detailed Description

Header file the [Tidal](#) class.

5.9.2 Enumeration Type Documentation

5.9.2.1 TidalPowerProductionModel

enum [TidalPowerProductionModel](#)

Enumerator

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

```

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

```

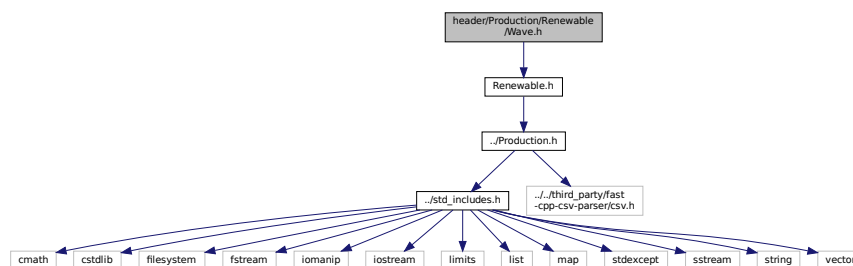
5.10 header/Production/Renewable/Wave.h File Reference

Header file the [Wave](#) class.

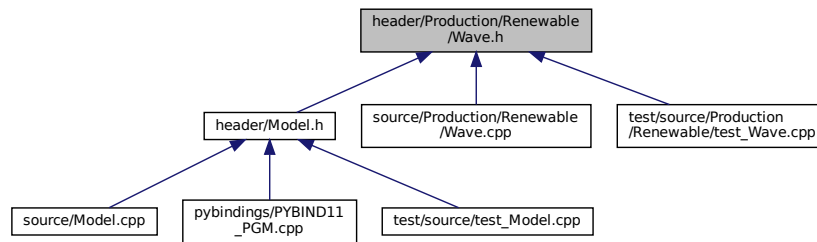
```

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

```



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



Classes

- struct [WaveInputs](#)
A structure which bundles the necessary inputs for the [Wave](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [RenewableInputs](#).
- class [Wave](#)
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

Enumerations

- enum [WavePowerProductionModel](#) { [WAVE_POWER_GAUSSIAN](#) , [WAVE_POWER_PARABOLOID](#) , [WAVE_POWER_LOOKUP](#) , [N_WAVE_POWER_PRODUCTION_MODELS](#) }

5.10.1 Detailed Description

Header file the [Wave](#) class.

5.10.2 Enumeration Type Documentation

5.10.2.1 WavePowerProductionModel

enum [WavePowerProductionModel](#)

Enumerator

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

```

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

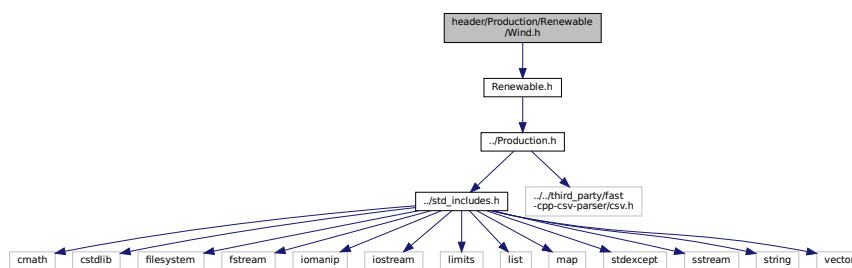
```

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

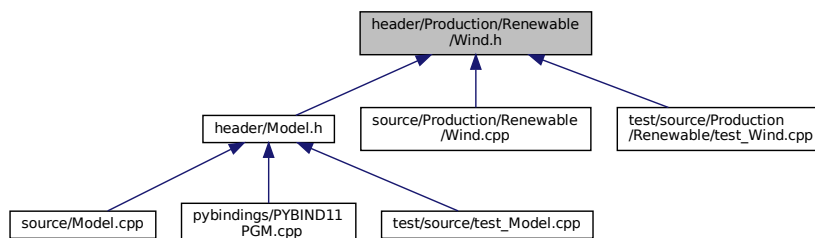
Header file the [Wind](#) class.

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

Include dependency graph for Wind.h:



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



Classes

- struct [WindInputs](#)

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

- class [Wind](#)

A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

Enumerations

- enum [WindPowerProductionModel](#) { [WIND_POWER_EXPONENTIAL](#) , [WIND_POWER_LOOKUP](#) , [N_WIND_POWER_PRODUCTION_MODELS](#) }

5.11.1 Detailed Description

Header file the [Wind](#) class.

5.11.2 Enumeration Type Documentation

5.11.2.1 WindPowerProductionModel

enum [WindPowerProductionModel](#)

Enumerator

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

```

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

```

5.12 header/Resources.h File Reference

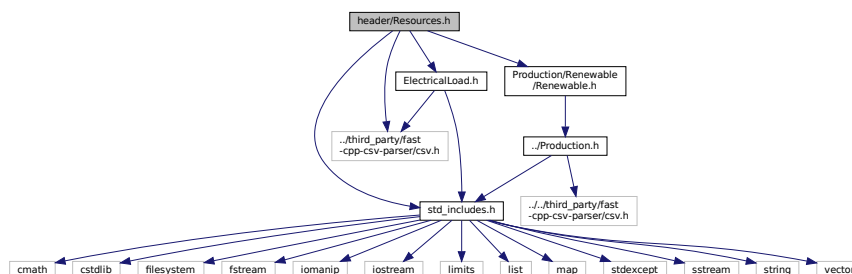
Header file the [Resources](#) class.

```

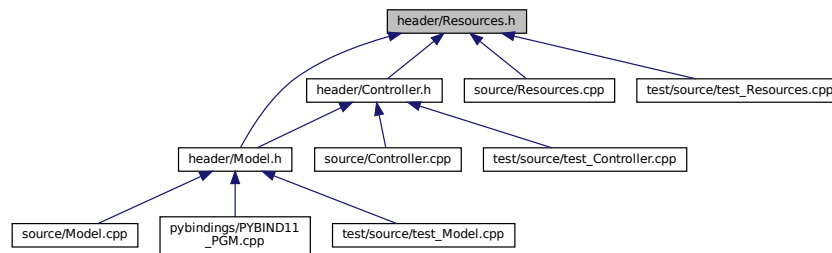
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/Renewable/Renewable.h"

```

Include dependency graph for Resources.h:



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



Classes

- class [Resources](#)

A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

5.12.1 Detailed Description

Header file the [Resources](#) class.

5.13 header/std_includes.h File Reference

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

```

#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <stdexcept>
#include <sstream>
#include <string>
#include <vector>
  
```

Include dependency graph for std_includes.h:



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



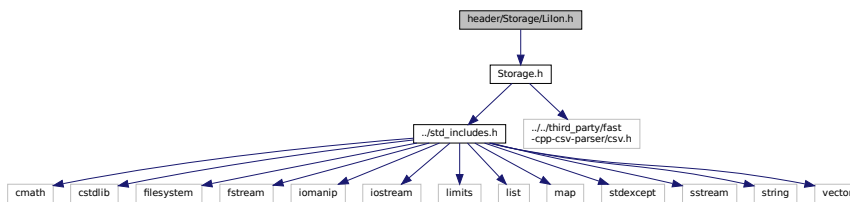
5.13.1 Detailed Description

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

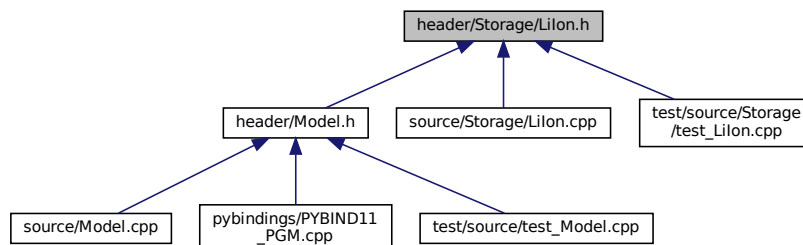
5.14 header/Storage/Lilon.h File Reference

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

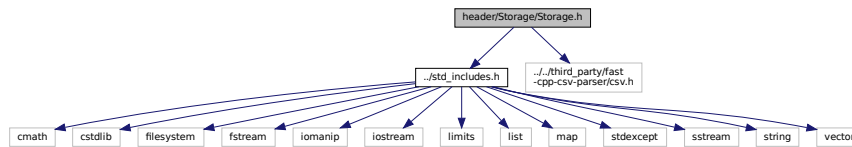
Header file the [Lilon](#) class.

5.15 header/Storage/Storage.h File Reference

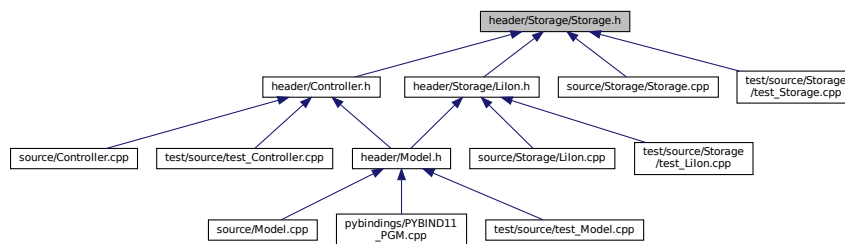
Header file the [Storage](#) class.

```
#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>
```



```

46     .def_readwrite("mean_load_kW", &ElectricalLoad::mean_load_kW)
47     .def_readwrite("min_load_kW", &ElectricalLoad::min_load_kW)
48     .def_readwrite("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs)
49     .def_readwrite("load_vec_kW", &ElectricalLoad::load_vec_kW)
50     .def_readwrite("time_vec_hrs", &ElectricalLoad::time_vec_hrs)
51
52     .def(pybind11::init<std::string>());
53 */
54 // ===== END ElectricalLoad ===== //
55
56
57
58 // ===== Model ===== //
59 /*
60 pybind11::class_<Model>(m, "Model")
61     .def(
62         pybind11::init<
63             ElectricalLoad*,
64             RenewableResources*
65         >()
66     );
67 */
68 // ===== END Model ===== //
69
70
71
72 // ===== RenewableResources ===== //
73 /*
74 pybind11::class_<RenewableResources>(m, "RenewableResources")
75     .def(pybind11::init());
76     /*
77     .def(pybind11::init<>());
78     */
79 */
80 // ===== END RenewableResources ===== //
81
82 } /* PYBIND11_MODULE() */

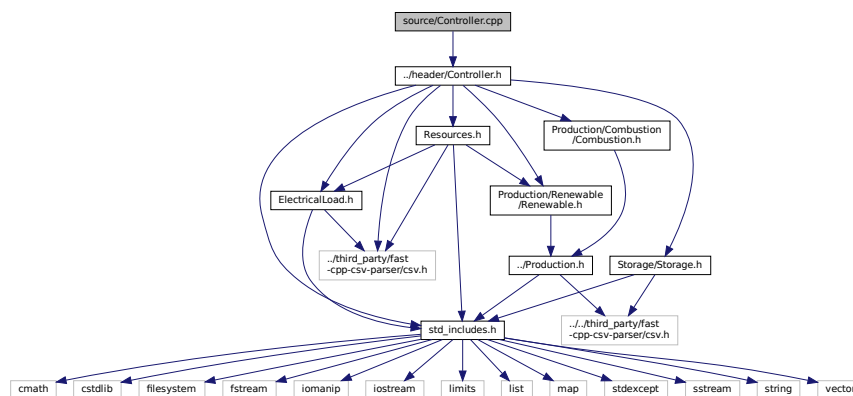
```

5.17 source/Controller.cpp File Reference

Implementation file for the [Controller](#) class.

```
#include "../header/Controller.h"
```

Include dependency graph for Controller.cpp:



5.17.1 Detailed Description

Implementation file for the [Controller](#) class.

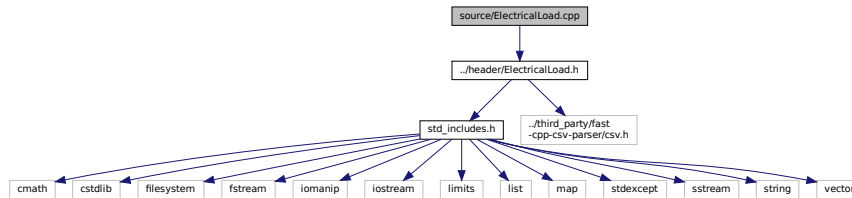
A class which contains a various dispatch control logic. Intended to serve as a component class of [Controller](#).

5.18 source/ElectricalLoad.cpp File Reference

Implementation file for the [ElectricalLoad](#) class.

```
#include "../header/ElectricalLoad.h"
```

Include dependency graph for ElectricalLoad.cpp:



5.18.1 Detailed Description

Implementation file for the [ElectricalLoad](#) class.

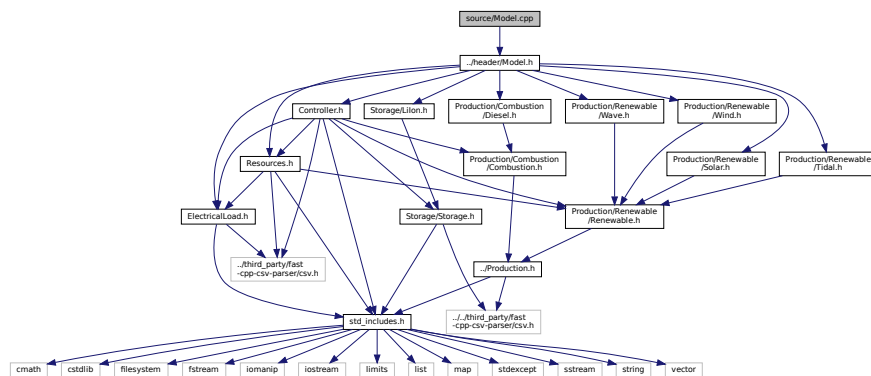
A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

5.19 source/Model.cpp File Reference

Implementation file for the [Model](#) class.

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

Include dependency graph for Model.cpp:



5.19.1 Detailed Description

Implementation file for the [Model](#) class.

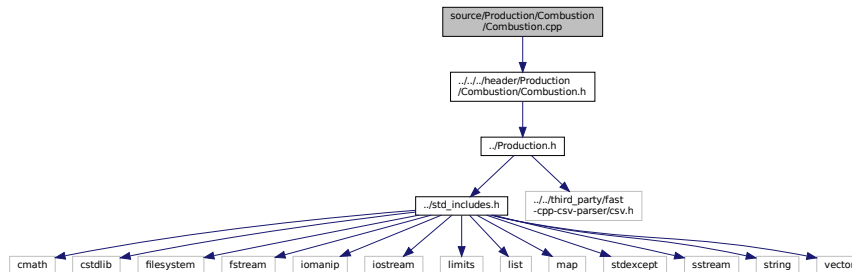
A container class which forms the centre of PGMcpp. The [Model](#) class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

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

Implementation file for the [Combustion](#) class.

```
#include "../../../../../header/Production/Combustion/Combustion.h"
```

Include dependency graph for Combustion.cpp:



5.20.1 Detailed Description

Implementation file for the [Combustion](#) class.

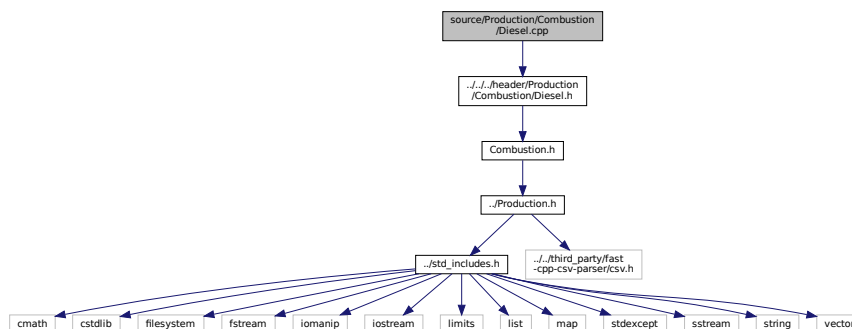
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Implementation file for the [Diesel](#) class.

