

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	9
4.1 Combustion Class Reference	9
4.1.1 Detailed Description	12
4.1.2 Constructor & Destructor Documentation	12
4.1.2.1 Combustion() [1/2]	12
4.1.2.2 Combustion() [2/2]	12
4.1.2.3 ~Combustion()	14
4.1.3 Member Function Documentation	14
4.1.3.1 __checkInputs()	14
4.1.3.2 __writeSummary()	15
4.1.3.3 __writeTimeSeries()	15
4.1.3.4 commit()	15
4.1.3.5 computeEconomics()	16
4.1.3.6 computeFuelAndEmissions()	16
4.1.3.7 getEmissionskg()	17
4.1.3.8 getFuelConsumptionL()	17
4.1.3.9 handleReplacement()	18
4.1.3.10 requestProductionkW()	19
4.1.3.11 writeResults()	19
4.1.4 Member Data Documentation	20
4.1.4.1 CH4_emissions_intensity_kgL	20
4.1.4.2 CH4_emissions_vec_kg	20
4.1.4.3 CO2_emissions_intensity_kgL	20
4.1.4.4 CO2_emissions_vec_kg	20
4.1.4.5 CO_emissions_intensity_kgL	20
4.1.4.6 CO_emissions_vec_kg	21
4.1.4.7 cycle_charging_setpoint	21
4.1.4.8 fuel_consumption_vec_L	21
4.1.4.9 fuel_cost_L	21
4.1.4.10 fuel_cost_vec	21
4.1.4.11 fuel_mode	21
4.1.4.12 fuel_mode_str	22
4.1.4.13 linear_fuel_intercept_LkWh	22
4.1.4.14 linear_fuel_slope_LkWh	22

4.1.4.15 nominal_fuel_escalation_annual	22
4.1.4.16 NOx_emissions_intensity_kgL	22
4.1.4.17 NOx_emissions_vec_kg	22
4.1.4.18 PM_emissions_intensity_kgL	23
4.1.4.19 PM_emissions_vec_kg	23
4.1.4.20 real_fuel_escalation_annual	23
4.1.4.21 SOx_emissions_intensity_kgL	23
4.1.4.22 SOx_emissions_vec_kg	23
4.1.4.23 total_emissions	23
4.1.4.24 total_fuel_consumed_L	24
4.1.4.25 type	24
4.2 CombustionInputs Struct Reference	24
4.2.1 Detailed Description	25
4.2.2 Member Data Documentation	25
4.2.2.1 cycle_charging_setpoint	25
4.2.2.2 fuel_mode	25
4.2.2.3 nominal_fuel_escalation_annual	25
4.2.2.4 path_2_fuel_interp_data	25
4.2.2.5 production_inputs	25
4.3 Controller Class Reference	26
4.3.1 Detailed Description	27
4.3.2 Constructor & Destructor Documentation	27
4.3.2.1 Controller()	27
4.3.2.2 ~Controller()	27
4.3.3 Member Function Documentation	28
4.3.3.1 __applyCycleChargingControl_CHARGING()	28
4.3.3.2 __applyCycleChargingControl_DISCHARGING()	28
4.3.3.3 __applyLoadFollowingControl_CHARGING()	30
4.3.3.4 __applyLoadFollowingControl_DISCHARGING()	31
4.3.3.5 __computeNetLoad()	32
4.3.3.6 __constructCombustionMap()	33
4.3.3.7 __getRenewableProduction()	34
4.3.3.8 __handleCombustionDispatch()	36
4.3.3.9 __handleNoncombustionDispatch()	37
4.3.3.10 __handleStorageCharging() [1/2]	38
4.3.3.11 __handleStorageCharging() [2/2]	39
4.3.3.12 __handleStorageDischarging()	41
4.3.3.13 applyDispatchControl()	42
4.3.3.14 clear()	43
4.3.3.15 init()	43
4.3.3.16 setControlMode()	44
4.3.4 Member Data Documentation	44

4.3.4.1 combustion_map	44
4.3.4.2 control_mode	45
4.3.4.3 control_string	45
4.3.4.4 missed_load_vec_kW	45
4.3.4.5 net_load_vec_kW	45
4.4 Diesel Class Reference	46
4.4.1 Detailed Description	48
4.4.2 Constructor & Destructor Documentation	48
4.4.2.1 Diesel() [1/2]	48
4.4.2.2 Diesel() [2/2]	48
4.4.2.3 ~Diesel()	49
4.4.3 Member Function Documentation	49
4.4.3.1 __checkInputs()	49
4.4.3.2 __getGenericCapitalCost()	51
4.4.3.3 __getGenericFuelIntercept()	52
4.4.3.4 __getGenericFuelSlope()	52
4.4.3.5 __getGenericOpMaintCost()	53
4.4.3.6 __handleStartStop()	53
4.4.3.7 __writeSummary()	54
4.4.3.8 __writeTimeSeries()	56
4.4.3.9 commit()	57
4.4.3.10 handleReplacement()	58
4.4.3.11 requestProductionkW()	58
4.4.4 Member Data Documentation	59
4.4.4.1 minimum_load_ratio	59
4.4.4.2 minimum_runtime_hrs	59
4.4.4.3 time_since_last_start_hrs	59
4.5 DieselInputs Struct Reference	60
4.5.1 Detailed Description	61
4.5.2 Member Data Documentation	61
4.5.2.1 capital_cost	61
4.5.2.2 CH4_emissions_intensity_kgL	61
4.5.2.3 CO2_emissions_intensity_kgL	62
4.5.2.4 CO_emissions_intensity_kgL	62
4.5.2.5 combustion_inputs	62
4.5.2.6 fuel_cost_L	62
4.5.2.7 linear_fuel_intercept_LkWh	62
4.5.2.8 linear_fuel_slope_LkWh	62
4.5.2.9 minimum_load_ratio	63
4.5.2.10 minimum_runtime_hrs	63
4.5.2.11 NOx_emissions_intensity_kgL	63
4.5.2.12 operation_maintenance_cost_kWh	63

4.5.2.13 PM_emissions_intensity_kgL	63
4.5.2.14 replace_running_hrs	63
4.5.2.15 SOx_emissions_intensity_kgL	64
4.6 ElectricalLoad Class Reference	64
4.6.1 Detailed Description	65
4.6.2 Constructor & Destructor Documentation	65
4.6.2.1 ElectricalLoad() [1/2]	65
4.6.2.2 ElectricalLoad() [2/2]	65
4.6.2.3 ~ElectricalLoad()	65
4.6.3 Member Function Documentation	65
4.6.3.1 clear()	66
4.6.3.2 readLoadData()	66
4.6.4 Member Data Documentation	67
4.6.4.1 dt_vec_hrs	67
4.6.4.2 load_vec_kW	67
4.6.4.3 max_load_kW	67
4.6.4.4 mean_load_kW	68
4.6.4.5 min_load_kW	68
4.6.4.6 n_points	68
4.6.4.7 n_years	68
4.6.4.8 path_2_electrical_load_time_series	68
4.6.4.9 time_vec_hrs	68
4.7 Emissions Struct Reference	69
4.7.1 Detailed Description	69
4.7.2 Member Data Documentation	69
4.7.2.1 CH4_kg	69
4.7.2.2 CO2_kg	69
4.7.2.3 CO_kg	70
4.7.2.4 NOx_kg	70
4.7.2.5 PM_kg	70
4.7.2.6 SOx_kg	70
4.8 Hydro Class Reference	71
4.8.1 Detailed Description	73
4.8.2 Constructor & Destructor Documentation	73
4.8.2.1 Hydro() [1/2]	73
4.8.2.2 Hydro() [2/2]	74
4.8.2.3 ~Hydro()	75
4.8.3 Member Function Documentation	75
4.8.3.1 __checkInputs()	75
4.8.3.2 __flowToPower()	76
4.8.3.3 __getAcceptableFlow()	76
4.8.3.4 __getAvailableFlow()	77

4.8.3.5	__getEfficiencyFactor()	77
4.8.3.6	__getGenericCapitalCost()	78
4.8.3.7	__getGenericOpMaintCost()	79
4.8.3.8	__getMaximumFlowm3hr()	79
4.8.3.9	__getMinimumFlowm3hr()	79
4.8.3.10	__initInterpolator()	80
4.8.3.11	__powerToFlow()	81
4.8.3.12	__updateState()	82
4.8.3.13	__writeSummary()	83
4.8.3.14	__writeTimeSeries()	85
4.8.3.15	commit()	85
4.8.3.16	handleReplacement()	86
4.8.3.17	requestProductionkW()	86
4.8.4	Member Data Documentation	87
4.8.4.1	fluid_density_kgm3	88
4.8.4.2	init_reservoir_state	88
4.8.4.3	maximum_flow_m3hr	88
4.8.4.4	minimum_flow_m3hr	88
4.8.4.5	minimum_power_kW	88
4.8.4.6	net_head_m	88
4.8.4.7	reservoir_capacity_m3	89
4.8.4.8	spill_rate_vec_m3hr	89
4.8.4.9	stored_volume_m3	89
4.8.4.10	stored_volume_vec_m3	89
4.8.4.11	turbine_flow_vec_m3hr	89
4.8.4.12	turbine_type	89
4.9	HydroInputs Struct Reference	90
4.9.1	Detailed Description	91
4.9.2	Member Data Documentation	91
4.9.2.1	capital_cost	91
4.9.2.2	fluid_density_kgm3	91
4.9.2.3	init_reservoir_state	91
4.9.2.4	net_head_m	91
4.9.2.5	noncombustion_inputs	91
4.9.2.6	operation_maintenance_cost_kWh	92
4.9.2.7	reservoir_capacity_m3	92
4.9.2.8	resource_key	92
4.9.2.9	turbine_type	92
4.10	Interpolator Class Reference	92
4.10.1	Detailed Description	94
4.10.2	Constructor & Destructor Documentation	94
4.10.2.1	Interpolator()	94

4.10.2.2 ~Interpolator()	94
4.10.3 Member Function Documentation	94
4.10.3.1 __checkBounds1D()	94
4.10.3.2 __checkBounds2D()	95
4.10.3.3 __checkDataKey1D()	96
4.10.3.4 __checkDataKey2D()	97
4.10.3.5 __getDataStringMatrix()	97
4.10.3.6 __getInterpolationIndex()	98
4.10.3.7 __isNonNumeric()	98
4.10.3.8 __readData1D()	99
4.10.3.9 __readData2D()	100
4.10.3.10 __splitCommaSeparatedString()	101
4.10.3.11 __throwReadError()	102
4.10.3.12 addData1D()	102
4.10.3.13 addData2D()	103
4.10.3.14 interp1D()	103
4.10.3.15 interp2D()	104
4.10.4 Member Data Documentation	105
4.10.4.1 interp_map_1D	105
4.10.4.2 interp_map_2D	105
4.10.4.3 path_map_1D	105
4.10.4.4 path_map_2D	105
4.11 InterpolatorStruct1D Struct Reference	106
4.11.1 Detailed Description	106
4.11.2 Member Data Documentation	106
4.11.2.1 max_x	106
4.11.2.2 min_x	106
4.11.2.3 n_points	107
4.11.2.4 x_vec	107
4.11.2.5 y_vec	107
4.12 InterpolatorStruct2D Struct Reference	107
4.12.1 Detailed Description	108
4.12.2 Member Data Documentation	108
4.12.2.1 max_x	108
4.12.2.2 max_y	108
4.12.2.3 min_x	108
4.12.2.4 min_y	108
4.12.2.5 n_cols	108
4.12.2.6 n_rows	109
4.12.2.7 x_vec	109
4.12.2.8 y_vec	109
4.12.2.9 z_matrix	109