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

Include dependency graph for Diesel.cpp:



5.21.1 Detailed Description

Implementation file for the [Diesel](#) class.

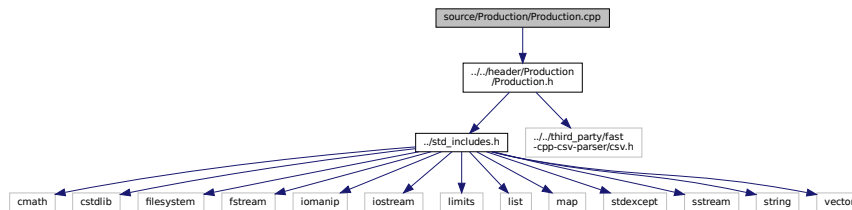
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

5.22 source/Production/Production.cpp File Reference

Implementation file for the [Production](#) class.

```
#include "../..//header/Production/Production.h"
```

Include dependency graph for Production.cpp:



5.22.1 Detailed Description

Implementation file for the [Production](#) class.

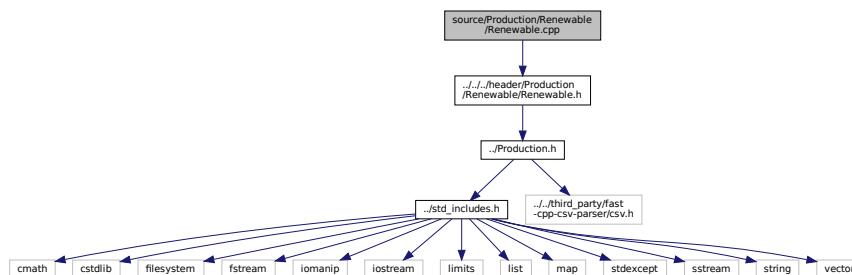
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

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

Implementation file for the [Renewable](#) class.

```
#include "../..//header/Production/Renewable/Renewable.h"
```

Include dependency graph for Renewable.cpp:



5.23.1 Detailed Description

Implementation file for the [Renewable](#) class.

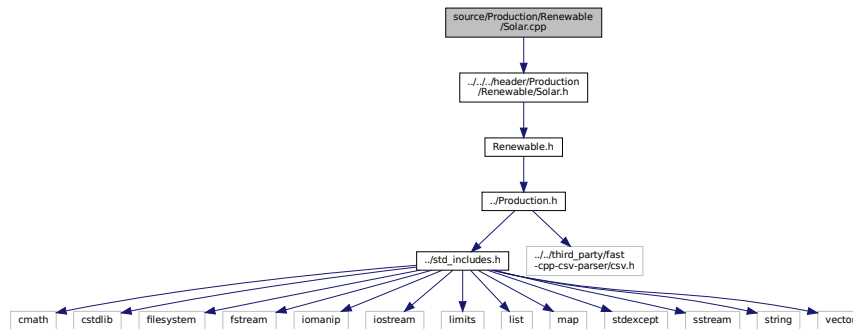
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

5.24 source/Production/Renewable/Solar.cpp File Reference

Implementation file for the [Solar](#) class.

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

Include dependency graph for Solar.cpp:



5.24.1 Detailed Description

Implementation file for the [Solar](#) class.

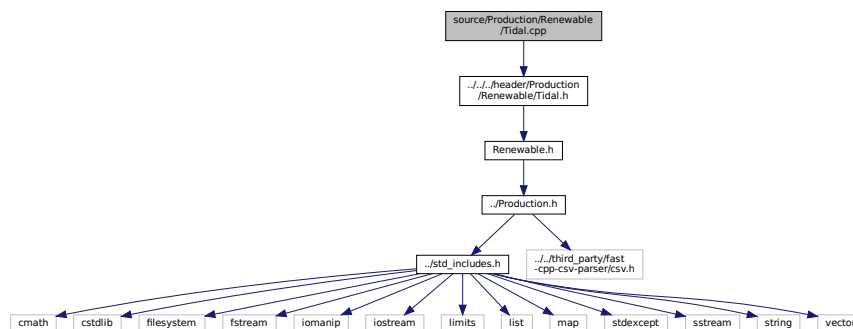
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

5.25 source/Production/Renewable/Tidal.cpp File Reference

Implementation file for the [Tidal](#) class.

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

Include dependency graph for Tidal.cpp:



5.25.1 Detailed Description

Implementation file for the [Tidal](#) class.

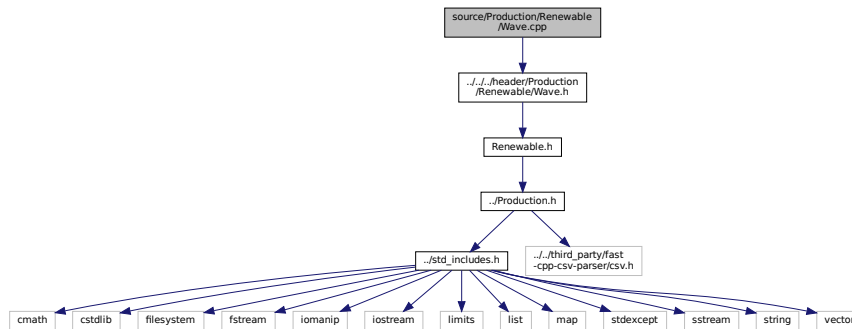
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

5.26 source/Production/Renewable/Wave.cpp File Reference

Implementation file for the [Wave](#) class.

```
#include "../.../header/Production/Renewable/Wave.h"
```

Include dependency graph for Wave.cpp:



5.26.1 Detailed Description

Implementation file for the [Wave](#) class.

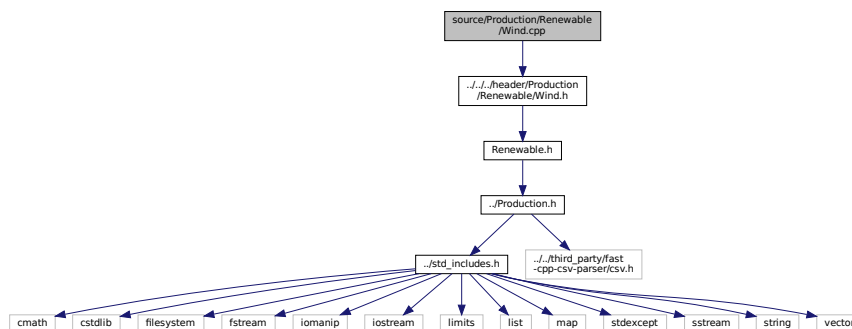
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

5.27 source/Production/Renewable/Wind.cpp File Reference

Implementation file for the [Wind](#) class.

```
#include "../.../header/Production/Renewable/Wind.h"
```

Include dependency graph for Wind.cpp:



5.27.1 Detailed Description

Implementation file for the [Wind](#) class.

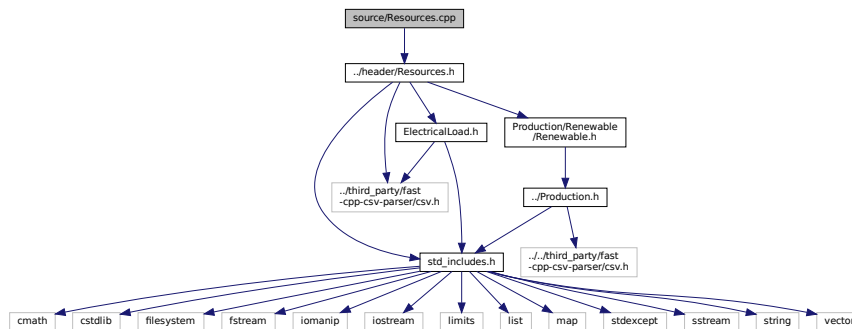
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

5.28 source/Resources.cpp File Reference

Implementation file for the [Resources](#) class.

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

Include dependency graph for Resources.cpp:



5.28.1 Detailed Description

Implementation file for the [Resources](#) class.

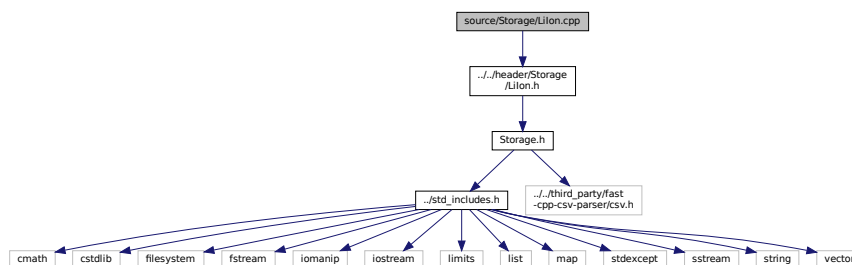
A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

5.29 source/Storage/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.

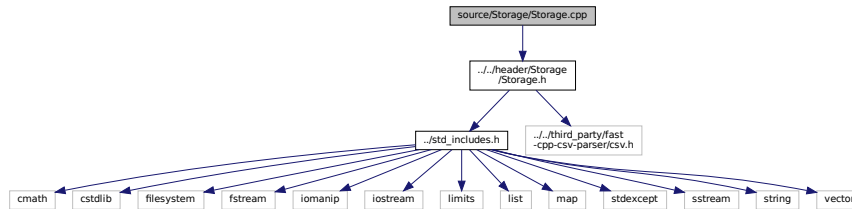
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

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.

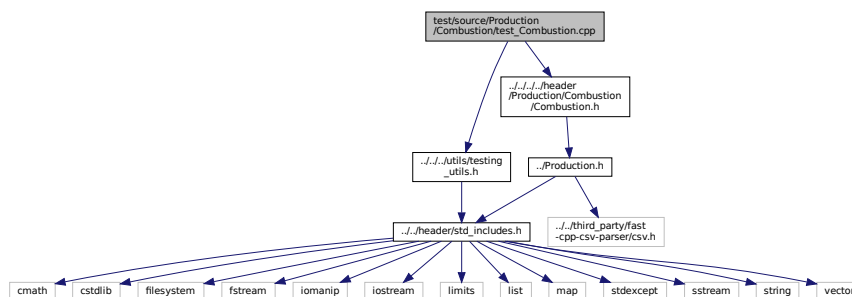
5.31 test/source/Production/Combustion/test_Combustion.cpp File Reference

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 )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Combustion");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         CombustionInputs combustion_inputs;
42
43         Combustion test_combustion(8760, 1, combustion_inputs);
44
45         // ===== END CONSTRUCTION ===== //
46
47
48
49         // ===== ATTRIBUTES ===== //
50
51         testTruth(
52             not combustion_inputs.production_inputs.print_flag,
53             __FILE__,
54             __LINE__
55 );
56
57         testFloatEquals(
58             test_combustion.fuel_consumption_vec_L.size(),
59             8760,
60             __FILE__,
61             __LINE__
62 );
63
64         testFloatEquals(
65             test_combustion.fuel_cost_vec.size(),
66             8760,
67             __FILE__,
68             __LINE__
69 );
70
71         testFloatEquals(
72             test_combustion.CO2_emissions_vec_kg.size(),
73             8760,
74             __FILE__,
75             __LINE__
76 );
77
78         testFloatEquals(
79             test_combustion.CO_emissions_vec_kg.size(),
80             8760,
81             __FILE__,
82             __LINE__
83 );
84
85         testFloatEquals(
```

```

86     test_combustion.NOx_emissions_vec_kg.size(),
87     8760,
88     __FILE__,
89     __LINE__
90 );
91
92 testFloatEquals(
93     test_combustion.SOx_emissions_vec_kg.size(),
94     8760,
95     __FILE__,
96     __LINE__
97 );
98
99 testFloatEquals(
100     test_combustion.CH4_emissions_vec_kg.size(),
101     8760,
102     __FILE__,
103     __LINE__
104 );
105
106 testFloatEquals(
107     test_combustion.PM_emissions_vec_kg.size(),
108     8760,
109     __FILE__,
110     __LINE__
111 );
112
113 // ===== END ATTRIBUTES ===== //
114
115 }    /* try */
116
117
118 catch (...) {
119     //...
120
121     printGold(" ..... ");
122     printRed("FAIL");
123     std::cout << std::endl;
124     throw;
125 }
126
127
128 printGold(" ..... ");
129 printGreen("PASS");
130 std::cout << std::endl;
131 return 0;
132
133 }    /* main() */

```

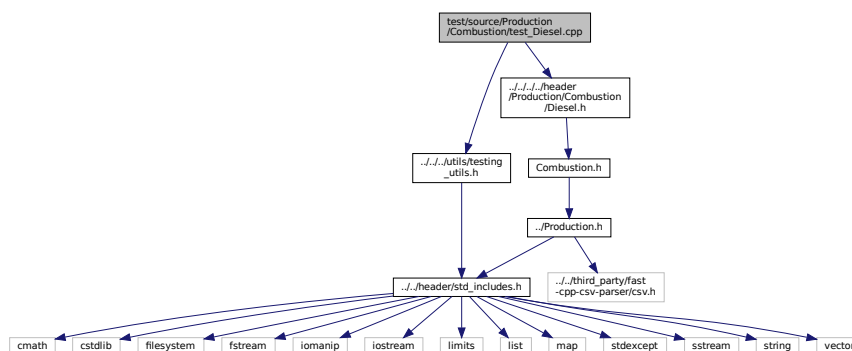
5.32 test/source/Production/Combustion/test_Diesel.cpp File Reference

Testing suite for [Diesel](#) class.

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

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

Include dependency graph for test_Diesel.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.32.1 Detailed Description

Testing suite for [Diesel](#) class.

A suite of tests for the [Diesel](#) class.

5.32.2 Function Documentation

5.32.2.1 main()

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

```

78     CombustionType :: DIESEL,
79     __FILE__,
80     __LINE__
81 );
82
83 testTruth(
84     test_diesel_ptr->type_str == "DIESEL",
85     __FILE__,
86     __LINE__
87 );
88
89 testFloatEquals(
90     test_diesel_ptr->linear_fuel_slope_LkWh,
91     0.265675,
92     __FILE__,
93     __LINE__
94 );
95
96 testFloatEquals(
97     test_diesel_ptr->linear_fuel_intercept_LkWh,
98     0.026676,
99     __FILE__,
100    __LINE__
101 );
102
103 testFloatEquals(
104     test_diesel_ptr->capital_cost,
105     94125.375446,
106     __FILE__,
107     __LINE__
108 );
109
110 testFloatEquals(
111     test_diesel_ptr->operation_maintenance_cost_kWh,
112     0.069905,
113     __FILE__,
114     __LINE__
115 );
116
117 testFloatEquals(
118     ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
119     0.2,
120     __FILE__,
121     __LINE__
122 );
123
124 testFloatEquals(
125     ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
126     4,
127     __FILE__,
128     __LINE__
129 );
130
131 testFloatEquals(
132     test_diesel_ptr->replace_running_hrs,
133     30000,
134     __FILE__,
135     __LINE__
136 );
137
138 // ===== END ATTRIBUTES ===== //
139
140
141
142 // ===== METHODS ===== //
143
144 // test capacity constraint
145 testFloatEquals(
146     test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
147     test_diesel_ptr->capacity_kW,
148     __FILE__,
149     __LINE__
150 );
151
152 // test minimum load ratio constraint
153 testFloatEquals(
154     test_diesel_ptr->requestProductionkW(
155         0,
156         1,
157         0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
158             test_diesel_ptr->capacity_kW
159     ),
160     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
161     __FILE__,
162     __LINE__
163 );
164

```



```

165 // test commit()
166 std::vector<double> dt_vec_hrs (48, 1);
167
168 std::vector<double> load_vec_kW = {
169     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
170     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
171     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
172     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
173 };
174
175 std::vector<bool> expected_is_running_vec = {
176     1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
177     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
178     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
179     1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
180 };
181
182 double load_kW = 0;
183 double production_kW = 0;
184 double roll = 0;
185
186 for (int i = 0; i < 48; i++) {
187     roll = (double)rand() / RAND_MAX;
188
189     if (roll >= 0.95) {
190         roll = 1.25;
191     }
192
193     load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
194     load_kW = load_vec_kW[i];
195
196     production_kW = test_diesel_ptr->requestProductionkW(
197         i,
198         dt_vec_hrs[i],
199         load_kW
200     );
201
202     load_kW = test_diesel_ptr->commit(
203         i,
204         dt_vec_hrs[i],
205         production_kW,
206         load_kW
207     );
208
209     // load_kW <= load_vec_kW (i.e., after vs before)
210     testLessThanOrEqualTo(
211         load_kW,
212         load_vec_kW[i],
213         __FILE__,
214         __LINE__
215     );
216
217     // production = dispatch + storage + curtailment
218     testFloatEquals(
219         test_diesel_ptr->production_vec_kW[i] -
220         test_diesel_ptr->dispatch_vec_kW[i] -
221         test_diesel_ptr->storage_vec_kW[i] -
222         test_diesel_ptr->curtailment_vec_kW[i],
223         0,
224         __FILE__,
225         __LINE__
226     );
227
228     // capacity constraint
229     if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
230         testFloatEquals(
231             test_diesel_ptr->production_vec_kW[i],
232             test_diesel_ptr->capacity_kW,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // minimum load ratio constraint
239     else if (
240         test_diesel_ptr->is_running and
241         test_diesel_ptr->production_vec_kW[i] > 0 and
242         load_vec_kW[i] <
243         ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
244     ) {
245         testFloatEquals(
246             test_diesel_ptr->production_vec_kW[i],
247             ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
248             test_diesel_ptr->capacity_kW,
249             __FILE__,
250             __LINE__
251         );

```

```

252     }
253
254     // minimum runtime constraint
255     testFloatEquals(
256         test_diesel_ptr->is_running_vec[i],
257         expected_is_running_vec[i],
258         __FILE__,
259         __LINE__
260     );
261
262     // O&M, fuel consumption, and emissions > 0 whenever diesel is running
263     if (test_diesel_ptr->is_running) {
264         testGreaterThan(
265             test_diesel_ptr->operation_maintenance_cost_vec[i],
266             0,
267             __FILE__,
268             __LINE__
269         );
270
271         testGreaterThan(
272             test_diesel_ptr->fuel_consumption_vec_L[i],
273             0,
274             __FILE__,
275             __LINE__
276         );
277
278         testGreaterThan(
279             test_diesel_ptr->fuel_cost_vec[i],
280             0,
281             __FILE__,
282             __LINE__
283         );
284
285         testGreaterThan(
286             test_diesel_ptr->CO2_emissions_vec_kg[i],
287             0,
288             __FILE__,
289             __LINE__
290         );
291
292         testGreaterThan(
293             test_diesel_ptr->CO_emissions_vec_kg[i],
294             0,
295             __FILE__,
296             __LINE__
297         );
298
299         testGreaterThan(
300             test_diesel_ptr->NOx_emissions_vec_kg[i],
301             0,
302             __FILE__,
303             __LINE__
304         );
305
306         testGreaterThan(
307             test_diesel_ptr->SOx_emissions_vec_kg[i],
308             0,
309             __FILE__,
310             __LINE__
311         );
312
313         testGreaterThan(
314             test_diesel_ptr->CH4_emissions_vec_kg[i],
315             0,
316             __FILE__,
317             __LINE__
318         );
319
320         testGreaterThan(
321             test_diesel_ptr->PM_emissions_vec_kg[i],
322             0,
323             __FILE__,
324             __LINE__
325         );
326     }
327
328     // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
329     else {
330         testFloatEquals(
331             test_diesel_ptr->operation_maintenance_cost_vec[i],
332             0,
333             __FILE__,
334             __LINE__
335         );
336
337         testFloatEquals(
338             test_diesel_ptr->fuel_consumption_vec_L[i],

```

```

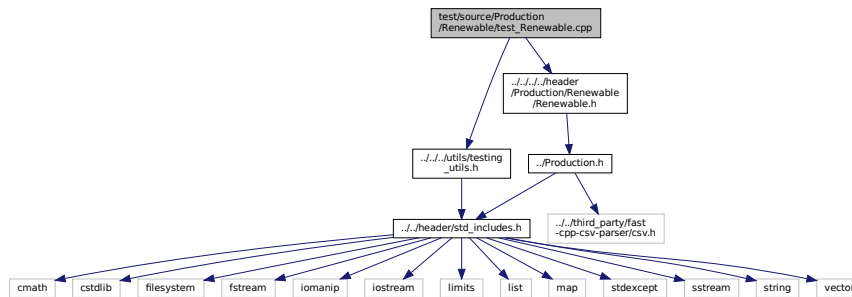
339         0,
340         __FILE__,
341         __LINE__
342     );
343
344     testFloatEquals(
345         test_diesel_ptr->fuel_cost_vec[i],
346         0,
347         __FILE__,
348         __LINE__
349     );
350
351     testFloatEquals(
352         test_diesel_ptr->CO2_emissions_vec_kg[i],
353         0,
354         __FILE__,
355         __LINE__
356     );
357
358     testFloatEquals(
359         test_diesel_ptr->CO_emissions_vec_kg[i],
360         0,
361         __FILE__,
362         __LINE__
363     );
364
365     testFloatEquals(
366         test_diesel_ptr->NOx_emissions_vec_kg[i],
367         0,
368         __FILE__,
369         __LINE__
370     );
371
372     testFloatEquals(
373         test_diesel_ptr->SOx_emissions_vec_kg[i],
374         0,
375         __FILE__,
376         __LINE__
377     );
378
379     testFloatEquals(
380         test_diesel_ptr->CH4_emissions_vec_kg[i],
381         0,
382         __FILE__,
383         __LINE__
384     );
385
386     testFloatEquals(
387         test_diesel_ptr->PM_emissions_vec_kg[i],
388         0,
389         __FILE__,
390         __LINE__
391     );
392 }
393 }
394
395 // ===== END METHODS ===== //
396
397 } /* try */
398
399
400 catch (...) {
401     delete test_diesel_ptr;
402
403     printGold(" ... ");
404     printRed("FAIL");
405     std::cout << std::endl;
406     throw;
407 }
408
409
410 delete test_diesel_ptr;
411
412 printGold(" ... ");
413 printGreen("PASS");
414 std::cout << std::endl;
415 return 0;
416
417 } /* main() */

```

5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference

Testing suite for [Renewable](#) class.

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



Functions

- int [main](#) (int argc, char **argv)

5.33.1 Detailed Description

Testing suite for [Renewable](#) class.

A suite of tests for the [Renewable](#) class.

5.33.2 Function Documentation

5.33.2.1 main()

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

```

40
41 RenewableInputs renewable_inputs;
42
43 Renewable test_renewable(8760, 1, renewable_inputs);
44
45 // ===== END CONSTRUCTION ===== //
46
47
48
49 // ===== ATTRIBUTES ===== //
50
51 testTruth(
52     not renewable_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56
57 // ===== END ATTRIBUTES ===== //
58
59 } /* try */
60
61
62 catch (...) {
63     //...
64
65     printGold(" ..... ");
66     printRed("FAIL");
67     std::cout << std::endl;
68     throw;
69 }
70
71
72 printGold(" ..... ");
73 printGreen("PASS");
74 std::cout << std::endl;
75 return 0;
76 } /* main() */

```

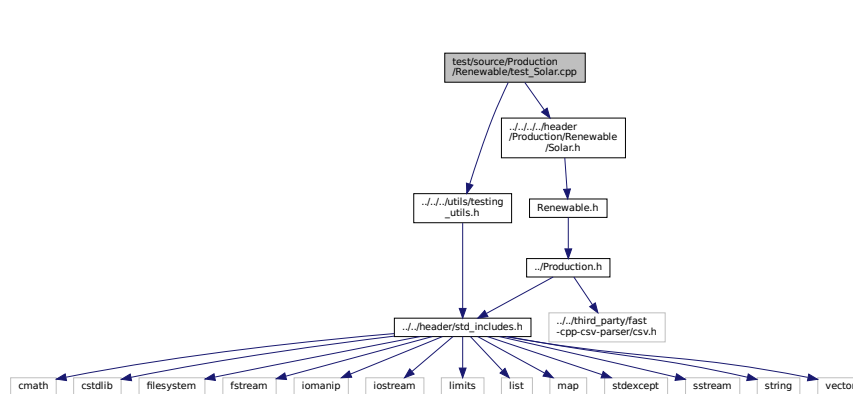
5.34 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for [Solar](#) class.

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

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

Include dependency graph for test_Solar.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.34.1 Detailed Description

Testing suite for [Solar](#) class.

A suite of tests for the [Solar](#) class.

5.34.2 Function Documentation

5.34.2.1 main()

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

    printGold("\tTesting Production <-- Renewable <-- Solar");
    srand(time(NULL));

    Renewable* test_solar_ptr;

    try {
        // ===== CONSTRUCTION ===== //
        bool error_flag = true;
        try {
            SolarInputs bad_solar_inputs;
            bad_solar_inputs.derating = -1;
            Solar bad_solar(8760, 1, bad_solar_inputs);
            error_flag = false;
        } catch (...) {
            // Task failed successfully! =P
        }
        if (not error_flag) {
            expectedErrorNotDetected(__FILE__, __LINE__);
        }
        SolarInputs solar_inputs;
        test_solar_ptr = new Solar(8760, 1, solar_inputs);
        // ===== END CONSTRUCTION ===== //

        // ===== ATTRIBUTES ===== //
        testTruth(
            not solar_inputs.renewable_inputs.production_inputs.print_flag,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_solar_ptr->type,
            RenewableType :: SOLAR,
            __FILE__,
            __LINE__
        );
        testTruth(
            test_solar_ptr->type_str == "SOLAR",
            __FILE__,
            __LINE__
        );
    }
```

```

86
87 testFloatEquals(
88     test_solar_ptr->capital_cost,
89     350118.723363,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_solar_ptr->operation_maintenance_cost_kWh,
96     0.01,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_solar_ptr->computeProductionkW(0, 1, 2),
110     100,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_solar_ptr->computeProductionkW(0, 1, -1),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double solar_resource_kWm2 = 0;
136
137 for (int i = 0; i < 48; i++) {
138     roll = (double)rand() / RAND_MAX;
139
140     solar_resource_kWm2 = roll;
141
142     roll = (double)rand() / RAND_MAX;
143
144     if (roll <= 0.1) {
145         solar_resource_kWm2 = 0;
146     }
147
148     else if (roll >= 0.95) {
149         solar_resource_kWm2 = 1.25;
150     }
151
152     roll = (double)rand() / RAND_MAX;
153
154     if (roll >= 0.95) {
155         roll = 1.25;
156     }
157
158     load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
159     load_kW = load_vec_kW[i];
160
161     production_kW = test_solar_ptr->computeProductionkW(
162         i,
163         dt_vec_hrs[i],
164         solar_resource_kWm2
165     );
166
167     load_kW = test_solar_ptr->commit(
168         i,
169         dt_vec_hrs[i],
170         production_kW,
171         load_kW
172     );

```

```

173
174 // is running (or not) as expected
175 if (solar_resource_kWm2 > 0) {
176     testTruth(
177         test_solar_ptr->is_running,
178         __FILE__,
179         __LINE__
180     );
181 }
182
183 else {
184     testTruth(
185         not test_solar_ptr->is_running,
186         __FILE__,
187         __LINE__
188     );
189 }
190
191 // load_kW <= load_vec_kW (i.e., after vs before)
192 testLessThanOrEqualTo(
193     load_kW,
194     load_vec_kW[i],
195     __FILE__,
196     __LINE__
197 );
198
199 // production = dispatch + storage + curtailment
200 testFloatEquals(
201     test_solar_ptr->production_vec_kW[i] -
202     test_solar_ptr->dispatch_vec_kW[i] -
203     test_solar_ptr->storage_vec_kW[i] -
204     test_solar_ptr->curtailment_vec_kW[i],
205     0,
206     __FILE__,
207     __LINE__
208 );
209
210 // capacity constraint
211 if (solar_resource_kWm2 > 1) {
212     testFloatEquals(
213         test_solar_ptr->production_vec_kW[i],
214         test_solar_ptr->capacity_kW,
215         __FILE__,
216         __LINE__
217     );
218 }
219
220 // resource, O&M > 0 whenever solar is running (i.e., producing)
221 if (test_solar_ptr->is_running) {
222     testGreaterThan(
223         solar_resource_kWm2,
224         0,
225         __FILE__,
226         __LINE__
227     );
228
229     testGreaterThan(
230         test_solar_ptr->operation_maintenance_cost_vec[i],
231         0,
232         __FILE__,
233         __LINE__
234     );
235 }
236
237 // resource, O&M = 0 whenever solar is not running (i.e., not producing)
238 else {
239     testFloatEquals(
240         solar_resource_kWm2,
241         0,
242         __FILE__,
243         __LINE__
244     );
245
246     testFloatEquals(
247         test_solar_ptr->operation_maintenance_cost_vec[i],
248         0,
249         __FILE__,
250         __LINE__
251     );
252 }
253 }
254
255
256 // ===== END METHODS ===== //
257
258 } /* try */
259

```



```

260
261 catch (...) {
262     delete test_solar_ptr;
263
264     printGold(" ..... ");
265     printRed("FAIL");
266     std::cout << std::endl;
267     throw;
268 }
269
270
271 delete test_solar_ptr;
272
273 printGold(" ..... ");
274 printGreen("PASS");
275 std::cout << std::endl;
276 return 0;
277 } /* main() */

```

5.35 test/source/Production/Renewable/test_Tidal.cpp File Reference

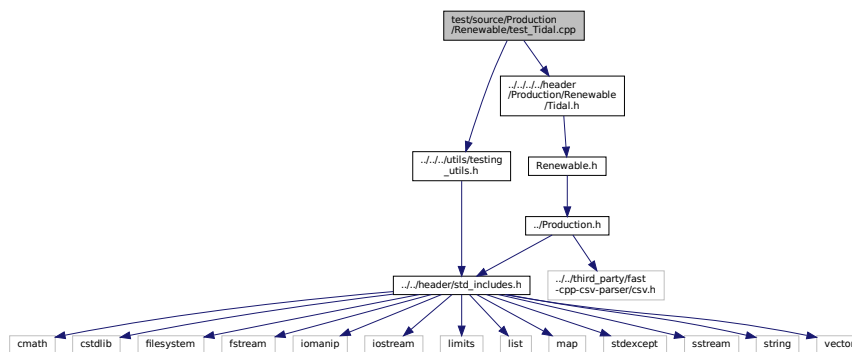
Testing suite for [Tidal](#) class.

```

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

```

Include dependency graph for test_Tidal.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.35.1 Detailed Description

Testing suite for [Tidal](#) class.

A suite of tests for the [Tidal](#) class.