4.13 Lilon Class Reference	110
4.13.1 Detailed Description	112
4.13.2 Constructor & Destructor Documentation	112
4.13.2.1 Lilon() [1/2]	112
4.13.2.2 Lilon() [2/2]	113
4.13.2.3 ~Lilon()	114
4.13.3 Member Function Documentation	114
4.13.3.1 __checkInputs()	114
4.13.3.2 __getBcal()	116
4.13.3.3 __getEacal()	117
4.13.3.4 __getGenericCapitalCost()	117
4.13.3.5 __getGenericOpMaintCost()	118
4.13.3.6 __handleDegradation()	118
4.13.3.7 __modelDegradation()	119
4.13.3.8 __toggleDepleted()	119
4.13.3.9 __writeSummary()	120
4.13.3.10 __writeTimeSeries()	121
4.13.3.11 commitCharge()	122
4.13.3.12 commitDischarge()	123
4.13.3.13 getAcceptablekW()	123
4.13.3.14 getAvailablekW()	124
4.13.3.15 handleReplacement()	125
4.13.4 Member Data Documentation	125
4.13.4.1 charging_efficiency	125
4.13.4.2 degradation_a_cal	126
4.13.4.3 degradation_alpha	126
4.13.4.4 degradation_B_hat_cal_0	126
4.13.4.5 degradation_beta	126
4.13.4.6 degradation_Ea_cal_0	126
4.13.4.7 degradation_r_cal	126
4.13.4.8 degradation_s_cal	127
4.13.4.9 discharging_efficiency	127
4.13.4.10 dynamic_energy_capacity_kWh	127
4.13.4.11 dynamic_power_capacity_kW	127
4.13.4.12 gas_constant_JmolK	127
4.13.4.13 hysteresis_SOC	127
4.13.4.14 init_SOC	128
4.13.4.15 max_SOC	128
4.13.4.16 min_SOC	128
4.13.4.17 power_degradation_flag	128
4.13.4.18 replace_SOH	128
4.13.4.19 SOH	128

4.13.4.20 SOH_vec	129
4.13.4.21 temperature_K	129
4.14 LilonInputs Struct Reference	129
4.14.1 Detailed Description	130
4.14.2 Member Data Documentation	131
4.14.2.1 capital_cost	131
4.14.2.2 charging_efficiency	131
4.14.2.3 degradation_a_cal	131
4.14.2.4 degradation_alpha	131
4.14.2.5 degradation_B_hat_cal_0	131
4.14.2.6 degradation_beta	132
4.14.2.7 degradation_Ea_cal_0	132
4.14.2.8 degradation_r_cal	132
4.14.2.9 degradation_s_cal	132
4.14.2.10 discharging_efficiency	132
4.14.2.11 gas_constant_JmolK	132
4.14.2.12 hysteresis_SOC	133
4.14.2.13 init_SOC	133
4.14.2.14 max_SOC	133
4.14.2.15 min_SOC	133
4.14.2.16 operation_maintenance_cost_kWh	133
4.14.2.17 power_degradation_flag	133
4.14.2.18 replace_SOH	134
4.14.2.19 storage_inputs	134
4.14.2.20 temperature_K	134
4.15 Model Class Reference	134
4.15.1 Detailed Description	136
4.15.2 Constructor & Destructor Documentation	136
4.15.2.1 Model() [1/2]	137
4.15.2.2 Model() [2/2]	137
4.15.2.3 ~Model()	137
4.15.3 Member Function Documentation	137
4.15.3.1 __checkInputs()	138
4.15.3.2 __computeEconomics()	138
4.15.3.3 __computeFuelAndEmissions()	138
4.15.3.4 __computeLevellizedCostOfEnergy()	139
4.15.3.5 __computeNetPresentCost()	139
4.15.3.6 __writeSummary()	140
4.15.3.7 __writeTimeSeries()	143
4.15.3.8 addDiesel()	144
4.15.3.9 addHydro()	144
4.15.3.10 addLilon()	145

4.15.3.11 addResource() [1/2]	145
4.15.3.12 addResource() [2/2]	146
4.15.3.13 addSolar()	146
4.15.3.14 addTidal()	147
4.15.3.15 addWave()	147
4.15.3.16 addWind()	147
4.15.3.17 clear()	148
4.15.3.18 reset()	148
4.15.3.19 run()	149
4.15.3.20 writeResults()	149
4.15.4 Member Data Documentation	151
4.15.4.1 combustion_ptr_vec	151
4.15.4.2 controller	151
4.15.4.3 electrical_load	151
4.15.4.4 levlized_cost_of_energy_kWh	151
4.15.4.5 net_present_cost	151
4.15.4.6 noncombustion_ptr_vec	152
4.15.4.7 renewable_ptr_vec	152
4.15.4.8 resources	152
4.15.4.9 storage_ptr_vec	152
4.15.4.10 total_dispatch_discharge_kWh	152
4.15.4.11 total_emissions	152
4.15.4.12 total_fuel_consumed_L	153
4.15.4.13 total_renewable_dispatch_kWh	153
4.16 ModellInputs Struct Reference	153
4.16.1 Detailed Description	153
4.16.2 Member Data Documentation	153
4.16.2.1 control_mode	154
4.16.2.2 path_2_electrical_load_time_series	154
4.17 Noncombustion Class Reference	154
4.17.1 Detailed Description	156
4.17.2 Constructor & Destructor Documentation	156
4.17.2.1 Noncombustion() [1/2]	156
4.17.2.2 Noncombustion() [2/2]	156
4.17.2.3 ~Noncombustion()	157
4.17.3 Member Function Documentation	157
4.17.3.1 __checkInputs()	157
4.17.3.2 __handleStartStop()	157
4.17.3.3 __writeSummary()	158
4.17.3.4 __writeTimeSeries()	158
4.17.3.5 commit() [1/2]	158
4.17.3.6 commit() [2/2]	159

4.17.3.7 computeEconomics()	159
4.17.3.8 handleReplacement()	160
4.17.3.9 requestProductionkW() [1/2]	160
4.17.3.10 requestProductionkW() [2/2]	160
4.17.3.11 writeResults()	161
4.17.4 Member Data Documentation	161
4.17.4.1 resource_key	161
4.17.4.2 type	162
4.18 NoncombustionInputs Struct Reference	162
4.18.1 Detailed Description	162
4.18.2 Member Data Documentation	162
4.18.2.1 production_inputs	163
4.19 Production Class Reference	163
4.19.1 Detailed Description	166
4.19.2 Constructor & Destructor Documentation	166
4.19.2.1 Production() [1/2]	166
4.19.2.2 Production() [2/2]	166
4.19.2.3 ~Production()	167
4.19.3 Member Function Documentation	167
4.19.3.1 __checkInputs()	168
4.19.3.2 __checkNormalizedProduction()	168
4.19.3.3 __checkTimePoint()	169
4.19.3.4 __readNormalizedProductionData()	169
4.19.3.5 __throwLengthError()	170
4.19.3.6 commit()	171
4.19.3.7 computeEconomics()	172
4.19.3.8 computeRealDiscountAnnual()	172
4.19.3.9 getProductionkW()	173
4.19.3.10 handleReplacement()	173
4.19.4 Member Data Documentation	174
4.19.4.1 capacity_kW	174
4.19.4.2 capital_cost	174
4.19.4.3 capital_cost_vec	174
4.19.4.4 curtailment_vec_kW	174
4.19.4.5 dispatch_vec_kW	175
4.19.4.6 interpolator	175
4.19.4.7 is_running	175
4.19.4.8 is_running_vec	175
4.19.4.9 is_sunk	175
4.19.4.10 levlized_cost_of_energy_kWh	175
4.19.4.11 n_points	176
4.19.4.12 n_replacements	176

4.19.4.13 n_starts	176
4.19.4.14 n_years	176
4.19.4.15 net_present_cost	176
4.19.4.16 nominal_discount_annual	176
4.19.4.17 nominal_inflation_annual	177
4.19.4.18 normalized_production_series_given	177
4.19.4.19 normalized_production_vec	177
4.19.4.20 operation_maintenance_cost_kWh	177
4.19.4.21 operation_maintenance_cost_vec	177
4.19.4.22 path_2_normalized_production_time_series	177
4.19.4.23 print_flag	178
4.19.4.24 production_vec_kW	178
4.19.4.25 real_discount_annual	178
4.19.4.26 replace_running_hrs	178
4.19.4.27 running_hours	178
4.19.4.28 storage_vec_kW	178
4.19.4.29 total_dispatch_kWh	179
4.19.4.30 type_str	179
4.20 ProductionInputs Struct Reference	179
4.20.1 Detailed Description	179
4.20.2 Member Data Documentation	180
4.20.2.1 capacity_kW	180
4.20.2.2 is_sunk	180
4.20.2.3 nominal_discount_annual	180
4.20.2.4 nominal_inflation_annual	180
4.20.2.5 path_2_normalized_production_time_series	180
4.20.2.6 print_flag	181
4.20.2.7 replace_running_hrs	181
4.21 Renewable Class Reference	181
4.21.1 Detailed Description	183
4.21.2 Constructor & Destructor Documentation	183
4.21.2.1 Renewable() [1/2]	183
4.21.2.2 Renewable() [2/2]	183
4.21.2.3 ~Renewable()	184
4.21.3 Member Function Documentation	184
4.21.3.1 __checkInputs()	184
4.21.3.2 __handleStartStop()	185
4.21.3.3 __writeSummary()	185
4.21.3.4 __writeTimeSeries()	185
4.21.3.5 commit()	185
4.21.3.6 computeEconomics()	186
4.21.3.7 computeProductionkW() [1/2]	186

4.21.3.8 computeProductionkW() [2/2]	187
4.21.3.9 handleReplacement()	187
4.21.3.10 writeResults()	187
4.21.4 Member Data Documentation	188
4.21.4.1 resource_key	188
4.21.4.2 type	189
4.22 RenewableInputs Struct Reference	189
4.22.1 Detailed Description	189
4.22.2 Member Data Documentation	189
4.22.2.1 production_inputs	190
4.23 Resources Class Reference	190
4.23.1 Detailed Description	191
4.23.2 Constructor & Destructor Documentation	191
4.23.2.1 Resources()	191
4.23.2.2 ~Resources()	191
4.23.3 Member Function Documentation	192
4.23.3.1 __checkResourceKey1D() [1/2]	192
4.23.3.2 __checkResourceKey1D() [2/2]	192
4.23.3.3 __checkResourceKey2D()	193
4.23.3.4 __checkTimePoint()	194
4.23.3.5 __readHydroResource()	194
4.23.3.6 __readSolarResource()	195
4.23.3.7 __readTidalResource()	196
4.23.3.8 __readWaveResource()	197
4.23.3.9 __readWindResource()	198
4.23.3.10 __throwLengthError()	199
4.23.3.11 addResource() [1/2]	200
4.23.3.12 addResource() [2/2]	201
4.23.3.13 clear()	202
4.23.4 Member Data Documentation	202
4.23.4.1 path_map_1D	202
4.23.4.2 path_map_2D	202
4.23.4.3 resource_map_1D	203
4.23.4.4 resource_map_2D	203
4.23.4.5 string_map_1D	203
4.23.4.6 string_map_2D	203
4.24 Solar Class Reference	204
4.24.1 Detailed Description	205
4.24.2 Constructor & Destructor Documentation	205
4.24.2.1 Solar() [1/2]	206
4.24.2.2 Solar() [2/2]	206
4.24.2.3 ~Solar()	207