5.35.2 Function Documentation

5.35.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Tidal");
33
34     srand(time(NULL));
35
36     Renewable* test_tidal_ptr;
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         bool error_flag = true;
43
44         try {
45             TidalInputs bad_tidal_inputs;
46             bad_tidal_inputs.design_speed_ms = -1;
47
48             Tidal bad_tidal(8760, 1, bad_tidal_inputs);
49
50             error_flag = false;
51         } catch (...) {
52             // Task failed successfully! =P
53         }
54         if (not error_flag) {
55             expectedErrorNotDetected(__FILE__, __LINE__);
56         }
57
58         TidalInputs tidal_inputs;
59
60         test_tidal_ptr = new Tidal(8760, 1, tidal_inputs);
61
62         // ===== END CONSTRUCTION ===== //
63
64
65
66         // ===== ATTRIBUTES ===== //
67
68         testTruth(
69             not tidal_inputs.renewable_inputs.production_inputs.print_flag,
70             __FILE__,
71             __LINE__
72 );
73
74         testFloatEquals(
75             test_tidal_ptr->type,
76             RenewableType :: TIDAL,
77             __FILE__,
78             __LINE__
79 );
80
81         testTruth(
82             test_tidal_ptr->type_str == "TIDAL",
83             __FILE__,
84             __LINE__
85 );
86
87         testFloatEquals(
88             test_tidal_ptr->capital_cost,
89             500237.446725,
90             __FILE__,
91             __LINE__
92 );
93
94         testFloatEquals(
95             test_tidal_ptr->operation_maintenance_cost_kWh,
96             0.069905,
97             __FILE__,
98             __LINE__
99 );
100
101         // ===== END ATTRIBUTES ===== //
102
103
104
105         // ===== METHODS ===== //
106

```

```

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

```

```

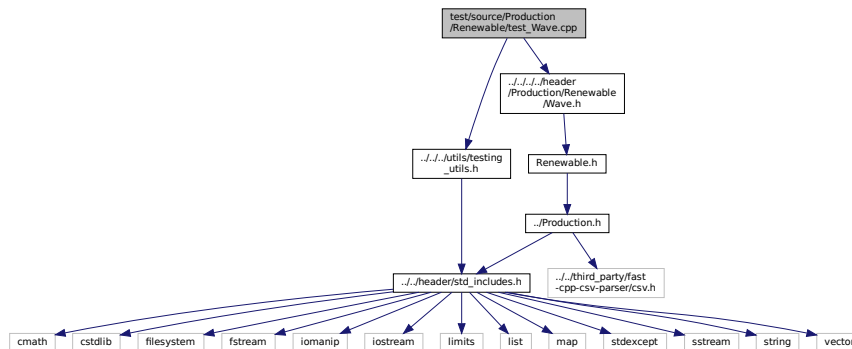
194     else {
195         testTruth(
196             not test_tidal_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],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

```

5.36 test/source/Production/Renewable/test_Wave.cpp File Reference

Testing suite for [Wave](#) class.

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



Functions

- int [main](#) (int argc, char **argv)

5.36.1 Detailed Description

Testing suite for [Wave](#) class.

A suite of tests for the [Wave](#) class.

5.36.2 Function Documentation

5.36.2.1 main()

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

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

    srand(time(NULL));

    Renewable* test_wave_ptr;

    try {
        // ===== CONSTRUCTION =====
        bool error_flag = true;

        try {
            WaveInputs bad_wave_inputs;
            bad_wave_inputs.design_significant_wave_height_m = -1;
        }
    }
}
```

```

47
48     Wave bad_wave(8760, 1, bad_wave_inputs);
49
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57
58 WaveInputs wave_inputs;
59
60 test_wave_ptr = new Wave(8760, 1, wave_inputs);
61
62 // ===== END CONSTRUCTION ===== //
63
64
65
66 // ===== ATTRIBUTES ===== //
67
68 testTruth(
69     not wave_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73
74 testFloatEquals(
75     test_wave_ptr->type,
76     RenewableType :: WAVE,
77     __FILE__,
78     __LINE__
79 );
80
81 testTruth(
82     test_wave_ptr->type_str == "WAVE",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_wave_ptr->capital_cost,
89     850831.063539,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wave_ptr->operation_maintenance_cost_kWh,
96     0.069905,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;

```

```

134 double roll = 0;
135 double significant_wave_height_m = 0;
136 double energy_period_s = 0;
137
138 for (int i = 0; i < 48; i++) {
139     roll = (double)rand() / RAND_MAX;
140
141     if (roll <= 0.05) {
142         roll = 0;
143     }
144
145     significant_wave_height_m = roll *
146         ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
147
148     roll = (double)rand() / RAND_MAX;
149
150     if (roll <= 0.05) {
151         roll = 0;
152     }
153
154     energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
155
156     roll = (double)rand() / RAND_MAX;
157
158     if (roll >= 0.95) {
159         roll = 1.25;
160     }
161
162     load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
163     load_kW = load_vec_kW[i];
164
165     production_kW = test_wave_ptr->computeProductionkW(
166         i,
167         dt_vec_hrs[i],
168         significant_wave_height_m,
169         energy_period_s
170     );
171
172     load_kW = test_wave_ptr->commit(
173         i,
174         dt_vec_hrs[i],
175         production_kW,
176         load_kW
177     );
178
179     // is running (or not) as expected
180     if (production_kW > 0) {
181         testTruth(
182             test_wave_ptr->is_running,
183             __FILE__,
184             __LINE__
185         );
186     }
187
188     else {
189         testTruth(
190             not test_wave_ptr->is_running,
191             __FILE__,
192             __LINE__
193         );
194     }
195
196     // load_kW <= load_vec_kW (i.e., after vs before)
197     testLessThanOrEqualTo(
198         load_kW,
199         load_vec_kW[i],
200         __FILE__,
201         __LINE__
202     );
203
204     // production = dispatch + storage + curtailment
205     testFloatEquals(
206         test_wave_ptr->production_vec_kW[i] -
207         test_wave_ptr->dispatch_vec_kW[i] -
208         test_wave_ptr->storage_vec_kW[i] -
209         test_wave_ptr->curtailment_vec_kW[i],
210         0,
211         __FILE__,
212         __LINE__
213     );
214
215     // resource, O&M > 0 whenever wave is running (i.e., producing)
216     if (test_wave_ptr->is_running) {
217         testGreaterThan(
218             significant_wave_height_m,
219             0,
220             __FILE__,

```

```

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

```

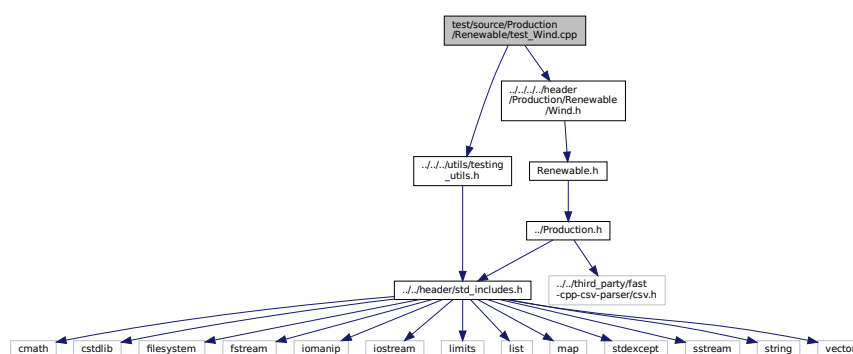
5.37 test/source/Production/Renewable/test_Wind.cpp File Reference

Testing suite for [Wind](#) class.

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

```
#include "../.../header/Production/Renewable/Wind.h"
```

Include dependency graph for test_Wind.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.37.1 Detailed Description

Testing suite for [Wind](#) class.

A suite of tests for the [Wind](#) class.

5.37.2 Function Documentation

5.37.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wind");
33
34     srand(time(NULL));
35
36     Renewable* test_wind_ptr;
37
38     try {
39
40 // ===== CONSTRUCTION ===== //
41
42 bool error_flag = true;
43
44     try {
45         WindInputs bad_wind_inputs;
46         bad_wind_inputs.design_speed_ms = -1;
47
48         Wind bad_wind(8760, 1, bad_wind_inputs);
49
50         error_flag = false;
51     } catch (...) {
52         // Task failed successfully! =P
53     }
54     if (not error_flag) {
55         expectedErrorNotDetected(__FILE__, __LINE__);
56     }
57
58     WindInputs wind_inputs;
59
60     test_wind_ptr = new Wind(8760, 1, wind_inputs);
61
62 // ===== END CONSTRUCTION ===== //
63
64
65
66 // ===== ATTRIBUTES ===== //
67
68     testTruth(
69         not wind_inputs.renewable_inputs.production_inputs.print_flag,
70         __FILE__,
71         __LINE__
72 );
73
74     testFloatEquals(
75         test_wind_ptr->type,
76         RenewableType :: WIND,
77         __FILE__,
```

```

78     __LINE__
79 );
80
81 testTruth(
82     test_wind_ptr->type_str == "WIND",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_wind_ptr->capital_cost,
89     450356.170088,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wind_ptr->operation_maintenance_cost_kWh,
96     0.034953,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wind_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wind_ptr->computeProductionkW(
117         0,
118         1,
119         ((Wind*)test_wind_ptr)->design_speed_ms
120     ),
121     test_wind_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_wind_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()
134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double wind_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         wind_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164

```

```

165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_wind_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         wind_resource_ms
176     );
177
178     load_kW = test_wind_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_wind_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193
194     else {
195         testTruth(
196             not test_wind_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_wind_ptr->production_vec_kW[i] -
213         test_wind_ptr->dispatch_vec_kW[i] -
214         test_wind_ptr->storage_vec_kW[i] -
215         test_wind_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220
221     // resource, O&M > 0 whenever wind is running (i.e., producing)
222     if (test_wind_ptr->is_running) {
223         testGreaterThan(
224             wind_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229
230         testGreaterThan(
231             test_wind_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // O&M = 0 whenever wind is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_wind_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248
249
250 // ===== END METHODS ===== //
251

```

```

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

```

5.38 test/source/Production/test_Production.cpp File Reference

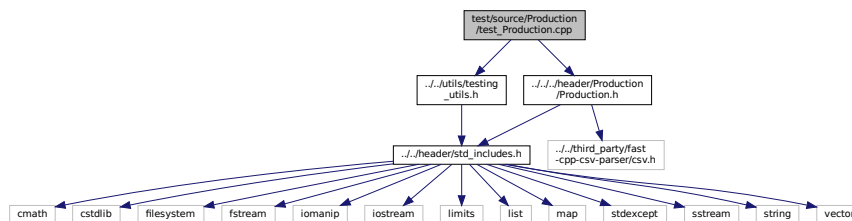
Testing suite for [Production](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Production/Production.h"

```

Include dependency graph for test_Production.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.38.1 Detailed Description

Testing suite for [Production](#) class.

A suite of tests for the [Production](#) class.

5.38.2 Function Documentation

5.38.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\n\tTesting Production");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         bool error_flag = true;
42
43         try {
44             ProductionInputs production_inputs;
45
46             Production bad_production(0, 1, production_inputs);
47
48             error_flag = false;
49         } catch (...) {
50             // Task failed successfully! =P
51         }
52         if (not error_flag) {
53             expectedErrorNotDetected(__FILE__, __LINE__);
54         }
55
56         ProductionInputs production_inputs;
57
58         Production test_production(8760, 1, production_inputs);
59
60         // ===== END CONSTRUCTION ===== //
61
62
63
64         // ===== ATTRIBUTES ===== //
65
66         testTruth(
67             not production_inputs.print_flag,
68             __FILE__,
69             __LINE__
70 );
71
72         testFloatEquals(
73             production_inputs.nominal_inflation_annual,
74             0.02,
75             __FILE__,
76             __LINE__
77 );
78
79         testFloatEquals(
80             production_inputs.nominal_discount_annual,
81             0.04,
82             __FILE__,
83             __LINE__
84 );
85
86         testFloatEquals(
87             test_production.n_points,
88             8760,
89             __FILE__,
90             __LINE__
91 );
92
93         testFloatEquals(
94             test_production.capacity_kW,
95             100,
96             __FILE__,
97             __LINE__
98 );
99
100         testFloatEquals(
101             test_production.real_discount_annual,
102             0.0196078431372549,
103             __FILE__,
104             __LINE__
105 );
106

```

```

107 testFloatEquals(
108     test_production.production_vec_kW.size(),
109     8760,
110     __FILE__,
111     __LINE__
112 );
113
114 testFloatEquals(
115     test_production.dispatch_vec_kW.size(),
116     8760,
117     __FILE__,
118     __LINE__
119 );
120
121 testFloatEquals(
122     test_production.storage_vec_kW.size(),
123     8760,
124     __FILE__,
125     __LINE__
126 );
127
128 testFloatEquals(
129     test_production.curtailment_vec_kW.size(),
130     8760,
131     __FILE__,
132     __LINE__
133 );
134
135 testFloatEquals(
136     test_production.capital_cost_vec.size(),
137     8760,
138     __FILE__,
139     __LINE__
140 );
141
142 testFloatEquals(
143     test_production.operation_maintenance_cost_vec.size(),
144     8760,
145     __FILE__,
146     __LINE__
147 );
148
149 // ===== END ATTRIBUTES ===== //
150
151 } /* try */
152
153 catch (...) {
154     //...
155
156     printGold(" ..... ");
157     printRed("FAIL");
158     std::cout << std::endl;
159     throw;
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

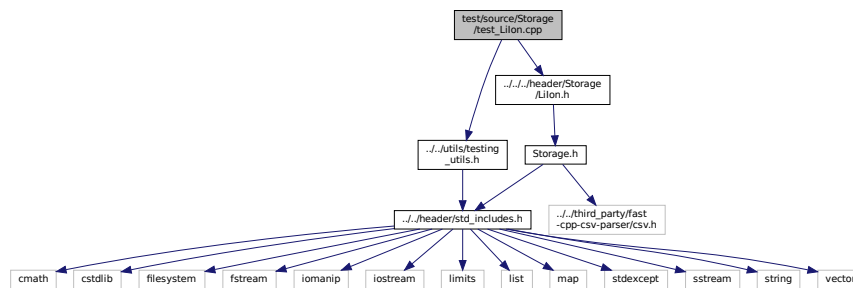
Testing suite for [Lilon](#) class.

```

#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()

```

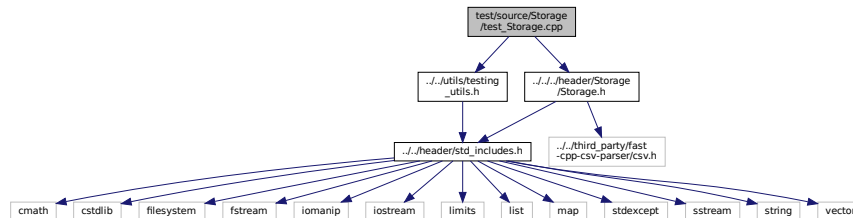
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Storage <-- LiIon");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38         //...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

```

5.40 test/source/Storage/test_Storage.cpp File Reference

Testing suite for [Storage](#) class.

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



Functions

- [int main](#) (int argc, char **argv)

5.40.1 Detailed Description

Testing suite for [Storage](#) class.

A suite of tests for the [Storage](#) class.

5.40.2 Function Documentation

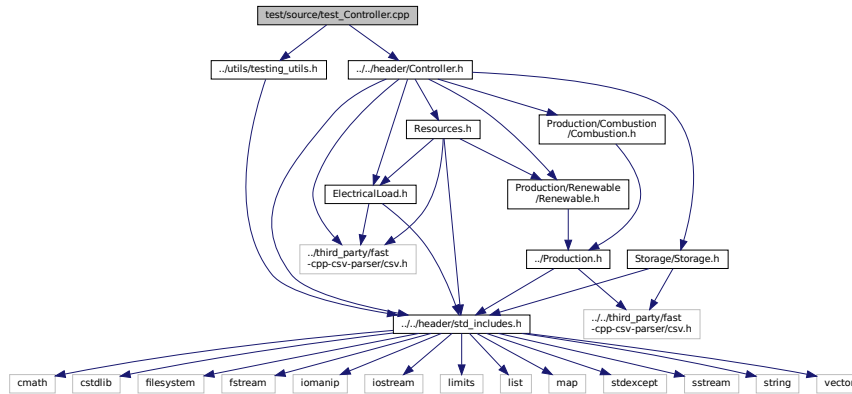
5.40.2.1 main()

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


5.41 test/source/test_Controller.cpp File Reference

Testing suite for [Controller](#) class.

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



Functions

- int [main](#) (int argc, char **argv)

5.41.1 Detailed Description

Testing suite for [Controller](#) class.

A suite of tests for the [Controller](#) class.

5.41.2 Function Documentation

5.41.2.1 main()

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

```

38
39 // ===== CONSTRUCTION ===== //
40
41 Controller test_controller;
42
43 // ===== END CONSTRUCTION =====//
44
45
46
47 // ===== ATTRIBUTES ===== //
48
49 //...
50
51 // ===== END ATTRIBUTES ===== //
52
53
54
55 // ===== METHODS ===== //
56
57 //...
58
59 // ===== END METHODS ===== //
60
61 } /* try */
62
63
64 catch (...) {
65     //...
66
67     printGold(" ..... ");
68     printRed("FAIL");
69     std::cout << std::endl;
70     throw;
71 }
72
73
74 printGold(" ..... ");
75 printGreen("PASS");
76 std::cout << std::endl;
77 return 0;
78 } /* main() */

```

5.42 test/source/test_ElectricalLoad.cpp File Reference

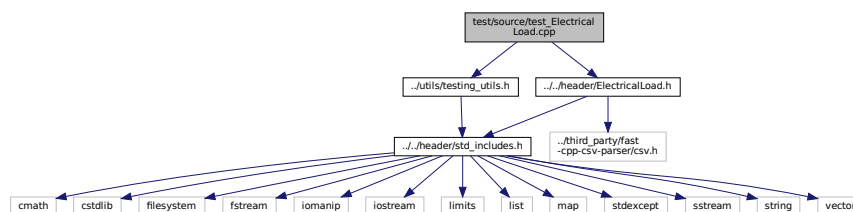
Testing suite for [ElectricalLoad](#) class.