4.24.3 Member Function Documentation	207
4.24.3.1 __checkInputs()	207
4.24.3.2 __getGenericCapitalCost()	208
4.24.3.3 __getGenericOpMaintCost()	208
4.24.3.4 __writeSummary()	208
4.24.3.5 __writeTimeSeries()	210
4.24.3.6 commit()	210
4.24.3.7 computeProductionkW()	211
4.24.3.8 handleReplacement()	212
4.24.4 Member Data Documentation	212
4.24.4.1 derating	212
4.25 SolarInputs Struct Reference	213
4.25.1 Detailed Description	213
4.25.2 Member Data Documentation	214
4.25.2.1 capital_cost	214
4.25.2.2 derating	214
4.25.2.3 operation_maintenance_cost_kWh	214
4.25.2.4 renewable_inputs	214
4.25.2.5 resource_key	214
4.26 Storage Class Reference	215
4.26.1 Detailed Description	217
4.26.2 Constructor & Destructor Documentation	217
4.26.2.1 Storage() [1/2]	217
4.26.2.2 Storage() [2/2]	217
4.26.2.3 ~Storage()	218
4.26.3 Member Function Documentation	218
4.26.3.1 __checkInputs()	219
4.26.3.2 __computeRealDiscountAnnual()	219
4.26.3.3 __writeSummary()	220
4.26.3.4 __writeTimeSeries()	220
4.26.3.5 commitCharge()	220
4.26.3.6 commitDischarge()	221
4.26.3.7 computeEconomics()	221
4.26.3.8 getAcceptablekW()	222
4.26.3.9 getAvailablekW()	222
4.26.3.10 handleReplacement()	222
4.26.3.11 writeResults()	223
4.26.4 Member Data Documentation	223
4.26.4.1 capital_cost	223
4.26.4.2 capital_cost_vec	224
4.26.4.3 charge_kWh	224
4.26.4.4 charge_vec_kWh	224

4.26.4.5 charging_power_vec_kW	224
4.26.4.6 discharging_power_vec_kW	224
4.26.4.7 energy_capacity_kWh	225
4.26.4.8 interpolator	225
4.26.4.9 is_depleted	225
4.26.4.10 is_sunk	225
4.26.4.11 levlized_cost_of_energy_kWh	225
4.26.4.12 n_points	225
4.26.4.13 n_replacements	226
4.26.4.14 n_years	226
4.26.4.15 net_present_cost	226
4.26.4.16 nominal_discount_annual	226
4.26.4.17 nominal_inflation_annual	226
4.26.4.18 operation_maintenance_cost_kWh	226
4.26.4.19 operation_maintenance_cost_vec	227
4.26.4.20 power_capacity_kW	227
4.26.4.21 power_kW	227
4.26.4.22 print_flag	227
4.26.4.23 real_discount_annual	227
4.26.4.24 total_discharge_kWh	227
4.26.4.25 type	228
4.26.4.26 type_str	228
4.27 StorageInputs Struct Reference	228
4.27.1 Detailed Description	228
4.27.2 Member Data Documentation	229
4.27.2.1 energy_capacity_kWh	229
4.27.2.2 is_sunk	229
4.27.2.3 nominal_discount_annual	229
4.27.2.4 nominal_inflation_annual	229
4.27.2.5 power_capacity_kW	229
4.27.2.6 print_flag	230
4.28 Tidal Class Reference	230
4.28.1 Detailed Description	232
4.28.2 Constructor & Destructor Documentation	232
4.28.2.1 Tidal() [1/2]	232
4.28.2.2 Tidal() [2/2]	232
4.28.2.3 ~Tidal()	234
4.28.3 Member Function Documentation	234
4.28.3.1 __checkInputs()	234
4.28.3.2 __computeCubicProductionkW()	235
4.28.3.3 __computeExponentialProductionkW()	235
4.28.3.4 __computeLookupProductionkW()	236

4.28.3.5 __getGenericCapitalCost()	237
4.28.3.6 __getGenericOpMaintCost()	237
4.28.3.7 __writeSummary()	237
4.28.3.8 __writeTimeSeries()	239
4.28.3.9 commit()	240
4.28.3.10 computeProductionkW()	240
4.28.3.11 handleReplacement()	241
4.28.4 Member Data Documentation	242
4.28.4.1 design_speed_ms	242
4.28.4.2 power_model	242
4.28.4.3 power_model_string	242
4.29 TidalInputs Struct Reference	243
4.29.1 Detailed Description	243
4.29.2 Member Data Documentation	244
4.29.2.1 capital_cost	244
4.29.2.2 design_speed_ms	244
4.29.2.3 operation_maintenance_cost_kWh	244
4.29.2.4 power_model	244
4.29.2.5 renewable_inputs	244
4.29.2.6 resource_key	245
4.30 Wave Class Reference	245
4.30.1 Detailed Description	247
4.30.2 Constructor & Destructor Documentation	247
4.30.2.1 Wave() [1/2]	247
4.30.2.2 Wave() [2/2]	247
4.30.2.3 ~Wave()	249
4.30.3 Member Function Documentation	249
4.30.3.1 __checkInputs()	249
4.30.3.2 __computeGaussianProductionkW()	250
4.30.3.3 __computeLookupProductionkW()	251
4.30.3.4 __computeParaboloidProductionkW()	251
4.30.3.5 __getGenericCapitalCost()	252
4.30.3.6 __getGenericOpMaintCost()	252
4.30.3.7 __writeSummary()	253
4.30.3.8 __writeTimeSeries()	254
4.30.3.9 commit()	255
4.30.3.10 computeProductionkW()	256
4.30.3.11 handleReplacement()	257
4.30.4 Member Data Documentation	257
4.30.4.1 design_energy_period_s	258
4.30.4.2 design_significant_wave_height_m	258
4.30.4.3 power_model	258

4.30.4.4 power_model_string	258
4.31 WaveInputs Struct Reference	259
4.31.1 Detailed Description	260
4.31.2 Member Data Documentation	260
4.31.2.1 capital_cost	260
4.31.2.2 design_energy_period_s	260
4.31.2.3 design_significant_wave_height_m	260
4.31.2.4 operation_maintenance_cost_kWh	260
4.31.2.5 path_2_normalized_performance_matrix	261
4.31.2.6 power_model	261
4.31.2.7 renewable_inputs	261
4.31.2.8 resource_key	261
4.32 Wind Class Reference	262
4.32.1 Detailed Description	263
4.32.2 Constructor & Destructor Documentation	264
4.32.2.1 Wind() [1/2]	264
4.32.2.2 Wind() [2/2]	264
4.32.2.3 ~Wind()	265
4.32.3 Member Function Documentation	265
4.32.3.1 __checkInputs()	266
4.32.3.2 __computeCubicProductionkW()	266
4.32.3.3 __computeExponentialProductionkW()	267
4.32.3.4 __computeLookupProductionkW()	268
4.32.3.5 __getGenericCapitalCost()	268
4.32.3.6 __getGenericOpMaintCost()	269
4.32.3.7 __writeSummary()	269
4.32.3.8 __writeTimeSeries()	270
4.32.3.9 commit()	271
4.32.3.10 computeProductionkW()	272
4.32.3.11 handleReplacement()	273
4.32.4 Member Data Documentation	274
4.32.4.1 design_speed_ms	274
4.32.4.2 power_model	274
4.32.4.3 power_model_string	274
4.33 WindInputs Struct Reference	275
4.33.1 Detailed Description	275
4.33.2 Member Data Documentation	276
4.33.2.1 capital_cost	276
4.33.2.2 design_speed_ms	276
4.33.2.3 operation_maintenance_cost_kWh	276
4.33.2.4 power_model	276
4.33.2.5 renewable_inputs	276

4.33.2.6 resource_key	276
5 File Documentation	277
5.1 header/Controller.h File Reference	277
5.1.1 Detailed Description	278
5.1.2 Enumeration Type Documentation	278
5.1.2.1 ControlMode	278
5.2 header/doxygen_cite.h File Reference	278
5.2.1 Detailed Description	278
5.3 header/ElectricalLoad.h File Reference	279
5.3.1 Detailed Description	279
5.4 header/Interpolator.h File Reference	279
5.4.1 Detailed Description	280
5.5 header/Model.h File Reference	280
5.5.1 Detailed Description	281
5.6 header/Production/Combustion/Combustion.h File Reference	281
5.6.1 Detailed Description	282
5.6.2 Enumeration Type Documentation	282
5.6.2.1 CombustionType	282
5.6.2.2 FuelMode	282
5.7 header/Production/Combustion/Diesel.h File Reference	284
5.7.1 Detailed Description	285
5.8 header/Production/Noncombustion/Hydro.h File Reference	285
5.8.1 Detailed Description	286
5.8.2 Enumeration Type Documentation	286
5.8.2.1 HydroInterpKeys	286
5.8.2.2 HydroTurbineType	286
5.9 header/Production/Noncombustion/Noncombustion.h File Reference	287
5.9.1 Enumeration Type Documentation	288
5.9.1.1 NoncombustionType	288
5.10 header/Production/Production.h File Reference	288
5.10.1 Detailed Description	289
5.11 header/Production/Renewable/Renewable.h File Reference	289
5.11.1 Detailed Description	290
5.11.2 Enumeration Type Documentation	290
5.11.2.1 RenewableType	290
5.12 header/Production/Renewable/Solar.h File Reference	290
5.12.1 Detailed Description	291
5.13 header/Production/Renewable/Tidal.h File Reference	291
5.13.1 Detailed Description	292
5.13.2 Enumeration Type Documentation	292
5.13.2.1 TidalPowerProductionModel	292

5.14 header/Production/Renewable/Wave.h File Reference	293
5.14.1 Detailed Description	294
5.14.2 Enumeration Type Documentation	294
5.14.2.1 WavePowerProductionModel	294
5.15 header/Production/Renewable/Wind.h File Reference	294
5.15.1 Detailed Description	295
5.15.2 Enumeration Type Documentation	295
5.15.2.1 WindPowerProductionModel	295
5.16 header/Resources.h File Reference	296
5.16.1 Detailed Description	296
5.17 header/std_includes.h File Reference	297
5.17.1 Detailed Description	297
5.18 header/Storage/Lilon.h File Reference	297
5.18.1 Detailed Description	298
5.19 header/Storage/Storage.h File Reference	298
5.19.1 Detailed Description	299
5.19.2 Enumeration Type Documentation	299
5.19.2.1 StorageType	299
5.20 projects/example.cpp File Reference	300
5.20.1 Function Documentation	300
5.20.1.1 main()	300
5.21 pybindings/PYBIND11_PGM.cpp File Reference	304
5.21.1 Detailed Description	305
5.21.2 Function Documentation	305
5.21.2.1 PYBIND11_MODULE()	305
5.22 pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp File Reference	306
5.22.1 Detailed Description	306
5.22.2 Function Documentation	307
5.22.2.1 def()	307
5.22.2.2 def_readwrite() [1/4]	307
5.22.2.3 def_readwrite() [2/4]	307
5.22.2.4 def_readwrite() [3/4]	307
5.22.2.5 def_readwrite() [4/4]	307
5.22.2.6 value() [1/2]	308
5.22.2.7 value() [2/2]	308
5.22.3 Variable Documentation	308
5.22.3.1 def_readwrite	308
5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference	308
5.23.1 Detailed Description	309
5.23.2 Function Documentation	309
5.23.2.1 def()	309
5.23.2.2 def_readwrite() [1/8]	310

5.23.2.3 def_readwrite() [2/8]	310
5.23.2.4 def_readwrite() [3/8]	310
5.23.2.5 def_readwrite() [4/8]	310
5.23.2.6 def_readwrite() [5/8]	310
5.23.2.7 def_readwrite() [6/8]	311
5.23.2.8 def_readwrite() [7/8]	311
5.23.2.9 def_readwrite() [8/8]	311
5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp File Reference	311
5.24.1 Detailed Description	312
5.24.2 Function Documentation	312
5.24.2.1 def()	312
5.24.2.2 def_readwrite() [1/9]	313
5.24.2.3 def_readwrite() [2/9]	313
5.24.2.4 def_readwrite() [3/9]	313
5.24.2.5 def_readwrite() [4/9]	313
5.24.2.6 def_readwrite() [5/9]	313
5.24.2.7 def_readwrite() [6/9]	313
5.24.2.8 def_readwrite() [7/9]	314
5.24.2.9 def_readwrite() [8/9]	314
5.24.2.10 def_readwrite() [9/9]	314
5.24.2.11 value() [1/2]	314
5.24.2.12 value() [2/2]	314
5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp File Reference	315
5.25.1 Detailed Description	315
5.25.2 Function Documentation	315
5.25.2.1 def()	315
5.25.2.2 value()	316
5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference	316
5.26.1 Detailed Description	317
5.26.2 Function Documentation	317
5.26.2.1 def()	318
5.26.2.2 def_readwrite() [1/17]	318
5.26.2.3 def_readwrite() [2/17]	318
5.26.2.4 def_readwrite() [3/17]	318
5.26.2.5 def_readwrite() [4/17]	318
5.26.2.6 def_readwrite() [5/17]	319
5.26.2.7 def_readwrite() [6/17]	319
5.26.2.8 def_readwrite() [7/17]	319
5.26.2.9 def_readwrite() [8/17]	319
5.26.2.10 def_readwrite() [9/17]	319
5.26.2.11 def_readwrite() [10/17]	320
5.26.2.12 def_readwrite() [11/17]	320

5.26.2.13 def_readwrite() [12/17]	320
5.26.2.14 def_readwrite() [13/17]	320
5.26.2.15 def_readwrite() [14/17]	320
5.26.2.16 def_readwrite() [15/17]	321
5.26.2.17 def_readwrite() [16/17]	321
5.26.2.18 def_readwrite() [17/17]	321
5.27 pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp File Reference	321
5.27.1 Detailed Description	322
5.27.2 Function Documentation	322
5.27.2.1 def()	322
5.27.2.2 value() [1/2]	322
5.27.2.3 value() [2/2]	322
5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference	323
5.28.1 Detailed Description	323
5.28.2 Function Documentation	323
5.28.2.1 def()	323
5.28.2.2 def_readwrite() [1/2]	324
5.28.2.3 def_readwrite() [2/2]	324
5.29 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference	324
5.29.1 Detailed Description	325
5.29.2 Function Documentation	325
5.29.2.1 def_readwrite() [1/2]	325
5.29.2.2 def_readwrite() [2/2]	325
5.29.2.3 value() [1/2]	325
5.29.2.4 value() [2/2]	325
5.29.3 Variable Documentation	326
5.29.3.1 def_readwrite	326
5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference	326
5.30.1 Detailed Description	327
5.30.2 Function Documentation	327
5.30.2.1 def_readwrite() [1/3]	327
5.30.2.2 def_readwrite() [2/3]	327
5.30.2.3 def_readwrite() [3/3]	327
5.30.2.4 value() [1/2]	328
5.30.2.5 value() [2/2]	328
5.30.3 Variable Documentation	328
5.30.3.1 def_readwrite	328
5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference	328
5.31.1 Detailed Description	329
5.31.2 Function Documentation	329
5.31.2.1 def_readwrite() [1/2]	329
5.31.2.2 def_readwrite() [2/2]	329

5.31.2.3 value()	329
5.31.3 Variable Documentation	330
5.31.3.1 def_readwrite	330
5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference	330
5.32.1 Detailed Description	331
5.32.2 Function Documentation	331
5.32.2.1 def() [1/3]	331
5.32.2.2 def() [2/3]	331
5.32.2.3 def() [3/3]	331
5.32.2.4 def_readwrite() [1/2]	331
5.32.2.5 def_readwrite() [2/2]	332
5.32.2.6 value()	332
5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference	332
5.33.1 Detailed Description	333
5.33.2 Function Documentation	333
5.33.2.1 def_readwrite() [1/4]	333
5.33.2.2 def_readwrite() [2/4]	333
5.33.2.3 def_readwrite() [3/4]	333
5.33.2.4 def_readwrite() [4/4]	333
5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference	334
5.34.1 Detailed Description	334
5.34.2 Function Documentation	334
5.34.2.1 def()	335
5.34.2.2 def_readwrite() [1/7]	335
5.34.2.3 def_readwrite() [2/7]	335
5.34.2.4 def_readwrite() [3/7]	335
5.34.2.5 def_readwrite() [4/7]	335
5.34.2.6 def_readwrite() [5/7]	335
5.34.2.7 def_readwrite() [6/7]	336
5.34.2.8 def_readwrite() [7/7]	336
5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference	336
5.35.1 Detailed Description	336
5.35.2 Variable Documentation	337
5.35.2.1 def_readwrite	337
5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference	337
5.36.1 Detailed Description	337
5.36.2 Function Documentation	338
5.36.2.1 def_readwrite() [1/2]	338
5.36.2.2 def_readwrite() [2/2]	338
5.37 pybindings/snippets/Storage/PYBIND11_Lilon.cpp File Reference	338
5.37.1 Detailed Description	339
5.37.2 Function Documentation	339

5.37.2.1 def_readwrite() [1/9]	339
5.37.2.2 def_readwrite() [2/9]	340
5.37.2.3 def_readwrite() [3/9]	340
5.37.2.4 def_readwrite() [4/9]	340
5.37.2.5 def_readwrite() [5/9]	340
5.37.2.6 def_readwrite() [6/9]	340
5.37.2.7 def_readwrite() [7/9]	341
5.37.2.8 def_readwrite() [8/9]	341
5.37.2.9 def_readwrite() [9/9]	341
5.37.3 Variable Documentation	341
5.37.3.1 def_readwrite	341
5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference	342
5.38.1 Detailed Description	342
5.38.2 Function Documentation	342
5.38.2.1 def_readwrite() [1/2]	343
5.38.2.2 def_readwrite() [2/2]	343
5.38.2.3 value()	343
5.38.3 Variable Documentation	343
5.38.3.1 def_readwrite	343
5.39 source/Controller.cpp File Reference	344
5.39.1 Detailed Description	344
5.40 source/ElectricalLoad.cpp File Reference	344
5.40.1 Detailed Description	344
5.41 source/Interpolator.cpp File Reference	345
5.41.1 Detailed Description	345
5.42 source/Model.cpp File Reference	345
5.42.1 Detailed Description	345
5.43 source/Production/Combustion/Combustion.cpp File Reference	346
5.43.1 Detailed Description	346
5.44 source/Production/Combustion/Diesel.cpp File Reference	346
5.44.1 Detailed Description	347
5.45 source/Production/Noncombustion/Hydro.cpp File Reference	347
5.45.1 Detailed Description	347
5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference	347
5.46.1 Detailed Description	348
5.47 source/Production/Production.cpp File Reference	348
5.47.1 Detailed Description	348
5.48 source/Production/Renewable/Renewable.cpp File Reference	349
5.48.1 Detailed Description	349
5.49 source/Production/Renewable/Solar.cpp File Reference	349
5.49.1 Detailed Description	350
5.50 source/Production/Renewable/Tidal.cpp File Reference	350

5.50.1 Detailed Description	350
5.51 source/Production/Renewable/Wave.cpp File Reference	350
5.51.1 Detailed Description	351
5.52 source/Production/Renewable/Wind.cpp File Reference	351
5.52.1 Detailed Description	351
5.53 source/Resources.cpp File Reference	352
5.53.1 Detailed Description	352
5.54 source/Storage/Lilon.cpp File Reference	352
5.54.1 Detailed Description	353
5.55 source/Storage/Storage.cpp File Reference	353
5.55.1 Detailed Description	353
5.56 test/source/Production/Combustion/test_Combustion.cpp File Reference	353
5.56.1 Detailed Description	354
5.56.2 Function Documentation	354
5.56.2.1 main()	354
5.56.2.2 testConstruct_Combustion()	354
5.57 test/source/Production/Combustion/test_Diesel.cpp File Reference	356
5.57.1 Detailed Description	357
5.57.2 Function Documentation	357
5.57.2.1 main()	357
5.57.2.2 testBadConstruct_Diesel()	358
5.57.2.3 testCapacityConstraint_Diesel()	358
5.57.2.4 testCommit_Diesel()	358
5.57.2.5 testConstruct_Diesel()	360
5.57.2.6 testConstructLookup_Diesel()	361
5.57.2.7 testEconomics_Diesel()	362
5.57.2.8 testFuelConsumptionEmissions_Diesel()	362
5.57.2.9 testFuelLookup_Diesel()	364
5.57.2.10 testMinimumLoadRatioConstraint_Diesel()	365
5.57.2.11 testMinimumRuntimeConstraint_Diesel()	366
5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference	366
5.58.1 Detailed Description	367
5.58.2 Function Documentation	367
5.58.2.1 main()	367
5.58.2.2 testCommit_Hydro()	368
5.58.2.3 testConstruct_Hydro()	369
5.58.2.4 testEfficiencyInterpolation_Hydro()	370
5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference	372
5.59.1 Detailed Description	372
5.59.2 Function Documentation	372
5.59.2.1 main()	373
5.59.2.2 testConstruct_Noncombustion()	373