```

#include "../utils/testing_utils.h"
#include "../..header/ElectricalLoad.h"

```

Include dependency graph for test_ElectricalLoad.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.42.1 Detailed Description

Testing suite for [ElectricalLoad](#) class.

A suite of tests for the [ElectricalLoad](#) class.

5.42.2 Function Documentation

5.42.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting ElectricalLoad");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         std::string path_2_electrical_load_time_series =
42             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
43
44         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
45
46         // ===== END CONSTRUCTION ===== //
47
48
49
50         // ===== ATTRIBUTES ===== //
51
52         testTruth(
53             test_electrical_load.path_2_electrical_load_time_series ==
54             path_2_electrical_load_time_series,
55             __FILE__,
56             __LINE__
57         );
58
59         testFloatEquals(
60             test_electrical_load.n_points,
61             8760,
62             __FILE__,
63             __LINE__
64         );
65
66         testFloatEquals(
67             test_electrical_load.n_years,
68             0.999886,
69             __FILE__,
70             __LINE__
71         );
72
73         testFloatEquals(
74             test_electrical_load.min_load_kW,
75             82.1211213927802,
76             __FILE__,
77             __LINE__
78         );
79
80         testFloatEquals(
81             test_electrical_load.mean_load_kW,
82             258.373472633202,
83             __FILE__,
84             __LINE__
85         );
86
87
88         testFloatEquals(
89             test_electrical_load.max_load_kW,
90             500,
91             __FILE__,
92             __LINE__
93         );
94
95
96         std::vector<double> expected_dt_vec_hrs (48, 1);
97

```

```

98 std::vector<double> expected_time_vec_hrs = {
99     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
100    12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
101    24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
102    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
103 };
104
105 std::vector<double> expected_load_vec_kW = {
106    360.253836463674,
107    355.171277826775,
108    353.776453532298,
109    353.75405737934,
110    346.592867404975,
111    340.132411175118,
112    337.354867340578,
113    340.644115618736,
114    363.639028500678,
115    378.787797779238,
116    372.215798201712,
117    395.093925731298,
118    402.325427142659,
119    386.907725462306,
120    380.709170928091,
121    372.062070914977,
122    372.328646856954,
123    391.841444284136,
124    394.029351759596,
125    383.369407765254,
126    381.093099675206,
127    382.604158946193,
128    390.744843709034,
129    383.13949492437,
130    368.150393976985,
131    364.629744480226,
132    363.572736804082,
133    359.854924202248,
134    355.207590170267,
135    349.094656012401,
136    354.365935871597,
137    343.380608328546,
138    404.673065729266,
139    486.296896820126,
140    480.225974100847,
141    457.318764401085,
142    418.177339948609,
143    414.399018364126,
144    409.678420185754,
145    404.768766016563,
146    401.699589920585,
147    402.44339040654,
148    398.138372541906,
149    396.010498627646,
150    390.165117432277,
151    375.850429417013,
152    365.567100746484,
153    365.429624610923
154 };
155
156 for (int i = 0; i < 48; i++) {
157     testFloatEquals(
158         test_electrical_load.dt_vec_hrs[i],
159         expected_dt_vec_hrs[i],
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals(
165         test_electrical_load.time_vec_hrs[i],
166         expected_time_vec_hrs[i],
167         __FILE__,
168         __LINE__
169     );
170
171     testFloatEquals(
172         test_electrical_load.load_vec_kW[i],
173         expected_load_vec_kW[i],
174         __FILE__,
175         __LINE__
176     );
177 }
178
179 // ===== END ATTRIBUTES ===== //
180
181 } /* try */
182
183
184 catch (...) {

```

```

185     //...
186
187     printGold(" ..... ");
188     printRed("FAIL");
189     std::cout << std::endl;
190     throw;
191 }
192
193
194 printGold(" ..... ");
195 printGreen("PASS");
196 std::cout << std::endl;
197 return 0;
198 } /* main() */

```

5.43 test/source/test_Model.cpp File Reference

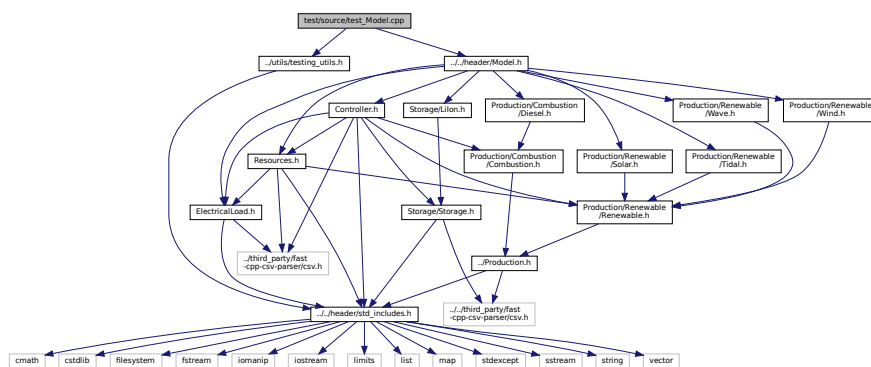
Testing suite for [Model](#) class.

```

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

```

Include dependency graph for test_Model.cpp:



Functions

- [int main](#) (int argc, char **argv)

5.43.1 Detailed Description

Testing suite for [Model](#) class.

A suite of tests for the [Model](#) class.

5.43.2 Function Documentation

5.43.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Model");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         bool error_flag = true;
42
43         try {
44             ModelInputs bad_model_inputs;
45             bad_model_inputs.path_2_electrical_load_time_series =
46                 "data/test/bad_path_240984069830.csv";
47
48             Model bad_model(bad_model_inputs);
49
50             error_flag = false;
51         } catch (...) {
52             // Task failed successfully! =P
53         }
54         if (not error_flag) {
55             expectedErrorNotDetected(__FILE__, __LINE__);
56         }
57
58         std::string path_2_electrical_load_time_series =
59             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
60
61         ModelInputs test_model_inputs;
62         test_model_inputs.path_2_electrical_load_time_series =
63             path_2_electrical_load_time_series;
64
65         Model test_model(test_model_inputs);
66
67         // ===== END CONSTRUCTION ===== //
68
69
70         // ===== ATTRIBUTES ===== //
71
72         testTruth(
73             test_model.electrical_load.path_2_electrical_load_time_series ==
74             path_2_electrical_load_time_series,
75             __FILE__,
76             __LINE__
77 );
78
79         testFloatEquals(
80             test_model.electrical_load.n_points,
81             8760,
82             __FILE__,
83             __LINE__
84 );
85
86         testFloatEquals(
87             test_model.electrical_load.n_years,
88             0.999886,
89             __FILE__,
90             __LINE__
91 );
92
93         testFloatEquals(
94             test_model.electrical_load.min_load_kW,
95             82.1211213927802,
96             __FILE__,
97             __LINE__
98 );
99
100        testFloatEquals(
101            test_model.electrical_load.mean_load_kW,
102            258.373472633202,
103            __FILE__,
104            __LINE__
105 );
106

```

```

107
108 testFloatEquals(
109     test_model.electrical_load.max_load_kW,
110     500,
111     __FILE__,
112     __LINE__
113 );
114
115
116 std::vector<double> expected_dt_vec_hrs (48, 1);
117
118 std::vector<double> expected_time_vec_hrs = {
119     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
120     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
121     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
122     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
123 };
124
125 std::vector<double> expected_load_vec_kW = {
126     360.253836463674,
127     355.171277826775,
128     353.776453532298,
129     353.75405737934,
130     346.592867404975,
131     340.132411175118,
132     337.354867340578,
133     340.644115618736,
134     363.639028500678,
135     378.787797779238,
136     372.215798201712,
137     395.093925731298,
138     402.325427142659,
139     386.907725462306,
140     380.709170928091,
141     372.062070914977,
142     372.328646856954,
143     391.841444284136,
144     394.029351759596,
145     383.369407765254,
146     381.093099675206,
147     382.604158946193,
148     390.744843709034,
149     383.13949492437,
150     368.150393976985,
151     364.629744480226,
152     363.572736804082,
153     359.854924202248,
154     355.207590170267,
155     349.094656012401,
156     354.365935871597,
157     343.380608328546,
158     404.673065729266,
159     486.296896820126,
160     480.225974100847,
161     457.318764401085,
162     418.177339948609,
163     414.399018364126,
164     409.678420185754,
165     404.768766016563,
166     401.699589920585,
167     402.44339040654,
168     398.138372541906,
169     396.010498627646,
170     390.165117432277,
171     375.850429417013,
172     365.567100746484,
173     365.429624610923
174 };
175
176 for (int i = 0; i < 48; i++) {
177     testFloatEquals(
178         test_model.electrical_load.dt_vec_hrs[i],
179         expected_dt_vec_hrs[i],
180         __FILE__,
181         __LINE__
182     );
183
184     testFloatEquals(
185         test_model.electrical_load.time_vec_hrs[i],
186         expected_time_vec_hrs[i],
187         __FILE__,
188         __LINE__
189     );
190
191     testFloatEquals(
192         test_model.electrical_load.load_vec_kW[i],
193         expected_load_vec_kW[i],

```

```

194     __FILE__,
195     __LINE__
196 );
197 }
198
199 // ===== END ATTRIBUTES ===== //
200
201
202
203 // ===== METHODS ===== //
204
205 // add Solar resource
206 int solar_resource_key = 0;
207 std::string path_2_solar_resource_data =
208     "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
209
210 test_model.addResource(
211     RenewableType :: SOLAR,
212     path_2_solar_resource_data,
213     solar_resource_key
214 );
215
216 std::vector<double> expected_solar_resource_vec_kWm2 = {
217     0,
218     0,
219     0,
220     0,
221     0,
222     0,
223     8.51702662684015E-05,
224     0.000348341567045,
225     0.00213793728593,
226     0.004099863613322,
227     0.000997135230553,
228     0.009534527624657,
229     0.022927996790616,
230     0.0136071715294,
231     0.002535134127751,
232     0.005206897515821,
233     0.005627658648597,
234     0.000701186722215,
235     0.00017119827089,
236     0,
237     0,
238     0,
239     0,
240     0,
241     0,
242     0,
243     0,
244     0,
245     0,
246     0,
247     0,
248     0.000141055102242,
249     0.00084525014743,
250     0.024893647822702,
251     0.091245556190749,
252     0.158722176731637,
253     0.152859680515876,
254     0.149922903895116,
255     0.13049996570866,
256     0.03081254222795,
257     0.001218928911125,
258     0.000206092647423,
259     0,
260     0,
261     0,
262     0,
263     0,
264     0
265 };
266
267 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
268     testFloatEquals(
269         test_model.resources.resource_map_1D[solar_resource_key][i],
270         expected_solar_resource_vec_kWm2[i],
271         __FILE__,
272         __LINE__
273     );
274 }
275
276
277 // add Tidal resource
278 int tidal_resource_key = 1;
279 std::string path_2_tidal_resource_data =
280     "data/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

```



```

281
282 test_model.addResource(
283     RenewableType :: TIDAL,
284     path_2_tidal_resource_data,
285     tidal_resource_key
286 );
287
288
289 // add Wave resource
290 int wave_resource_key = 2;
291 std::string path_2_wave_resource_data =
292     "data/test/waves_H_s_peak-8m_T_e_peak-15s_lyr_dt-1hr.csv";
293
294 test_model.addResource(
295     RenewableType :: WAVE,
296     path_2_wave_resource_data,
297     wave_resource_key
298 );
299
300
301 // add Wind resource
302 int wind_resource_key = 3;
303 std::string path_2_wind_resource_data =
304     "data/test/wind_speed_peak-25ms_lyr_dt-1hr.csv";
305
306 test_model.addResource(
307     RenewableType :: WIND,
308     path_2_wind_resource_data,
309     wind_resource_key
310 );
311
312
313 // add Diesel assets
314 DieselInputs diesel_inputs;
315 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
316 diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
317
318 test_model.addDiesel(diesel_inputs);
319
320 testFloatEquals(
321     test_model.combustion_ptr_vec.size(),
322     1,
323     __FILE__,
324     __LINE__
325 );
326
327 testFloatEquals(
328     test_model.combustion_ptr_vec[0]->type,
329     CombustionType :: DIESEL,
330     __FILE__,
331     __LINE__
332 );
333
334 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
335
336 test_model.addDiesel(diesel_inputs);
337
338 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
339
340 test_model.addDiesel(diesel_inputs);
341
342 testFloatEquals(
343     test_model.combustion_ptr_vec.size(),
344     3,
345     __FILE__,
346     __LINE__
347 );
348
349 std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
350
351 for (int i = 0; i < 3; i++) {
352     testFloatEquals(
353         test_model.combustion_ptr_vec[i]->capacity_kW,
354         expected_diesel_capacity_vec_kW[i],
355         __FILE__,
356         __LINE__
357     );
358 }
359
360 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
361
362 for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
363     test_model.addDiesel(diesel_inputs);
364 }
365
366
367 // add Solar asset

```