5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference	374
5.60.1 Detailed Description	374
5.60.2 Function Documentation	375
5.60.2.1 main()	375
5.60.2.2 testConstruct_Renewable()	375
5.61 test/source/Production/Renewable/test_Solar.cpp File Reference	376
5.61.1 Detailed Description	377
5.61.2 Function Documentation	377
5.61.2.1 main()	377
5.61.2.2 testBadConstruct_Solar()	378
5.61.2.3 testCommit_Solar()	378
5.61.2.4 testConstruct_Solar()	380
5.61.2.5 testEconomics_Solar()	381
5.61.2.6 testProductionConstraint_Solar()	381
5.61.2.7 testProductionOverride_Solar()	382
5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference	383
5.62.1 Detailed Description	384
5.62.2 Function Documentation	384
5.62.2.1 main()	384
5.62.2.2 testBadConstruct_Tidal()	384
5.62.2.3 testCommit_Tidal()	385
5.62.2.4 testConstruct_Tidal()	386
5.62.2.5 testEconomics_Tidal()	387
5.62.2.6 testProductionConstraint_Tidal()	388
5.63 test/source/Production/Renewable/test_Wave.cpp File Reference	388
5.63.1 Detailed Description	389
5.63.2 Function Documentation	389
5.63.2.1 main()	389
5.63.2.2 testBadConstruct_Wave()	390
5.63.2.3 testCommit_Wave()	390
5.63.2.4 testConstruct_Wave()	392
5.63.2.5 testConstructLookup_Wave()	393
5.63.2.6 testEconomics_Wave()	393
5.63.2.7 testProductionConstraint_Wave()	394
5.63.2.8 testProductionLookup_Wave()	394
5.64 test/source/Production/Renewable/test_Wind.cpp File Reference	395
5.64.1 Detailed Description	396
5.64.2 Function Documentation	396
5.64.2.1 main()	397
5.64.2.2 testBadConstruct_Wind()	397
5.64.2.3 testCommit_Wind()	398
5.64.2.4 testConstruct_Wind()	399

5.64.2.5 testEconomics_Wind()	400
5.64.2.6 testProductionConstraint_Wind()	400
5.65 test/source/Production/test_Production.cpp File Reference	401
5.65.1 Detailed Description	402
5.65.2 Function Documentation	402
5.65.2.1 main()	402
5.65.2.2 testBadConstruct_Production()	402
5.65.2.3 testConstruct_Production()	403
5.66 test/source/Storage/test_Lilon.cpp File Reference	404
5.66.1 Detailed Description	405
5.66.2 Function Documentation	405
5.66.2.1 main()	405
5.66.2.2 testBadConstruct_Lilon()	406
5.66.2.3 testCommitCharge_Lilon()	406
5.66.2.4 testCommitDischarge_Lilon()	407
5.66.2.5 testConstruct_Lilon()	408
5.67 test/source/Storage/test_Storage.cpp File Reference	409
5.67.1 Detailed Description	410
5.67.2 Function Documentation	410
5.67.2.1 main()	410
5.67.2.2 testBadConstruct_Storage()	411
5.67.2.3 testConstruct_Storage()	411
5.68 test/source/test_Controller.cpp File Reference	412
5.68.1 Detailed Description	413
5.68.2 Function Documentation	413
5.68.2.1 main()	413
5.68.2.2 testConstruct_Controller()	414
5.69 test/source/test_ElectricalLoad.cpp File Reference	414
5.69.1 Detailed Description	414
5.69.2 Function Documentation	415
5.69.2.1 main()	415
5.69.2.2 testConstruct_ElectricalLoad()	415
5.69.2.3 testDataRead_ElectricalLoad()	416
5.69.2.4 testPostConstructionAttributes_ElectricalLoad()	417
5.70 test/source/test_Interpolator.cpp File Reference	418
5.70.1 Detailed Description	418
5.70.2 Function Documentation	418
5.70.2.1 main()	419
5.70.2.2 testBadIndexing1D_Interpolator()	419
5.70.2.3 testConstruct_Interpolator()	420
5.70.2.4 testDataRead1D_Interpolator()	420
5.70.2.5 testDataRead2D_Interpolator()	422

5.70.2.6 testInterpolation1D_Interpolator()	424
5.70.2.7 testInterpolation2D_Interpolator()	425
5.70.2.8 testInvalidInterpolation1D_Interpolator()	426
5.70.2.9 testInvalidInterpolation2D_Interpolator()	427
5.71 test/source/test_Model.cpp File Reference	428
5.71.1 Detailed Description	429
5.71.2 Function Documentation	429
5.71.2.1 main()	430
5.71.2.2 testAddDiesel_Model()	431
5.71.2.3 testAddHydro_Model()	432
5.71.2.4 testAddHydroResource_Model()	433
5.71.2.5 testAddLilon_Model()	434
5.71.2.6 testAddSolar_Model()	435
5.71.2.7 testAddSolar_productionOverride_Model()	435
5.71.2.8 testAddSolarResource_Model()	436
5.71.2.9 testAddTidal_Model()	437
5.71.2.10 testAddTidalResource_Model()	438
5.71.2.11 testAddWave_Model()	439
5.71.2.12 testAddWaveResource_Model()	439
5.71.2.13 testAddWind_Model()	441
5.71.2.14 testAddWindResource_Model()	442
5.71.2.15 testBadConstruct_Model()	443
5.71.2.16 testConstruct_Model()	444
5.71.2.17 testEconomics_Model()	444
5.71.2.18 testElectricalLoadData_Model()	444
5.71.2.19 testFuelConsumptionEmissions_Model()	446
5.71.2.20 testLoadBalance_Model()	446
5.71.2.21 testPostConstructionAttributes_Model()	448
5.72 test/source/test_Resources.cpp File Reference	448
5.72.1 Detailed Description	449
5.72.2 Function Documentation	450
5.72.2.1 main()	450
5.72.2.2 testAddHydroResource_Resources()	451
5.72.2.3 testAddSolarResource_Resources()	452
5.72.2.4 testAddTidalResource_Resources()	453
5.72.2.5 testAddWaveResource_Resources()	456
5.72.2.6 testAddWindResource_Resources()	457
5.72.2.7 testBadAdd_Resources()	459
5.72.2.8 testConstruct_Resources()	460
5.73 test/utls/testing_utls.cpp File Reference	460
5.73.1 Detailed Description	461
5.73.2 Function Documentation	461

5.73.2.1 expectedErrorNotDetected()	461
5.73.2.2 printGold()	462
5.73.2.3 printGreen()	462
5.73.2.4 printRed()	462
5.73.2.5 testFloatEquals()	463
5.73.2.6 testGreaterThan()	463
5.73.2.7 testGreaterThanOrEqualTo()	464
5.73.2.8 testLessThan()	465
5.73.2.9 testLessThanOrEqualTo()	465
5.73.2.10 testTruth()	466
5.74 test/utls/testing_utils.h File Reference	466
5.74.1 Detailed Description	467
5.74.2 Macro Definition Documentation	468
5.74.2.1 FLOAT_TOLERANCE	468
5.74.3 Function Documentation	468
5.74.3.1 expectedErrorNotDetected()	468
5.74.3.2 printGold()	468
5.74.3.3 printGreen()	469
5.74.3.4 printRed()	469
5.74.3.5 testFloatEquals()	469
5.74.3.6 testGreaterThan()	470
5.74.3.7 testGreaterThanOrEqualTo()	471
5.74.3.8 testLessThan()	471
5.74.3.9 testLessThanOrEqualTo()	472
5.74.3.10 testTruth()	472
Bibliography	476
Index	477

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

CombustionInputs	24
Controller	26
DieselInputs	60
ElectricalLoad	64
Emissions	69
HydroInputs	90
Interpolator	92
InterpolatorStruct1D	106
InterpolatorStruct2D	107
LilonInputs	129
Model	134
ModelInputs	153
NoncombustionInputs	162
Production	163
Combustion	9
Diesel	46
Noncombustion	154
Hydro	71
Renewable	181
Solar	204
Tidal	230
Wave	245
Wind	262
ProductionInputs	179
RenewableInputs	189
Resources	190
SolarInputs	213
Storage	215
Lilon	110
StorageInputs	228
TidalInputs	243
WaveInputs	259
WindInputs	275

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	9
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 . . .	24
Controller	A class which contains a various dispatch control logic. Intended to serve as a component class of Model	26
Diesel	A derived class of the Combustion branch of Production which models production using a diesel generator	46
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 . . .	60
ElectricalLoad	A class which contains time and electrical load data. Intended to serve as a component class of Model	64
Emissions	A structure which bundles the emitted masses of various emissions chemistries	69
Hydro	A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not)	71
HydroInputs	A structure which bundles the necessary inputs for the Hydro constructor. Provides default values for every necessary input. Note that this structure encapsulates NoncombustionInputs	90
Interpolator	A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies	92
InterpolatorStruct1D	A struct which holds two parallel vectors for use in 1D interpolation	106
InterpolatorStruct2D	A struct which holds two parallel vectors and a matrix for use in 2D interpolation	107
Lilon	A derived class of Storage which models energy storage by way of lithium-ion batteries	110

LilonInputs	A structure which bundles the necessary inputs for the Lilon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs	129
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	134
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)	153
Noncombustion	The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion	154
NoncombustionInputs	A structure which bundles the necessary inputs for the Noncombustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs	162
Production	The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise	163
ProductionInputs	A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input	179
Renewable	The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy	181
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	189
Resources	A class which contains renewable resource data. Intended to serve as a component class of Model	190
Solar	A derived class of the Renewable branch of Production which models solar production	204
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	213
Storage	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	215
StorageInputs	A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input	228
Tidal	A derived class of the Renewable branch of Production which models tidal production	230
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	243
Wave	A derived class of the Renewable branch of Production which models wave production	245
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	259
Wind	A derived class of the Renewable branch of Production which models wind production	262
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	275

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

header/ Controller.h	
Header file for the Controller class	277
header/ doxygen_cite.h	
Header file which simply cites the doxygen tool	278
header/ ElectricalLoad.h	
Header file for the ElectricalLoad class	279
header/ Interpolator.h	
Header file for the Interpolator class	279
header/ Model.h	
Header file for the Model class	280
header/ Resources.h	
Header file for the Resources class	296
header/ std_includes.h	
Header file which simply batches together some standard includes	297
header/Production/ Production.h	
Header file for the Production class	288
header/Production/Combustion/ Combustion.h	
Header file for the Combustion class	281
header/Production/Combustion/ Diesel.h	
Header file for the Diesel class	284
header/Production/Noncombustion/ Hydro.h	
Header file for the Hydro class	285
header/Production/Noncombustion/ Noncombustion.h	
Header file for the Noncombustion class	287
header/Production/Renewable/ Renewable.h	
Header file for the Renewable class	289
header/Production/Renewable/ Solar.h	
Header file for the Solar class	290
header/Production/Renewable/ Tidal.h	
Header file for the Tidal class	291
header/Production/Renewable/ Wave.h	
Header file for the Wave class	293
header/Production/Renewable/ Wind.h	
Header file for the Wind class	294
header/Storage/ Lilon.h	
Header file for the Lilon class	297

header/Storage/Storage.h	
Header file for the Storage class	298
projects/example.cpp	300
pybindings/PYBIND11_PGM.cpp	
Bindings file for PGMcpp	304
pybindings/snippets/PYBIND11_Controller.cpp	
Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp	330
pybindings/snippets/PYBIND11_ElectricalLoad.cpp	
Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp	332
pybindings/snippets/PYBIND11_Interpolator.cpp	
Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp	334
pybindings/snippets/PYBIND11_Model.cpp	
Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp	336
pybindings/snippets/PYBIND11_Resources.cpp	
Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp	337
pybindings/snippets/Production/PYBIND11_Production.cpp	
Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp	316
pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp	
Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp	306
pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp	
Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp	308
pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp	
Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp	311
pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp	
Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp	315
pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp	
Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp	321
pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp	
Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp	323
pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp	
Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp	324
pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp	
Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp	326
pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp	
Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp	328
pybindings/snippets/Storage/PYBIND11_Lilon.cpp	
Bindings file for the Lilon class. Intended to be #include'd in PYBIND11_PGM.cpp	338
pybindings/snippets/Storage/PYBIND11_Storage.cpp	
Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp	342
source/Controller.cpp	
Implementation file for the Controller class	344
source/ElectricalLoad.cpp	
Implementation file for the ElectricalLoad class	344
source/Interpolator.cpp	
Implementation file for the Interpolator class	345
source/Model.cpp	
Implementation file for the Model class	345
source/Resources.cpp	
Implementation file for the Resources class	352
source/Production/Production.cpp	
Implementation file for the Production class	348
source/Production/Combustion/Combustion.cpp	
Implementation file for the Combustion class	346
source/Production/Combustion/Diesel.cpp	
Implementation file for the Diesel class	346
source/Production/Noncombustion/Hydro.cpp	
Implementation file for the Hydro class	347

source/Production/Noncombustion/ Noncombustion.cpp	
Implementation file for the Noncombustion class	347
source/Production/Renewable/ Renewable.cpp	
Implementation file for the Renewable class	349
source/Production/Renewable/ Solar.cpp	
Implementation file for the Solar class	349
source/Production/Renewable/ Tidal.cpp	
Implementation file for the Tidal class	350
source/Production/Renewable/ Wave.cpp	
Implementation file for the Wave class	350
source/Production/Renewable/ Wind.cpp	
Implementation file for the Wind class	351
source/Storage/ Lilon.cpp	
Implementation file for the Lilon class	352
source/Storage/ Storage.cpp	
Implementation file for the Storage class	353
test/source/ test_Controller.cpp	
Testing suite for Controller class	412
test/source/ test_ElectricalLoad.cpp	
Testing suite for ElectricalLoad class	414
test/source/ test_Interpolator.cpp	
Testing suite for Interpolator class	418
test/source/ test_Model.cpp	
Testing suite for Model class	428
test/source/ test_Resources.cpp	
Testing suite for Resources class	448
test/source/Production/ test_Production.cpp	
Testing suite for Production class	401
test/source/Production/Combustion/ test_Combustion.cpp	
Testing suite for Combustion class	353
test/source/Production/Combustion/ test_Diesel.cpp	
Testing suite for Diesel class	356
test/source/Production/Noncombustion/ test_Hydro.cpp	
Testing suite for Hydro class	366
test/source/Production/Noncombustion/ test_Noncombustion.cpp	
Testing suite for Noncombustion class	372
test/source/Production/Renewable/ test_Renewable.cpp	
Testing suite for Renewable class	374
test/source/Production/Renewable/ test_Solar.cpp	
Testing suite for Solar class	376
test/source/Production/Renewable/ test_Tidal.cpp	
Testing suite for Tidal class	383
test/source/Production/Renewable/ test_Wave.cpp	
Testing suite for Wave class	388
test/source/Production/Renewable/ test_Wind.cpp	
Testing suite for Wind class	395
test/source/Storage/ test_Lilon.cpp	
Testing suite for Lilon class	404
test/source/Storage/ test_Storage.cpp	
Testing suite for Storage class	409
test/utills/ testing_utils.cpp	
Implementation file for various PGMcpp testing utilities	460
test/utills/ testing_utils.h	
Header file for various PGMcpp testing utilities	466

Chapter 4

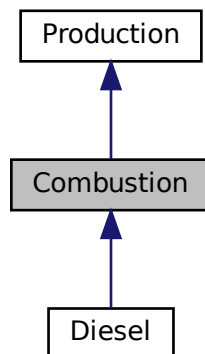
Class Documentation

4.1 Combustion Class Reference

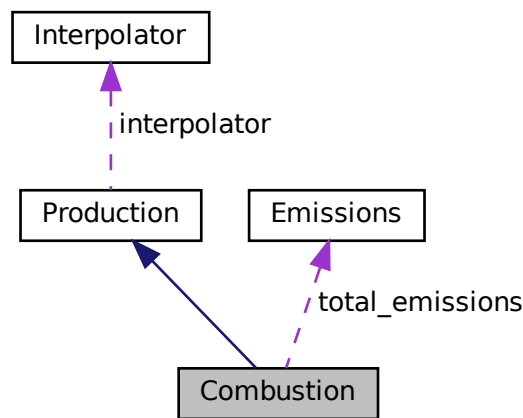
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Public Attributes

- [CombustionType](#) type
The type (CombustionType) of the asset.
- [FuelMode](#) [fuel_mode](#)
The fuel mode to use in modelling fuel consumption.
- [Emissions](#) [total_emissions](#)
An [Emissions](#) structure for holding total emissions [kg].
- double [fuel_cost_L](#)
The cost of fuel [1/L] (undefined currency).
- double [nominal_fuel_escalation_annual](#)
The nominal, annual fuel escalation rate to use in computing model economics.
- double [real_fuel_escalation_annual](#)
The real, annual fuel escalation rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double [linear_fuel_slope_LkWh](#)
The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double [linear_fuel_intercept_LkWh](#)
The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double [cycle_charging_setpoint](#)
The cycle charging set point (the load ratio at which to produce when running in cycle charging mode).
- 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.
- std::string [fuel_mode_str](#)
A string describing the fuel mode of the asset.
- std::vector< double > [fuel_consumption_vec_L](#)
A vector of fuel consumed [L] over each modelling time step.
- std::vector< double > [fuel_cost_vec](#)
A vector of fuel costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > [CO2_emissions_vec_kg](#)
A vector of carbon dioxide (CO2) emitted [kg] over each modelling time step.
- std::vector< double > [CO_emissions_vec_kg](#)
A vector of carbon monoxide (CO) emitted [kg] over each modelling time step.
- std::vector< double > [NOx_emissions_vec_kg](#)
A vector of nitrogen oxide (NOx) emitted [kg] over each modelling time step.
- std::vector< double > [SOx_emissions_vec_kg](#)
A vector of sulfur oxide (SOx) emitted [kg] over each modelling time step.
- std::vector< double > [CH4_emissions_vec_kg](#)
A vector of methane (CH4) emitted [kg] over each modelling time step.
- std::vector< double > [PM_emissions_vec_kg](#)
A vector of particulate matter (PM) emitted [kg] over each modelling time step.

Private Member Functions

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

4.1.1 Detailed Description

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

4.1.2 Constructor & Destructor Documentation

4.1.2.1 `Combustion()` [1/2]

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

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

```
117 {
118     return;
119 } /* Combustion() */
```

4.1.2.2 `Combustion()` [2/2]

```
Combustion::Combustion (
    int n_points,
    double n_years,
    CombustionInputs combustion_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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

Parameters

<code>n_points</code>	The number of points in the modelling time series.
<code>n_years</code>	The number of years being modelled.
<code>combustion_inputs</code>	A structure of Combustion constructor inputs.
<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.

```
151 :
152 Production(
153     n_points,
154     n_years,
155     combustion_inputs.production_inputs,
156     time_vec_hrs_ptr
157 )
```

```

158 {
159     // 1. check inputs
160     this->__checkInputs(combustion_inputs);
161
162     // 2. set attributes
163     this->fuel_mode = combustion_inputs.fuel_mode;
164
165     switch (this->fuel_mode) {
166         case (FuelMode :: FUEL_MODE_LINEAR): {
167             this->fuel_mode_str = "FUEL_MODE_LINEAR";
168
169             break;
170         }
171
172         case (FuelMode :: FUEL_MODE_LOOKUP): {
173             this->fuel_mode_str = "FUEL_MODE_LOOKUP";
174
175             this->interpolator.addData1D(
176                 0,
177                 combustion_inputs.path_2_fuel_interp_data
178             );
179
180             break;
181         }
182
183         default: {
184             std::string error_str = "ERROR: Combustion(): ";
185             error_str += "fuel mode ";
186             error_str += std::to_string(this->fuel_mode);
187             error_str += " not recognized";
188
189             #ifdef _WIN32
190                 std::cout << error_str << std::endl;
191             #endif
192
193             throw std::runtime_error(error_str);
194
195             break;
196         }
197     }
198
199     this->fuel_cost_L = 0;
200     this->nominal_fuel_escalation_annual =
201         combustion_inputs.nominal_fuel_escalation_annual;
202
203     this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
204         combustion_inputs.nominal_fuel_escalation_annual,
205         combustion_inputs.production_inputs.nominal_discount_annual
206     );
207
208     this->linear_fuel_slope_LkWh = 0;
209     this->linear_fuel_intercept_LkWh = 0;
210
211     this->cycle_charging_setpoint = combustion_inputs.cycle_charging_setpoint;
212
213     this->CO2_emissions_intensity_kgL = 0;
214     this->CO_emissions_intensity_kgL = 0;
215     this->NOx_emissions_intensity_kgL = 0;
216     this->SOx_emissions_intensity_kgL = 0;
217     this->CH4_emissions_intensity_kgL = 0;
218     this->PM_emissions_intensity_kgL = 0;
219
220     this->total_fuel_consumed_L = 0;
221
222     this->fuel_consumption_vec_L.resize(this->n_points, 0);
223     this->fuel_cost_vec.resize(this->n_points, 0);
224
225     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
226     this->CO_emissions_vec_kg.resize(this->n_points, 0);
227     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
228     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
229     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
230     this->PM_emissions_vec_kg.resize(this->n_points, 0);
231
232     // 3. construction print
233     if (this->print_flag) {
234         std::cout << "Combustion object constructed at " << this << std::endl;
235     }
236
237     return;
238 } /* Combustion() */

```

4.1.2.3 ~Combustion()

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

Destructor for the [Combustion](#) class.