```
368 SolarInputs solar_inputs;
369 solar_inputs.resource_key = solar_resource_key;
370
371 test_model.addSolar(solar_inputs);
372
373 testFloatEquals(
374     test_model.renewable_ptr_vec.size(),
375     1,
376     __FILE__,
377     __LINE__
378 );
379
380 testFloatEquals(
381     test_model.renewable_ptr_vec[0]->type,
382     RenewableType :: SOLAR,
383     __FILE__,
384     __LINE__
385 );
386
387
388 // add Tidal asset
389 TidalInputs tidal_inputs;
390 tidal_inputs.resource_key = tidal_resource_key;
391
392 test_model.addTidal(tidal_inputs);
393
394 testFloatEquals(
395     test_model.renewable_ptr_vec.size(),
396     2,
397     __FILE__,
398     __LINE__
399 );
400
401 testFloatEquals(
402     test_model.renewable_ptr_vec[1]->type,
403     RenewableType :: TIDAL,
404     __FILE__,
405     __LINE__
406 );
407
408
409 // add Wave asset
410 WaveInputs wave_inputs;
411 wave_inputs.resource_key = wave_resource_key;
412
413 test_model.addWave(wave_inputs);
414
415 testFloatEquals(
416     test_model.renewable_ptr_vec.size(),
417     3,
418     __FILE__,
419     __LINE__
420 );
421
422 testFloatEquals(
423     test_model.renewable_ptr_vec[2]->type,
424     RenewableType :: WAVE,
425     __FILE__,
426     __LINE__
427 );
428
429
430 // add Wind asset
431 WindInputs wind_inputs;
432 wind_inputs.resource_key = wind_resource_key;
433
434 test_model.addWind(wind_inputs);
435
436 testFloatEquals(
437     test_model.renewable_ptr_vec.size(),
438     4,
439     __FILE__,
440     __LINE__
441 );
442
443 testFloatEquals(
444     test_model.renewable_ptr_vec[3]->type,
445     RenewableType :: WIND,
446     __FILE__,
447     __LINE__
448 );
449
450
451 // run
452 test_model.run();
453
454
```

```

455 // write results
456 test_model.writeResults("test/test_results/");
457
458
459 // test post-run attributes
460 for (int i = 0; i < test_model.electrical_load.n_points; i++) {
461     testLessThanOrEqualTo(
462         test_model.controller.net_load_vec_kW[i],
463         test_model.electrical_load.max_load_kW,
464         __FILE__,
465         __LINE__
466     );
467 }
468
469 testGreaterThan(
470     test_model.net_present_cost,
471     0,
472     __FILE__,
473     __LINE__
474 );
475
476 testFloatEquals(
477     test_model.total_dispatch_discharge_kWh,
478     2263351.62026685,
479     __FILE__,
480     __LINE__
481 );
482
483 testGreaterThan(
484     test_model.levellized_cost_of_energy_kWh,
485     0,
486     __FILE__,
487     __LINE__
488 );
489
490 testGreaterThan(
491     test_model.total_fuel_consumed_L,
492     0,
493     __FILE__,
494     __LINE__
495 );
496
497 testGreaterThan(
498     test_model.total_emissions.CO2_kg,
499     0,
500     __FILE__,
501     __LINE__
502 );
503
504 testGreaterThan(
505     test_model.total_emissions.CO_kg,
506     0,
507     __FILE__,
508     __LINE__
509 );
510
511 testGreaterThan(
512     test_model.total_emissions.NOx_kg,
513     0,
514     __FILE__,
515     __LINE__
516 );
517
518 testGreaterThan(
519     test_model.total_emissions.SOx_kg,
520     0,
521     __FILE__,
522     __LINE__
523 );
524
525 testGreaterThan(
526     test_model.total_emissions.CH4_kg,
527     0,
528     __FILE__,
529     __LINE__
530 );
531
532 testGreaterThan(
533     test_model.total_emissions.PM_kg,
534     0,
535     __FILE__,
536     __LINE__
537 );
538
539 // ===== END METHODS ===== //
540
541 } /* try */

```

```

542
543
544 catch (...) {
545     //...
546
547     printGold(" ..... ");
548     printRed("FAIL");
549     std::cout << std::endl;
550     throw;
551 }
552
553
554 printGold(" ..... ");
555 printGreen("PASS");
556 std::cout << std::endl;
557 return 0;
558 } /* main() */

```

5.44 test/source/test_Resources.cpp File Reference

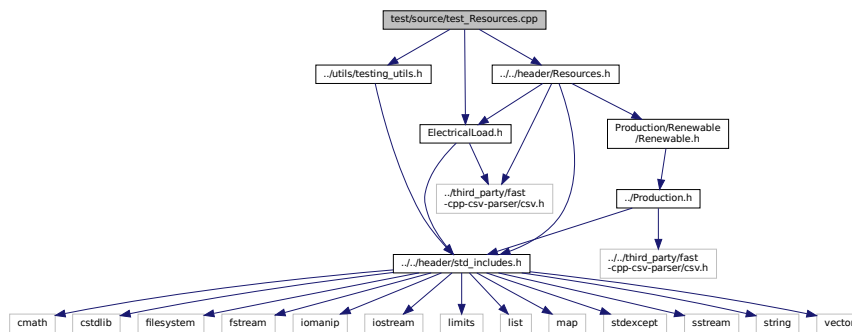
Testing suite for [Resources](#) class.

```

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

```

Include dependency graph for test_Resources.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.44.1 Detailed Description

Testing suite for [Resources](#) class.

A suite of tests for the [Resources](#) class.

5.44.2 Function Documentation

5.44.2.1 main()

```

int main (
    int argc,
    char ** argv )

28 {
29     #ifdef _WIN32
30         activateVirtualTerminal();
31     #endif /* _WIN32 */
32
33     printGold("\tTesting Resources");
34
35     srand(time(NULL));
36
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         std::string path_2_electrical_load_time_series =
43             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
44
45         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
46
47         Resources test_resources;
48
49         // ===== END CONSTRUCTION ===== //
50
51
52
53         // ===== ATTRIBUTES ===== //
54
55         testFloatEquals(
56             test_resources.resource_map_1D.size(),
57             0,
58             __FILE__,
59             __LINE__
60         );
61
62         testFloatEquals(
63             test_resources.path_map_1D.size(),
64             0,
65             __FILE__,
66             __LINE__
67         );
68
69         testFloatEquals(
70             test_resources.resource_map_2D.size(),
71             0,
72             __FILE__,
73             __LINE__
74         );
75
76         testFloatEquals(
77             test_resources.path_map_2D.size(),
78             0,
79             __FILE__,
80             __LINE__
81         );
82
83         // ===== END ATTRIBUTES ===== //
84
85
86         // ===== METHODS ===== //
87
88         int solar_resource_key = 0;
89         std::string path_2_solar_resource_data =
90             "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
91
92         test_resources.addResource(
93             RenewableType::SOLAR,
94             path_2_solar_resource_data,
95             solar_resource_key,
96             &test_electrical_load
97         );
98
99         bool error_flag = true;
100         try {
101             test_resources.addResource(
102                 RenewableType::SOLAR,
103                 path_2_solar_resource_data,
104                 solar_resource_key,
105                 &test_electrical_load
106             );
107

```

```

108     error_flag = false;
109 } catch (...) {
110     // Task failed successfully! =P
111 }
112 if (not error_flag) {
113     expectedErrorNotDetected(__FILE__, __LINE__);
114 }
115
116
117 try {
118     std::string path_2_solar_resource_data_BAD_TIMES =
119         "data/test/solar_GHI_peak-1kWm2_lyr_dt-1hr_BAD_TIMES.csv";
120
121     test_resources.addResource(
122         RenewableType::SOLAR,
123         path_2_solar_resource_data_BAD_TIMES,
124         -1,
125         &test_electrical_load
126     );
127
128     error_flag = false;
129 } catch (...) {
130     // Task failed successfully! =P
131 }
132 if (not error_flag) {
133     expectedErrorNotDetected(__FILE__, __LINE__);
134 }
135
136
137 try {
138     std::string path_2_solar_resource_data_BAD_LENGTH =
139         "data/test/solar_GHI_peak-1kWm2_lyr_dt-1hr_BAD_LENGTH.csv";
140
141     test_resources.addResource(
142         RenewableType::SOLAR,
143         path_2_solar_resource_data_BAD_LENGTH,
144         -2,
145         &test_electrical_load
146     );
147
148     error_flag = false;
149 } catch (...) {
150     // Task failed successfully! =P
151 }
152 if (not error_flag) {
153     expectedErrorNotDetected(__FILE__, __LINE__);
154 }
155
156 std::vector<double> expected_solar_resource_vec_kWm2 = {
157     0,
158     0,
159     0,
160     0,
161     0,
162     0,
163     8.51702662684015E-05,
164     0.000348341567045,
165     0.00213793728593,
166     0.004099863613322,
167     0.000997135230553,
168     0.009534527624657,
169     0.022927996790616,
170     0.0136071715294,
171     0.002535134127751,
172     0.005206897515821,
173     0.005627658648597,
174     0.000701186722215,
175     0.00017119827089,
176     0,
177     0,
178     0,
179     0,
180     0,
181     0,
182     0,
183     0,
184     0,
185     0,
186     0,
187     0,
188     0.000141055102242,
189     0.00084525014743,
190     0.024893647822702,
191     0.091245556190749,
192     0.158722176731637,
193     0.152859680515876,
194     0.149922903895116,

```

```

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

```

```

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

```



```

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

```

```

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

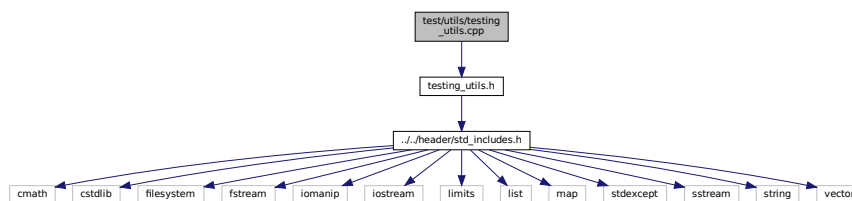
```

5.45 test/utls/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 $x \geq y$.
- void `testLessThan` (double x, double y, std::string file, int line)
Tests if $x < y$.
- void `testLessThanOrEqualTo` (double x, double y, std::string file, int line)
Tests if $x \leq y$.
- void `testTruth` (bool statement, std::string file, int line)
Tests if the given statement is true.
- void `expectedErrorNotDetected` (std::string file, int line)
A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.45.1 Detailed Description

Header file for various PGMcpp testing utilities.

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

5.45.2 Function Documentation

5.45.2.1 `expectedErrorNotDetected()`

```
void expectedErrorNotDetected (
    std::string file,
    int line )
```

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

Parameters

<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```
432 {
433     std::string error_str = "\n ERROR   failed to throw expected error prior to line ";
434     error_str += std::to_string(line);
```

```

435     error_str += " of ";
436     error_str += file;
437
438     #ifdef _WIN32
439         std::cout << error_str << std::endl;
440     #endif
441
442     throw std::runtime_error(error_str);
443     return;
444 } /* expectedErrorNotDetected() */

```

5.45.2.2 printGold()

```

void printGold (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */

```

5.45.2.3 printGreen()

```

void printGreen (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

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

```

5.45.2.4 printRed()

```

void printRed (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to <code>std::cout</code> .
------------------	---

```

104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */

```

5.45.2.5 testFloatEquals()

```

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

```

Tests for the equality of two floating point numbers *x* and *y* (to within `FLOAT_TOLERANCE`).

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```

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

```

5.45.2.6 testGreaterThan()

```

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

```

Tests if $x > y$.

Parameters

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

```

191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209
210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */

```

5.45.2.7 testGreaterThanOrEqualTo()

```

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

```

Tests if $x \geq y$.

Parameters

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

```

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

```

```

262     return;
263 } /* testGreaterThanOrEqualTo() */

```

5.45.2.8 testLessThan()

```

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

```

Tests if $x < y$.

Parameters

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

```

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

```

5.45.2.9 testLessThanOrEqualTo()

```

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

```

Tests if $x \leq y$.

Parameters

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

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.45.2.10 testTruth()

```

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

```

Tests if the given statement is true.

Parameters

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

```

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

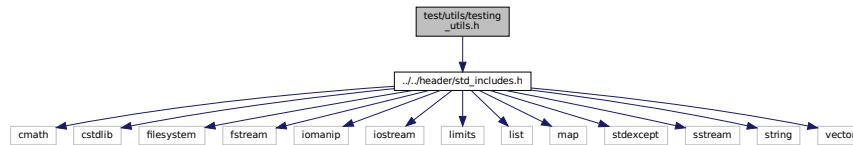
```

5.46 test/utils/testing_utils.h File Reference

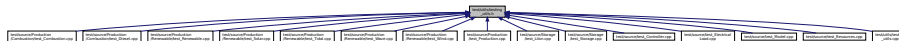
Header file for various PGMcpp testing utilities.


```
#include "../..//header/std_includes.h"
```

Include dependency graph for testing_utils.h:



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



Macros

- `#define FLOAT_TOLERANCE 1e-6`
A tolerance for application to floating point equality tests.

Functions

- void [printGreen](#) (std::string)
A function that sends green text to std::cout.
- void [printGold](#) (std::string)
A function that sends gold text to std::cout.
- void [printRed](#) (std::string)
A function that sends red text to std::cout.
- void [testFloatEquals](#) (double, double, std::string, int)
Tests for the equality of two floating point numbers x and y (to within `FLOAT_TOLERANCE`).
- void [testGreaterThan](#) (double, double, std::string, int)
Tests if $x > y$.
- void [testGreaterThanOrEqualTo](#) (double, double, std::string, int)
Tests if $x \geq y$.
- void [testLessThan](#) (double, double, std::string, int)
Tests if $x < y$.
- void [testLessThanOrEqualTo](#) (double, double, std::string, int)
Tests if $x \leq y$.
- void [testTruth](#) (bool, std::string, int)
Tests if the given statement is true.
- void [expectedErrorNotDetected](#) (std::string, int)
A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.46.1 Detailed Description

Header file for various PGMcpp testing utilities.

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

5.46.2 Macro Definition Documentation

5.46.2.1 FLOAT_TOLERANCE

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

A tolerance for application to floating point equality tests.

5.46.3 Function Documentation

5.46.3.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
    std::string file,
    int line )
```

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

Parameters

<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

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

5.46.3.2 printGold()

```
void printGold (
    std::string input_str )
```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */

```

5.46.3.3 printGreen()

```

void printGreen (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

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

```

5.46.3.4 printRed()

```

void printRed (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */

```

5.46.3.5 testFloatEquals()

```

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

```

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

Parameters

<i>x</i>	The first of two numbers to test.
----------	-----------------------------------

Parameters

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

```

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

```

5.46.3.6 testGreaterThan()

```

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

```

Tests if $x > y$.

Parameters

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

```

191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209

```

```

210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */

```

5.46.3.7 testGreaterThanOrEqualTo()

```

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

```

Tests if $x \geq y$.

Parameters

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

```

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

```

5.46.3.8 testLessThan()

```

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

```

Tests if $x < y$.

Parameters

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

```

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

```

5.46.3.9 testLessThanOrEqualTo()

```

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

```

Tests if $x \leq y$.

Parameters

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

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.46.3.10 testTruth()

```

void testTruth (

```

```
bool statement,  
std::string file,  
int line )
```

Tests if the given statement is true.

Parameters

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

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


Bibliography

- Dr. B. Buckham, Dr. C. Crawford, Dr. I. Beya Marshall, and Dr. B. Whitby. Wei Wai Kum Tidal Prefeasibility Study - Tidal Resource Assessment. Technical report, PRIMED, 2023. Internal: P2202E_BRKLYG+WEI WAI KUM_R01_V20230613v3. 125
- HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 84
- HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 82, 84
- HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 41, 50
- HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 41, 50
- HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 41, 50
- HOMER. How HOMER Calculates the PV Array Power Output, 2023f. URL https://www.homerenergy.com/products/pro/docs/latest/how_homer_calculates_the_pv_array_power_output.html. 116
- HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 84
- HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 82
- HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 84
- Dr. S.L. MacDougall. Commercial Potential of Marine Renewables in British Columbia. Technical report, S.L. MacDougall Research & Consulting, 2019. Submitted to Natural Resources Canada. 127, 128, 141, 142
- Dr. B. Robertson, Dr. H. Bailey, M. Leary, and Dr. B. Buckham. A methodology for architecture agnostic and time flexible representations of wave energy converter performance. *Applied Energy*, 287, 2021. doi:10.1016/j.apenergy.2021.116588. 140

Index

- __applyCycleChargingControl_CHARGING
Controller, [22](#)
- __applyCycleChargingControl_DISCHARGING
Controller, [23](#)
- __applyLoadFollowingControl_CHARGING
Controller, [24](#)
- __applyLoadFollowingControl_DISCHARGING
Controller, [24](#)
- __checkInputs
Combustion, [11](#)
Diesel, [39](#)
Model, [64](#)
Production, [81](#)
Renewable, [94](#)
Solar, [113](#)
Tidal, [125](#)
Wave, [138](#)
Wind, [153](#)
- __checkResourceKey1D
Resources, [101](#)
- __checkResourceKey2D
Resources, [101](#)
- __checkTimePoint
Resources, [102](#)
- __computeCubicProductionkW
Tidal, [125](#)
- __computeEconomics
Model, [64](#)
- __computeExponentialProductionkW
Tidal, [126](#)
Wind, [154](#)
- __computeFuelAndEmissions
Model, [65](#)
- __computeGaussianProductionkW
Wave, [139](#)
- __computeLevellizedCostOfEnergy
Model, [65](#)
- __computeLookupProductionkW
Tidal, [127](#)
Wave, [140](#)
Wind, [154](#)
- __computeNetLoad
Controller, [25](#)
- __computeNetPresentCost
Model, [66](#)
- __computeParaboloidProductionkW
Wave, [140](#)
- __computeRealDiscountAnnual
Production, [82](#)
- __constructCombustionMap
Controller, [26](#)
- __getGenericCapitalCost
Diesel, [40](#)
Solar, [113](#)
Tidal, [127](#)
Wave, [141](#)
Wind, [155](#)
- __getGenericFuelIntercept
Diesel, [41](#)
- __getGenericFuelSlope
Diesel, [41](#)
- __getGenericOpMaintCost
Diesel, [41](#)
Solar, [113](#)
Tidal, [127](#)
Wave, [141](#)
Wind, [155](#)
- __getRenewableProduction
Controller, [28](#)
- __handleCombustionDispatch
Controller, [29](#)
- __handleReplacement
Production, [82](#)
- __handleStartStop
Diesel, [42](#)
Renewable, [94](#)
- __handleStorageCharging
Controller, [30](#)
- __handleStorageDischarging
Controller, [31](#)
- __readSolarResource
Resources, [103](#)
- __readTidalResource
Resources, [104](#)
- __readWaveResource
Resources, [104](#)
- __readWindResource
Resources, [105](#)
- __throwLengthError
Resources, [106](#)
- __writeSummary
Combustion, [11](#)
Diesel, [43](#)
Model, [66](#)
Renewable, [94](#)
Solar, [113](#)
Tidal, [128](#)
Wave, [142](#)

- Wind, [155](#)
- __writeTimeSeries
 - Combustion, [11](#)
 - Diesel, [44](#)
 - Model, [69](#)
 - Renewable, [94](#)
 - Solar, [115](#)
 - Tidal, [129](#)
 - Wave, [143](#)
 - Wind, [157](#)
- ~Combustion
 - Combustion, [11](#)
- ~Controller
 - Controller, [22](#)
- ~Diesel
 - Diesel, [38](#)
- ~ElectricalLoad
 - ElectricalLoad, [54](#)
- ~Lilon
 - Lilon, [61](#)
- ~Model
 - Model, [64](#)
- ~Production
 - Production, [81](#)
- ~Renewable
 - Renewable, [93](#)
- ~Resources
 - Resources, [100](#)
- ~Solar
 - Solar, [112](#)
- ~Storage
 - Storage, [120](#)
- ~Tidal
 - Tidal, [124](#)
- ~Wave
 - Wave, [138](#)
- ~Wind
 - Wind, [153](#)
- addDiesel
 - Model, [69](#)
- addResource
 - Model, [70](#)
 - Resources, [107](#)
- addSolar
 - Model, [70](#)
- addTidal
 - Model, [71](#)
- addWave
 - Model, [71](#)
- addWind
 - Model, [71](#)
- applyDispatchControl
 - Controller, [31](#)
- capacity_kW
 - Production, [85](#)
 - ProductionInputs, [90](#)
- capital_cost
 - DieselInputs, [50](#)
 - Production, [85](#)
 - SolarInputs, [118](#)
 - TidalInputs, [134](#)
 - WaveInputs, [148](#)
 - WindInputs, [161](#)
- capital_cost_vec
 - Production, [85](#)
- CH4_emissions_intensity_kgL
 - Combustion, [16](#)
 - DieselInputs, [50](#)
- CH4_emissions_vec_kg
 - Combustion, [16](#)
- CH4_kg
 - Emissions, [58](#)
- clear
 - Controller, [33](#)
 - ElectricalLoad, [54](#)
 - Model, [72](#)
 - Resources, [108](#)
- CO2_emissions_intensity_kgL
 - Combustion, [16](#)
 - DieselInputs, [50](#)
- CO2_emissions_vec_kg
 - Combustion, [16](#)
- CO2_kg
 - Emissions, [58](#)
- CO_emissions_intensity_kgL
 - Combustion, [16](#)
 - DieselInputs, [51](#)
- CO_emissions_vec_kg
 - Combustion, [16](#)
- CO_kg
 - Emissions, [58](#)
- Combustion, [7](#)
 - __checkInputs, [11](#)
 - __writeSummary, [11](#)
 - __writeTimeSeries, [11](#)
 - ~Combustion, [11](#)
 - CH4_emissions_intensity_kgL, [16](#)
 - CH4_emissions_vec_kg, [16](#)
 - CO2_emissions_intensity_kgL, [16](#)
 - CO2_emissions_vec_kg, [16](#)
 - CO_emissions_intensity_kgL, [16](#)
 - CO_emissions_vec_kg, [16](#)
 - Combustion, [9, 10](#)
 - commit, [12](#)
 - computeEconomics, [13](#)
 - computeFuelAndEmissions, [13](#)
 - fuel_consumption_vec_L, [17](#)
 - fuel_cost_L, [17](#)
 - fuel_cost_vec, [17](#)
 - getEmissionskg, [13](#)
 - getFuelConsumptionL, [14](#)
 - linear_fuel_intercept_LkWh, [17](#)
 - linear_fuel_slope_LkWh, [17](#)
 - NOx_emissions_intensity_kgL, [17](#)
 - NOx_emissions_vec_kg, [18](#)

- PM_emissions_intensity_kgL, 18
- PM_emissions_vec_kg, 18
- requestProductionkW, 14
- SOx_emissions_intensity_kgL, 18
- SOx_emissions_vec_kg, 18
- total_emissions, 18
- total_fuel_consumed_L, 19
- type, 19
- writeResults, 15
- Combustion.h
 - CombustionType, 167
 - DIESEL, 168
 - N_COMBUSTION_TYPES, 168
- combustion_inputs
 - DieselInputs, 51
- combustion_map
 - Controller, 34
- combustion_ptr_vec
 - Model, 74
- CombustionInputs, 19
 - production_inputs, 20
- CombustionType
 - Combustion.h, 167
- commit
 - Combustion, 12
 - Diesel, 45
 - Production, 83
 - Renewable, 95
 - Solar, 115
 - Tidal, 130
 - Wave, 144
 - Wind, 158
- computeEconomics
 - Combustion, 13
 - Production, 84
 - Renewable, 95
- computeFuelAndEmissions
 - Combustion, 13
- computeProductionkW
 - Renewable, 96
 - Solar, 116
 - Tidal, 131
 - Wave, 145
 - Wind, 158
- control_mode
 - Controller, 34
 - ModelInputs, 77
- control_string
 - Controller, 34
- Controller, 20
 - __applyCycleChargingControl_CHARGING, 22
 - __applyCycleChargingControl_DISCHARGING, 23
 - __applyLoadFollowingControl_CHARGING, 24
 - __applyLoadFollowingControl_DISCHARGING, 24
 - __computeNetLoad, 25
 - __constructCombustionMap, 26
 - __getRenewableProduction, 28
 - __handleCombustionDispatch, 29
 - __handleStorageCharging, 30
 - __handleStorageDischarging, 31
 - ~Controller, 22
 - applyDispatchControl, 31
 - clear, 33
 - combustion_map, 34
 - control_mode, 34
 - control_string, 34
 - Controller, 22
 - init, 33
 - missed_load_vec_kW, 34
 - net_load_vec_kW, 35
 - setControlMode, 33
- controller
 - Model, 74
- Controller.h
 - ControlMode, 164
 - CYCLE_CHARGING, 164
 - LOAD_FOLLOWING, 164
 - N_CONTROL_MODES, 164
- ControlMode
 - Controller.h, 164
- curtailment_vec_kW
 - Production, 85
- CYCLE_CHARGING
 - Controller.h, 164
- derating
 - Solar, 117
 - SolarInputs, 119
- design_energy_period_s
 - Wave, 146
 - WaveInputs, 148
- design_significant_wave_height_m
 - Wave, 146
 - WaveInputs, 148
- design_speed_ms
 - Tidal, 132
 - TidalInputs, 134
 - Wind, 159
 - WindInputs, 161
- DIESEL
 - Combustion.h, 168
- Diesel, 35
 - __checkInputs, 39
 - __getGenericCapitalCost, 40
 - __getGenericFuelIntercept, 41
 - __getGenericFuelSlope, 41
 - __getGenericOpMaintCost, 41
 - __handleStartStop, 42
 - __writeSummary, 43
 - __writeTimeSeries, 44
 - ~Diesel, 38
 - commit, 45
 - Diesel, 37
 - minimum_load_ratio, 48
 - minimum_runtime_hrs, 48
 - requestProductionkW, 47
 - time_since_last_start_hrs, 48

- DieselInputs, 49
 - capital_cost, 50
 - CH4_emissions_intensity_kgL, 50
 - CO2_emissions_intensity_kgL, 50
 - CO_emissions_intensity_kgL, 51
 - combustion_inputs, 51
 - fuel_cost_L, 51
 - linear_fuel_intercept_LkWh, 51
 - linear_fuel_slope_LkWh, 51
 - minimum_load_ratio, 51
 - minimum_runtime_hrs, 52
 - NOx_emissions_intensity_kgL, 52
 - operation_maintenance_cost_kWh, 52
 - PM_emissions_intensity_kgL, 52
 - replace_running_hrs, 52
 - SOx_emissions_intensity_kgL, 52
- dispatch_vec_kW
 - Production, 85
- dt_vec_hrs
 - ElectricalLoad, 56
- electrical_load
 - Model, 75
- ElectricalLoad, 53
 - ~ElectricalLoad, 54
 - clear, 54
 - dt_vec_hrs, 56
 - ElectricalLoad, 54
 - load_vec_kW, 56
 - max_load_kW, 56
 - mean_load_kW, 56
 - min_load_kW, 57
 - n_points, 57
 - n_years, 57
 - path_2_electrical_load_time_series, 57
 - readLoadData, 55
 - time_vec_hrs, 57
- Emissions, 58
 - CH4_kg, 58
 - CO2_kg, 58
 - CO_kg, 58
 - NOx_kg, 59
 - PM_kg, 59
 - SOx_kg, 59
- expectedErrorNotDetected
 - testing_utils.cpp, 235
 - testing_utils.h, 242
- FLOAT_TOLERANCE
 - testing_utils.h, 242
- fuel_consumption_vec_L
 - Combustion, 17
- fuel_cost_L
 - Combustion, 17
 - DieselInputs, 51
- fuel_cost_vec
 - Combustion, 17
- getEmissionskg
 - Combustion, 13
- getFuelConsumptionL
 - Combustion, 14
- header/Controller.h, 163
- header/ElectricalLoad.h, 164
- header/Model.h, 165
- header/Production/Combustion/Combustion.h, 166
- header/Production/Combustion/Diesel.h, 168
- header/Production/Production.h, 169
- header/Production/Renewable/Renewable.h, 170
- header/Production/Renewable/Solar.h, 171
- header/Production/Renewable/Tidal.h, 172
- header/Production/Renewable/Wave.h, 173
- header/Production/Renewable/Wind.h, 175
- header/Resources.h, 176
- header/std_includes.h, 177
- header/Storage/Lilon.h, 178
- header/Storage/Storage.h, 179
- init
 - Controller, 33
- is_running
 - Production, 85
- is_running_vec
 - Production, 86
- is_sunk
 - Production, 86
 - ProductionInputs, 90
- levellized_cost_of_energy_kWh
 - Model, 75
 - Production, 86
- Lilon, 60
 - ~Lilon, 61
 - Lilon, 61
- linear_fuel_intercept_LkWh
 - Combustion, 17
 - DieselInputs, 51
- linear_fuel_slope_LkWh
 - Combustion, 17
 - DieselInputs, 51
- LOAD_FOLLOWING
 - Controller.h, 164
- load_vec_kW
 - ElectricalLoad, 56
- main
 - test_Combustion.cpp, 189
 - test_Controller.cpp, 217
 - test_Diesel.cpp, 191
 - test_ElectricalLoad.cpp, 219
 - test_Lilon.cpp, 215
 - test_Model.cpp, 221
 - test_Production.cpp, 212
 - test_Renewable.cpp, 196
 - test_Resources.cpp, 228
 - test_Solar.cpp, 198
 - test_Storage.cpp, 216

- test_Tidal.cpp, 201
- test_Wave.cpp, 205
- test_Wind.cpp, 209
- max_load_kW
 - ElectricalLoad, 56
- mean_load_kW
 - ElectricalLoad, 56
- min_load_kW
 - ElectricalLoad, 57
- minimum_load_ratio
 - Diesel, 48
 - DieselInputs, 51
- minimum_runtime_hrs
 - Diesel, 48
 - DieselInputs, 52
- missed_load_vec_kW
 - Controller, 34
- Model, 61
 - __checkInputs, 64
 - __computeEconomics, 64
 - __computeFuelAndEmissions, 65
 - __computeLevellizedCostOfEnergy, 65
 - __computeNetPresentCost, 66
 - __writeSummary, 66
 - __writeTimeSeries, 69
 - ~Model, 64
 - addDiesel, 69
 - addResource, 70
 - addSolar, 70
 - addTidal, 71
 - addWave, 71
 - addWind, 71
 - clear, 72
 - combustion_ptr_vec, 74
 - controller, 74
 - electrical_load, 75
 - levellized_cost_of_energy_kWh, 75
 - Model, 63
 - net_present_cost, 75
 - renewable_ptr_vec, 75
 - reset, 72
 - resources, 75
 - run, 73
 - storage_ptr_vec, 75
 - total_dispatch_discharge_kWh, 76
 - total_emissions, 76
 - total_fuel_consumed_L, 76
 - writeResults, 73
- ModelInputs, 76
 - control_mode, 77
 - path_2_electrical_load_time_series, 77
- N_COMBUSTION_TYPES
 - Combustion.h, 168
- N_CONTROL_MODES
 - Controller.h, 164
- n_points
 - ElectricalLoad, 57
 - Production, 86
- N_RENEWABLE_TYPES
 - Renewable.h, 171
- n_replacements
 - Production, 86
- n_starts
 - Production, 86
- N_TIDAL_POWER_PRODUCTION_MODELS
 - Tidal.h, 173
- N_WAVE_POWER_PRODUCTION_MODELS
 - Wave.h, 174
- N_WIND_POWER_PRODUCTION_MODELS
 - Wind.h, 176
- n_years
 - ElectricalLoad, 57
 - Production, 87
- net_load_vec_kW
 - Controller, 35
- net_present_cost
 - Model, 75
 - Production, 87
- nominal_discount_annual
 - Production, 87
 - ProductionInputs, 90
- nominal_inflation_annual
 - Production, 87
 - ProductionInputs, 90
- NOx_emissions_intensity_kgL
 - Combustion, 17
 - DieselInputs, 52
- NOx_emissions_vec_kg
 - Combustion, 18
- NOx_kg
 - Emissions, 59
- operation_maintenance_cost_kWh
 - DieselInputs, 52
 - Production, 87
 - SolarInputs, 119
 - TidalInputs, 134
 - WaveInputs, 149
 - WindInputs, 161
- operation_maintenance_cost_vec
 - Production, 87
- path_2_electrical_load_time_series
 - ElectricalLoad, 57
 - ModelInputs, 77
- path_map_1D
 - Resources, 108
- path_map_2D
 - Resources, 109
- PM_emissions_intensity_kgL
 - Combustion, 18
 - DieselInputs, 52
- PM_emissions_vec_kg
 - Combustion, 18
- PM_kg
 - Emissions, 59
- power_model