```
576 {
577     // 1. destruction print
578     if (this->print_flag) {
579         std::cout << "Combustion object at " << this << " destroyed" << std::endl;
580     }
581
582     return;
583 } /* ~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.
--------------------------	---

```
65 {
66     // 1. if FUEL_MODE_LOOKUP, check that path is given
67     if (
68         combustion_inputs.fuel_mode == FuelMode :: FUEL_MODE_LOOKUP and
69         combustion_inputs.path_2_fuel_interp_data.empty()
70     ) {
71         std::string error_str = "ERROR: Combustion() fuel mode was set to ";
72         error_str += "FuelMode::FUEL_MODE_LOOKUP, but no path to fuel interpolation ";
73         error_str += "data was given";
74
75         #ifdef _WIN32
76             std::cout << error_str << std::endl;
77         #endif
78
79         throw std::invalid_argument(error_str);
80     }
81
82     // 2. cycle charging setpoint
83     if (
84         combustion_inputs.cycle_charging_setpoint < 0 or
85         combustion_inputs.cycle_charging_setpoint > 1
86     ) {
87         std::string error_str = "ERROR: Combustion() cycle charging set point ";
88         error_str += "must be in the closed interval [0, 1].";
89
90         #ifdef _WIN32
91             std::cout << error_str << std::endl;
92         #endif
93
94         throw std::invalid_argument(error_str);
95     }
96
97     return;
98 } /* __checkInputs() */
```

4.1.3.2 __writeSummary()

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

Reimplemented in [Diesel](#).

```
131 {return;}
```

4.1.3.3 __writeTimeSeries()

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

Reimplemented in [Diesel](#).

```
136 {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](#).

```
368 {
369     // 1. invoke base class method
370     load_kW = Production::commit(
371         timestep,
372         dt_hrs,
373         production_kW,
374         load_kW
```

```

375     );
376
377
378     if (this->is_running) {
379         // 2. compute and record fuel consumption
380         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
381         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
382
383         // 3. compute and record emissions
384         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
385         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
386         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
387         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
388         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
389         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
390         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
391
392         // 4. incur fuel costs
393         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
394     }
395
396     return load_kW;
397 } /* commit() */

```

4.1.3.5 computeEconomics()

```

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

```

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

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

Parameters

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

Reimplemented from [Production](#).

```

312 {
313     // 1. account for fuel costs in net present cost
314     double t_hrs = 0;
315     double real_fuel_escalation_scalar = 0;
316
317     for (int i = 0; i < this->n_points; i++) {
318         t_hrs = time_vec_hrs_ptr->at(i);
319
320         real_fuel_escalation_scalar = 1.0 / pow(
321             1 + this->real_fuel_escalation_annual,
322             t_hrs / 8760
323         );
324
325         this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
326     }
327
328     // 2. invoke base class method
329     Production :: computeEconomics(time_vec_hrs_ptr);
330
331     return;
332 } /* computeEconomics() */

```

4.1.3.6 computeFuelAndEmissions()

```

void Combustion::computeFuelAndEmissions (
    void )

```

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

```
280 {
281     for (int i = 0; i < n_points; i++) {
282         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
283
284         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
285         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
286         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
287         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
288         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
289         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
290     }
291
292     return;
293 } /* 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.

```
476     {
477     Emissions emissions;
478
479     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
480     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
481     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
482     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
483     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
484     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
485
486     return emissions;
487 } /* 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].

```

419 {
420     double fuel_consumed_L = 0;
421
422     switch (this->fuel_mode) {
423         case (FuelMode :: FUEL_MODE_LINEAR): {
424             fuel_consumed_L = (
425                 this->linear_fuel_slope_LkWh * production_kW +
426                 this->linear_fuel_intercept_LkWh * this->capacity_kW
427             ) * dt_hrs;
428
429             break;
430         }
431
432         case (FuelMode :: FUEL_MODE_LOOKUP): {
433             double load_ratio = production_kW / this->capacity_kW;
434
435             fuel_consumed_L = this->interpolator.interp1D(0, load_ratio) * dt_hrs;
436
437             break;
438         }
439
440         default: {
441             std::string error_str = "ERROR: Combustion::getFuelConsumptionL(): ";
442             error_str += "fuel mode ";
443             error_str += std::to_string(this->fuel_mode);
444             error_str += " not recognized";
445
446             #ifdef _WIN32
447                 std::cout << error_str << std::endl;
448             #endif
449
450             throw std::runtime_error(error_str);
451
452             break;
453         }
454     }
455
456     return fuel_consumed_L;
457 } /* getFuelConsumptionL() */

```

4.1.3.9 handleReplacement()

```

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

```

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

Parameters

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

Reimplemented from [Production](#).

Reimplemented in [Diesel](#).

```

256 {
257     // 1. reset attributes
258     //...
259
260     // 2. invoke base class method
261     Production :: handleReplacement(timestep);
262
263     return;
264 } /* __handleReplacement() */

```


4.1.3.10 requestProductionkW()

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

Reimplemented in [Diesel](#).

```
184 {return 0;}
```

4.1.3.11 writeResults()

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

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

Parameters

<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.

```
523 {
524     // 1. handle sentinel
525     if (max_lines < 0) {
526         max_lines = this->n_points;
527     }
528
529     // 2. create subdirectories
530     write_path += "Production/";
531     if (not std::filesystem::is_directory(write_path)) {
532         std::filesystem::create_directory(write_path);
533     }
534
535     write_path += "Combustion/";
536     if (not std::filesystem::is_directory(write_path)) {
537         std::filesystem::create_directory(write_path);
538     }
539
540     write_path += this->type_str;
541     write_path += "_";
542     write_path += std::to_string(int(ceil(this->capacity_kW)));
543     write_path += "kW_idx";
544     write_path += std::to_string(combustion_index);
545     write_path += "/";
546     std::filesystem::create_directory(write_path);
547
548     // 3. write summary
549     this->__writeSummary(write_path);
550
551     // 4. write time series
552     if (max_lines > this->n_points) {
553         max_lines = this->n_points;
554     }
555
556     if (max_lines > 0) {
557         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
558     }
```

```
559
560     return;
561 } /* 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 cycle_charging_setpoint

```
double Combustion::cycle_charging_setpoint
```

The cycle charging set point (the load ratio at which to produce when running in cycle charging mode).

4.1.4.8 fuel_consumption_vec_L

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

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

4.1.4.9 fuel_cost_L

```
double Combustion::fuel_cost_L
```

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

4.1.4.10 fuel_cost_vec

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

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

4.1.4.11 fuel_mode

```
FuelMode Combustion::fuel_mode
```

The fuel mode to use in modelling fuel consumption.

4.1.4.12 fuel_mode_str

```
std::string Combustion::fuel_mode_str
```

A string describing the fuel mode of the asset.

4.1.4.13 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.

4.1.4.14 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.

4.1.4.15 nominal_fuel_escalation_annual

```
double Combustion::nominal_fuel_escalation_annual
```

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

4.1.4.16 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.17 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.

4.1.4.18 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.19 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.

4.1.4.20 real_fuel_escalation_annual

```
double Combustion::real_fuel_escalation_annual
```

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

4.1.4.21 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.22 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.

4.1.4.23 total_emissions

```
Emissions Combustion::total_emissions
```

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

4.1.4.24 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

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

4.1.4.25 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

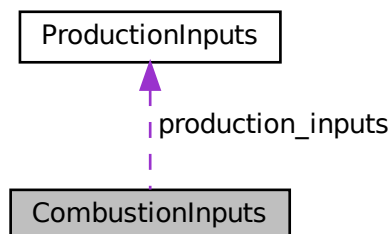
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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.
- [FuelMode](#) [fuel_mode](#) = [FuelMode](#) :: [FUEL_MODE_LINEAR](#)
The fuel mode to use in modelling fuel consumption.
- double [nominal_fuel_escalation_annual](#) = 0.05
The nominal, annual fuel escalation rate to use in computing model economics.
- double [cycle_charging_setpoint](#) = 0.85
The cycle charging set point (the load ratio at which to produce when running in cycle charging mode).
- std::string [path_2_fuel_interp_data](#) = ""
A path (either relative or absolute) to a set of fuel consumption data.

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 cycle_charging_setpoint

```
double CombustionInputs::cycle_charging_setpoint = 0.85
```

The cycle charging set point (the load ratio at which to produce when running in cycle charging mode).

4.2.2.2 fuel_mode

```
FuelMode CombustionInputs::fuel_mode = FuelMode :: FUEL_MODE_LINEAR
```

The fuel mode to use in modelling fuel consumption.

4.2.2.3 nominal_fuel_escalation_annual

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

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

4.2.2.4 path_2_fuel_interp_data

```
std::string CombustionInputs::path_2_fuel_interp_data = ""
```

A path (either relative or absolute) to a set of fuel consumption data.

4.2.2.5 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](#) *, [Resources](#) *, [std::vector](#)< [Combustion](#) * > *, [std::vector](#)< [Noncombustion](#) * > *, [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](#) *, [Resources](#) *, [std::vector](#)< [Combustion](#) * > *, [std::vector](#)< [Noncombustion](#) * > *, [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](#) *, [Resources](#) *, [std::vector](#)< [Combustion](#) * > *, [std::vector](#)< [Noncombustion](#) * > *, [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](#) *, [Resources](#) *, std::vector< [Combustion](#) * > *, std::vector< [Noncombustion](#) * > *, 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](#) *, [Resources](#) *, std::vector< [Combustion](#) * > *, std::vector< [Noncombustion](#) * > *, std::vector< [Renewable](#) * > *, std::vector< [Storage](#) * > *)
Helper method to apply cycle charging control action for given timestep of the [Model](#) run when net load > 0. Defaults to load following control if no depleted storage assets.
- void `__handleStorageCharging` (int, double, std::list< [Storage](#) * >, std::vector< [Combustion](#) * > *, std::vector< [Noncombustion](#) * > *, std::vector< [Renewable](#) * > *)
Helper method to handle the charging of the given [Storage](#) assets.
- void `__handleStorageCharging` (int, double, std::vector< [Storage](#) * > *, std::vector< [Combustion](#) * > *, std::vector< [Noncombustion](#) * > *, std::vector< [Renewable](#) * > *)
Helper method to handle the charging of the given [Storage](#) assets.
- double `__getRenewableProduction` (int, double, [Renewable](#) *, [Resources](#) *)
Helper method to compute the production from the given [Renewable](#) asset at the given point in time.
- double `__handleCombustionDispatch` (int, double, double, std::vector< [Combustion](#) * > *, bool)
bool is_cycle_charging)
- double `__handleNoncombustionDispatch` (int, double, double, std::vector< [Noncombustion](#) * > *, [Resources](#) *)
- 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.

```
1273 {
1274     return;
1275 } /* Controller() */
```

4.3.2.2 ~Controller()

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

Destructor for the [Controller](#) class.

```
1519 {
1520     this->clear();
1521
1522     return;
1523 } /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

```
void Controller::__applyCycleChargingControl_CHARGING (
    int timestep,
    ElectricalLoad * electrical_load_ptr,
    Resources * resources_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_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>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 .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion 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 .

```
475 {
476     // 1. default to load following
477     this->__applyLoadFollowingControl_CHARGING(
478         timestep,
479         electrical_load_ptr,
480         resources_ptr,
481         combustion_ptr_vec_ptr,
482         noncombustion_ptr_vec_ptr,
483         renewable_ptr_vec_ptr,
484         storage_ptr_vec_ptr
485     );
486     return;
487 } /* __applyCycleChargingControl_CHARGING() */
```

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

```
void Controller::__applyCycleChargingControl_DISCHARGING (
    int timestep,
    ElectricalLoad * electrical_load_ptr,
    Resources * resources_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_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>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 .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion 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