- Tidal, 132
- TidalInputs, 134
- Wave, 146
- WaveInputs, 149
- Wind, 159
- WindInputs, 161
- power_model_string
 - Tidal, 132
 - Wave, 147
 - Wind, 159
- print_flag
 - Production, 88
 - ProductionInputs, 90
- printGold
 - testing_utils.cpp, 236
 - testing_utils.h, 242
- printGreen
 - testing_utils.cpp, 236
 - testing_utils.h, 243
- printRed
 - testing_utils.cpp, 236
 - testing_utils.h, 243
- Production, 77
 - __checkInputs, 81
 - __computeRealDiscountAnnual, 82
 - __handleReplacement, 82
 - ~Production, 81
 - capacity_kW, 85
 - capital_cost, 85
 - capital_cost_vec, 85
 - commit, 83
 - computeEconomics, 84
 - curtailment_vec_kW, 85
 - dispatch_vec_kW, 85
 - is_running, 85
 - is_running_vec, 86
 - is_sunk, 86
 - levellized_cost_of_energy_kWh, 86
 - n_points, 86
 - n_replacements, 86
 - n_starts, 86
 - n_years, 87
 - net_present_cost, 87
 - nominal_discount_annual, 87
 - nominal_inflation_annual, 87
 - operation_maintenance_cost_kWh, 87
 - operation_maintenance_cost_vec, 87
 - print_flag, 88
 - Production, 79, 80
 - production_vec_kW, 88
 - real_discount_annual, 88
 - replace_running_hrs, 88
 - running_hours, 88
 - storage_vec_kW, 88
 - total_dispatch_kWh, 89
 - type_str, 89
- production_inputs
 - CombustionInputs, 20
 - RenewableInputs, 98
- production_vec_kW
 - Production, 88
- ProductionInputs, 89
 - capacity_kW, 90
 - is_sunk, 90
 - nominal_discount_annual, 90
 - nominal_inflation_annual, 90
 - print_flag, 90
 - replace_running_hrs, 90
- PYBIND11_MODULE
 - PYBIND11_PGM.cpp, 180
- PYBIND11_PGM.cpp
 - PYBIND11_MODULE, 180
- pybindings/PYBIND11_PGM.cpp, 179
- readLoadData
 - ElectricalLoad, 55
- real_discount_annual
 - Production, 88
- Renewable, 91
 - __checkInputs, 94
 - __handleStartStop, 94
 - __writeSummary, 94
 - __writeTimeSeries, 94
 - ~Renewable, 93
 - commit, 95
 - computeEconomics, 95
 - computeProductionkW, 96
 - Renewable, 92, 93
 - resource_key, 97
 - type, 97
 - writeResults, 96
- Renewable.h
 - N_RENEWABLE_TYPES, 171
 - RenewableType, 170
 - SOLAR, 171
 - TIDAL, 171
 - WAVE, 171
 - WIND, 171
- renewable_inputs
 - SolarInputs, 119
 - TidalInputs, 134
 - WaveInputs, 149
 - WindInputs, 162
- renewable_ptr_vec
 - Model, 75
- RenewableInputs, 98
 - production_inputs, 98
- RenewableType
 - Renewable.h, 170
- replace_running_hrs
 - DieselInputs, 52
 - Production, 88
 - ProductionInputs, 90
- requestProductionkW
 - Combustion, 14
 - Diesel, 47
- reset

- Model, 72
- resource_key
 - Renewable, 97
 - SolarInputs, 119
 - TidalInputs, 134
 - WaveInputs, 149
 - WindInputs, 162
- resource_map_1D
 - Resources, 109
- resource_map_2D
 - Resources, 109
- Resources, 99
 - __checkResourceKey1D, 101
 - __checkResourceKey2D, 101
 - __checkTimePoint, 102
 - __readSolarResource, 103
 - __readTidalResource, 104
 - __readWaveResource, 104
 - __readWindResource, 105
 - __throwLengthError, 106
 - ~Resources, 100
 - addResource, 107
 - clear, 108
 - path_map_1D, 108
 - path_map_2D, 109
 - resource_map_1D, 109
 - resource_map_2D, 109
 - Resources, 100
 - string_map_1D, 109
 - string_map_2D, 109
- resources
 - Model, 75
- run
 - Model, 73
- running_hours
 - Production, 88
- setControlMode
 - Controller, 33
- SOLAR
 - Renewable.h, 171
- Solar, 110
 - __checkInputs, 113
 - __getGenericCapitalCost, 113
 - __getGenericOpMaintCost, 113
 - __writeSummary, 113
 - __writeTimeSeries, 115
 - ~Solar, 112
 - commit, 115
 - computeProductionkW, 116
 - derating, 117
 - Solar, 111
- SolarInputs, 117
 - capital_cost, 118
 - derating, 119
 - operation_maintenance_cost_kWh, 119
 - renewable_inputs, 119
 - resource_key, 119
- source/Controller.cpp, 181
- source/ElectricalLoad.cpp, 182
- source/Model.cpp, 182
- source/Production/Combustion/Combustion.cpp, 183
- source/Production/Combustion/Diesel.cpp, 183
- source/Production/Production.cpp, 184
- source/Production/Renewable/Renewable.cpp, 184
- source/Production/Renewable/Solar.cpp, 185
- source/Production/Renewable/Tidal.cpp, 185
- source/Production/Renewable/Wave.cpp, 186
- source/Production/Renewable/Wind.cpp, 186
- source/Resources.cpp, 187
- source/Storage/Lilon.cpp, 187
- source/Storage/Storage.cpp, 188
- SOx_emissions_intensity_kgL
 - Combustion, 18
 - DieselInputs, 52
- SOx_emissions_vec_kg
 - Combustion, 18
- SOx_kg
 - Emissions, 59
- Storage, 120
 - ~Storage, 120
 - Storage, 120
- storage_ptr_vec
 - Model, 75
- storage_vec_kW
 - Production, 88
- string_map_1D
 - Resources, 109
- string_map_2D
 - Resources, 109
- test/source/Production/Combustion/test_Combustion.cpp, 188
- test/source/Production/Combustion/test_Diesel.cpp, 190
- test/source/Production/Renewable/test_Renewable.cpp, 196
- test/source/Production/Renewable/test_Solar.cpp, 197
- test/source/Production/Renewable/test_Tidal.cpp, 201
- test/source/Production/Renewable/test_Wave.cpp, 204
- test/source/Production/Renewable/test_Wind.cpp, 208
- test/source/Production/test_Production.cpp, 212
- test/source/Storage/test_Lilon.cpp, 214
- test/source/Storage/test_Storage.cpp, 216
- test/source/test_Controller.cpp, 217
- test/source/test_ElectricalLoad.cpp, 218
- test/source/test_Model.cpp, 221
- test/source/test_Resources.cpp, 228
- test/Utils/testing_utils.cpp, 234
- test/Utils/testing_utils.h, 240
- test_Combustion.cpp
 - main, 189
- test_Controller.cpp
 - main, 217
- test_Diesel.cpp
 - main, 191
- test_ElectricalLoad.cpp
 - main, 219

- test_Lilon.cpp
 - main, [215](#)
- test_Model.cpp
 - main, [221](#)
- test_Production.cpp
 - main, [212](#)
- test_Renewable.cpp
 - main, [196](#)
- test_Resources.cpp
 - main, [228](#)
- test_Solar.cpp
 - main, [198](#)
- test_Storage.cpp
 - main, [216](#)
- test_Tidal.cpp
 - main, [201](#)
- test_Wave.cpp
 - main, [205](#)
- test_Wind.cpp
 - main, [209](#)
- testFloatEquals
 - testing_utils.cpp, [237](#)
 - testing_utils.h, [243](#)
- testGreaterThan
 - testing_utils.cpp, [237](#)
 - testing_utils.h, [244](#)
- testGreaterThanOrEqualTo
 - testing_utils.cpp, [238](#)
 - testing_utils.h, [245](#)
- testing_utils.cpp
 - expectedErrorNotDetected, [235](#)
 - printGold, [236](#)
 - printGreen, [236](#)
 - printRed, [236](#)
 - testFloatEquals, [237](#)
 - testGreaterThan, [237](#)
 - testGreaterThanOrEqualTo, [238](#)
 - testLessThan, [239](#)
 - testLessThanOrEqualTo, [239](#)
 - testTruth, [240](#)
- testing_utils.h
 - expectedErrorNotDetected, [242](#)
 - FLOAT_TOLERANCE, [242](#)
 - printGold, [242](#)
 - printGreen, [243](#)
 - printRed, [243](#)
 - testFloatEquals, [243](#)
 - testGreaterThan, [244](#)
 - testGreaterThanOrEqualTo, [245](#)
 - testLessThan, [245](#)
 - testLessThanOrEqualTo, [246](#)
 - testTruth, [246](#)
- testLessThan
 - testing_utils.cpp, [239](#)
 - testing_utils.h, [245](#)
- testLessThanOrEqualTo
 - testing_utils.cpp, [239](#)
 - testing_utils.h, [246](#)
- testTruth
 - testing_utils.cpp, [240](#)
 - testing_utils.h, [246](#)
- TIDAL
 - Renewable.h, [171](#)
- Tidal, [121](#)
 - __checkInputs, [125](#)
 - __computeCubicProductionkW, [125](#)
 - __computeExponentialProductionkW, [126](#)
 - __computeLookupProductionkW, [127](#)
 - __getGenericCapitalCost, [127](#)
 - __getGenericOpMaintCost, [127](#)
 - __writeSummary, [128](#)
 - __writeTimeSeries, [129](#)
 - ~Tidal, [124](#)
 - commit, [130](#)
 - computeProductionkW, [131](#)
 - design_speed_ms, [132](#)
 - power_model, [132](#)
 - power_model_string, [132](#)
 - Tidal, [123](#)
- Tidal.h
 - N_TIDAL_POWER_PRODUCTION_MODELS, [173](#)
 - TIDAL_POWER_CUBIC, [173](#)
 - TIDAL_POWER_EXPONENTIAL, [173](#)
 - TIDAL_POWER_LOOKUP, [173](#)
 - TidalPowerProductionModel, [173](#)
- TIDAL_POWER_CUBIC
 - Tidal.h, [173](#)
- TIDAL_POWER_EXPONENTIAL
 - Tidal.h, [173](#)
- TIDAL_POWER_LOOKUP
 - Tidal.h, [173](#)
- TidalInputs, [133](#)
 - capital_cost, [134](#)
 - design_speed_ms, [134](#)
 - operation_maintenance_cost_kWh, [134](#)
 - power_model, [134](#)
 - renewable_inputs, [134](#)
 - resource_key, [134](#)
- TidalPowerProductionModel
 - Tidal.h, [173](#)
- time_since_last_start_hrs
 - Diesel, [48](#)
- time_vec_hrs
 - ElectricalLoad, [57](#)
- total_dispatch_discharge_kWh
 - Model, [76](#)
- total_dispatch_kWh
 - Production, [89](#)
- total_emissions
 - Combustion, [18](#)
 - Model, [76](#)
- total_fuel_consumed_L
 - Combustion, [19](#)
 - Model, [76](#)
- type

- Combustion, [19](#)
- Renewable, [97](#)
- type_str
 - Production, [89](#)
- WAVE
 - Renewable.h, [171](#)
- Wave, [135](#)
 - __checkInputs, [138](#)
 - __computeGaussianProductionkW, [139](#)
 - __computeLookupProductionkW, [140](#)
 - __computeParaboloidProductionkW, [140](#)
 - __getGenericCapitalCost, [141](#)
 - __getGenericOpMaintCost, [141](#)
 - __writeSummary, [142](#)
 - __writeTimeSeries, [143](#)
 - ~Wave, [138](#)
 - commit, [144](#)
 - computeProductionkW, [145](#)
 - design_energy_period_s, [146](#)
 - design_significant_wave_height_m, [146](#)
 - power_model, [146](#)
 - power_model_string, [147](#)
 - Wave, [137](#)
- Wave.h
 - N_WAVE_POWER_PRODUCTION_MODELS, [174](#)
 - WAVE_POWER_GAUSSIAN, [174](#)
 - WAVE_POWER_LOOKUP, [174](#)
 - WAVE_POWER_PARABOLOID, [174](#)
 - WavePowerProductionModel, [174](#)
- WAVE_POWER_GAUSSIAN
 - Wave.h, [174](#)
- WAVE_POWER_LOOKUP
 - Wave.h, [174](#)
- WAVE_POWER_PARABOLOID
 - Wave.h, [174](#)
- WaveInputs, [147](#)
 - capital_cost, [148](#)
 - design_energy_period_s, [148](#)
 - design_significant_wave_height_m, [148](#)
 - operation_maintenance_cost_kWh, [149](#)
 - power_model, [149](#)
 - renewable_inputs, [149](#)
 - resource_key, [149](#)
- WavePowerProductionModel
 - Wave.h, [174](#)
- WIND
 - Renewable.h, [171](#)
- Wind, [150](#)
 - __checkInputs, [153](#)
 - __computeExponentialProductionkW, [154](#)
 - __computeLookupProductionkW, [154](#)
 - __getGenericCapitalCost, [155](#)
 - __getGenericOpMaintCost, [155](#)
 - __writeSummary, [155](#)
 - __writeTimeSeries, [157](#)
 - ~Wind, [153](#)
 - commit, [158](#)
 - computeProductionkW, [158](#)
 - design_speed_ms, [159](#)
 - power_model, [159](#)
 - power_model_string, [159](#)
 - Wind, [151](#), [152](#)
- Wind.h
 - N_WIND_POWER_PRODUCTION_MODELS, [176](#)
 - WIND_POWER_EXPONENTIAL, [176](#)
 - WIND_POWER_LOOKUP, [176](#)
 - WindPowerProductionModel, [176](#)
- WIND_POWER_EXPONENTIAL
 - Wind.h, [176](#)
- WIND_POWER_LOOKUP
 - Wind.h, [176](#)
- WindInputs, [160](#)
 - capital_cost, [161](#)
 - design_speed_ms, [161](#)
 - operation_maintenance_cost_kWh, [161](#)
 - power_model, [161](#)
 - renewable_inputs, [162](#)
 - resource_key, [162](#)
- WindPowerProductionModel
 - Wind.h, [176](#)
- writeResults
 - Combustion, [15](#)
 - Model, [73](#)
 - Renewable, [96](#)