```

536 {
537     // 1. get dt_hrs, net load
538     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
539     double net_load_kW = this->net_load_vec_kW[timestep];
540
541     // 2. partition Storage assets into depleted and non-depleted
542     std::list<Storage*> depleted_storage_ptr_list;
543     std::list<Storage*> nondepleted_storage_ptr_list;
544
545     Storage* storage_ptr;
546     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
547         storage_ptr = storage_ptr_vec_ptr->at(i);
548
549         if (storage_ptr->is_depleted) {
550             depleted_storage_ptr_list.push_back(storage_ptr);
551         }
552         else {
553             nondepleted_storage_ptr_list.push_back(storage_ptr);
554         }
555     }
556
557     // 3. discharge non-depleted storage assets
558     net_load_kW = this->__handleStorageDischarging(
559         timestep,
560         dt_hrs,
561         net_load_kW,
562         nondepleted_storage_ptr_list
563     );
564
565     // 4. request optimal production from all Noncombustion assets
566     net_load_kW = this->__handleNoncombustionDispatch(
567         timestep,
568         dt_hrs,
569         net_load_kW,
570         noncombustion_ptr_vec_ptr,
571         resources_ptr
572     );
573
574     // 5. request optimal production from all Combustion assets
575     // default to load following if no depleted storage
576     if (depleted_storage_ptr_list.empty()) {
577         net_load_kW = this->__handleCombustionDispatch(
578             timestep,
579             dt_hrs,
580             net_load_kW,
581             combustion_ptr_vec_ptr,
582             false // is_cycle_charging
583         );
584     }
585     else {
586         net_load_kW = this->__handleCombustionDispatch(
587             timestep,
588             dt_hrs,
589             net_load_kW,
590             combustion_ptr_vec_ptr,
591             true // is_cycle_charging
592         );
593     }
594
595     // 6. attempt to charge depleted Storage assets using any and all available
596     // charge priority is Combustion, then Renewable
597     this->__handleStorageCharging(
598         timestep,
599         dt_hrs,
600         depleted_storage_ptr_list,

```

```

604         combustion_ptr_vec_ptr,
605         noncombustion_ptr_vec_ptr,
606         renewable_ptr_vec_ptr
607     );
608
609     // 7. record any missed load
610     if (net_load_kW > 1e-6) {
611         this->missed_load_vec_kW[timestep] = net_load_kW;
612     }
613
614     return;
615 } /* __applyCycleChargingControl_DISCHARGING() */

```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```

void Controller::__applyLoadFollowingControl_CHARGING (
    int timestep,
    ElectricalLoad * electrical_load_ptr,
    Resources * resources_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_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>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 .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion 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 .

```

280 {
281     // 1. get dt_hrs, set net load
282     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
283     double net_load_kW = 0;
284
285     // 2. request zero production from all Combustion assets
286     this->__handleCombustionDispatch(
287         timestep,
288         dt_hrs,
289         net_load_kW,
290         combustion_ptr_vec_ptr,
291         false // is_cycle_charging
292     );
293
294     // 3. request zero production from all Noncombustion assets
295     this->__handleNoncombustionDispatch(
296         timestep,
297         dt_hrs,
298         net_load_kW,
299         noncombustion_ptr_vec_ptr,
300         resources_ptr
301     );
302
303     // 4. attempt to charge all Storage assets using any and all available curtailment
304     // charge priority is Combustion, then Renewable
305     this->__handleStorageCharging(
306         timestep,
307         dt_hrs,
308         storage_ptr_vec_ptr,
309         combustion_ptr_vec_ptr,
310         noncombustion_ptr_vec_ptr,

```

```

311         renewable_ptr_vec_ptr
312     );
313
314     return;
315 } /* __applyLoadFollowingControl_CHARGING() */

```

4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

```

void Controller::__applyLoadFollowingControl_DISCHARGING (
    int timestep,
    ElectricalLoad * electrical_load_ptr,
    Resources * resources_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_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>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 .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion 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

```

362 {
363     // 1. get dt_hrs, net load
364     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
365     double net_load_kW = this->net_load_vec_kW[timestep];
366
367     // 2. partition Storage assets into depleted and non-depleted
368     std::list<Storage*> depleted_storage_ptr_list;
369     std::list<Storage*> nondepleted_storage_ptr_list;
370
371     Storage* storage_ptr;
372     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
373         storage_ptr = storage_ptr_vec_ptr->at(i);
374
375         if (storage_ptr->is_depleted) {
376             depleted_storage_ptr_list.push_back(storage_ptr);
377         }
378         else {
379             nondepleted_storage_ptr_list.push_back(storage_ptr);
380         }
381     }
382 }
383
384 // 3. discharge non-depleted storage assets
385 net_load_kW = this->__handleStorageDischarging(
386     timestep,
387     dt_hrs,
388     net_load_kW,
389     nondepleted_storage_ptr_list
390 );
391
392 // 4. request optimal production from all Noncombustion assets
393 net_load_kW = this->__handleNoncombustionDispatch(
394     timestep,
395     dt_hrs,
396     net_load_kW,

```

```

397         noncombustion_ptr_vec_ptr,
398         resources_ptr
399     );
400
401     // 5. request optimal production from all Combustion assets
402     net_load_kW = this->__handleCombustionDispatch(
403         timestep,
404         dt_hrs,
405         net_load_kW,
406         combustion_ptr_vec_ptr,
407         false // is_cycle_charging
408     );
409
410     // 6. attempt to charge depleted Storage assets using any and all available
411     //     charge priority is Combustion, then Renewable
412     this->__handleStorageCharging(
413         timestep,
414         dt_hrs,
415         depleted_storage_ptr_list,
416         combustion_ptr_vec_ptr,
417         noncombustion_ptr_vec_ptr,
418         renewable_ptr_vec_ptr
419     );
420
421
422     // 7. record any missed load
423     if (net_load_kW > 1e-6) {
424         this->missed_load_vec_kW[timestep] = net_load_kW;
425     }
426
427     return;
428 } /* __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 .

```

82 {
83     // 1. init
84     this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
85     this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
86
87     // 2. populate net load vector
88     double dt_hrs = 0;
89     double load_kW = 0;
90     double net_load_kW = 0;
91     double production_kW = 0;
92
93     Renewable* renewable_ptr;
94
95     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
96         dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
97         load_kW = electrical_load_ptr->load_vec_kW[i];
98         net_load_kW = load_kW;
99
100         for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {

```

```

101         renewable_ptr = renewable_ptr_vec_ptr->at(j);
102
103         production_kW = this->__getRenewableProduction(
104             i,
105             dt_hrs,
106             renewable_ptr,
107             resources_ptr
108         );
109
110         load_kW = renewable_ptr->commit(
111             i,
112             dt_hrs,
113             production_kW,
114             load_kW
115         );
116
117         net_load_kW -= production_kW;
118     }
119
120     this->net_load_vec_kW[i] = net_load_kW;
121 }
122
123 return;
124 } /* __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

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

```

146 {
147     // 1. get state table dimensions
148     int n_cols = combustion_ptr_vec_ptr->size();
149     int n_rows = pow(2, n_cols);
150
151     // 2. init state table (all possible on/off combinations)
152     std::vector<std::vector<bool>> state_table;
153     state_table.resize(n_rows, {});
154
155     int x = 0;
156     for (int i = 0; i < n_rows; i++) {
157         state_table[i].resize(n_cols, false);
158
159         x = i;
160         for (int j = 0; j < n_cols; j++) {
161             if (x % 2 == 0) {
162                 state_table[i][j] = true;
163             }
164             x /= 2;
165         }
166     }
167
168     // 3. construct combustion map (handle duplicates by keeping rows with minimum
169     //     trues)
170     double total_capacity_kW = 0;
171     int truth_count = 0;
172     int current_truth_count = 0;
173
174     for (int i = 0; i < n_rows; i++) {
175         total_capacity_kW = 0;
176         truth_count = 0;
177         current_truth_count = 0;
178
179         for (int j = 0; j < n_cols; j++) {
180             if (state_table[i][j]) {
181                 total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
182                 truth_count++;
183             }
184         }
185     }
186 }

```

```

185
186     if (this->combustion_map.count(total_capacity_kW) > 0) {
187         for (int j = 0; j < n_cols; j++) {
188             if (this->combustion_map[total_capacity_kW][j]) {
189                 current_truth_count++;
190             }
191         }
192
193         if (truth_count < current_truth_count) {
194             this->combustion_map.erase(total_capacity_kW);
195         }
196     }
197
198     this->combustion_map.insert(
199         std::pair<double, std::vector<bool>» (
200             total_capacity_kW,
201             state_table[i]
202         )
203     );
204 }
205
206 /*
207 // ==== TEST PRINT ==== //
208 std::cout << std::endl;
209
210 std::cout << "\t\t";
211 for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
212     std::cout << combustion_ptr_vec_ptr->at(i)->capacity_kW << "\t";
213 }
214 std::cout << std::endl;
215
216 std::map<double, std::vector<bool>::iterator iter;
217 for (
218     iter = this->combustion_map.begin();
219     iter != this->combustion_map.end();
220     iter++
221 ) {
222     std::cout << iter->first << "\t\t";
223
224     for (size_t i = 0; i < iter->second.size(); i++) {
225         std::cout << iter->second[i] << "\t";
226     }
227     std::cout << "]" << std::endl;
228 }
229 // ==== END TEST PRINT ==== //
230 //*/
231
232 return;
233 } /* __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.

```

904 {
905     double production_kW = 0;
906
907     switch (renewable_ptr->type) {
908         case (RenewableType :: SOLAR): {
909             double resource_value = 0;
910
911             if (not renewable_ptr->normalized_production_series_given) {
912                 resource_value =
913                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
914             }
915
916             production_kW = renewable_ptr->computeProductionkW(
917                 timestep,
918                 dt_hrs,
919                 resource_value
920             );
921
922             break;
923         }
924
925         case (RenewableType :: TIDAL): {
926             double resource_value = 0;
927
928             if (not renewable_ptr->normalized_production_series_given) {
929                 resource_value =
930                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
931             }
932
933             production_kW = renewable_ptr->computeProductionkW(
934                 timestep,
935                 dt_hrs,
936                 resource_value
937             );
938
939             break;
940         }
941
942         case (RenewableType :: WAVE): {
943             double significant_wave_height_m = 0;
944             double energy_period_s = 0;
945
946             if (not renewable_ptr->normalized_production_series_given) {
947                 significant_wave_height_m =
948                     resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0];
949
950                 energy_period_s =
951                     resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1];
952             }
953
954             production_kW = renewable_ptr->computeProductionkW(
955                 timestep,
956                 dt_hrs,
957                 significant_wave_height_m,
958                 energy_period_s
959             );
960
961             break;
962         }
963
964         case (RenewableType :: WIND): {
965             double resource_value = 0;
966
967             if (not renewable_ptr->normalized_production_series_given) {
968                 resource_value =
969                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
970             }
971
972             production_kW = renewable_ptr->computeProductionkW(
973                 timestep,
974                 dt_hrs,
975                 resource_value
976             );
977
978             break;
979         }
980
981         default: {
982             std::string error_str = "ERROR: Controller::__getRenewableProduction(): ";
983             error_str += "renewable type ";
984             error_str += std::to_string(renewable_ptr->type);
985             error_str += " not recognized";
986
987             #ifdef _WIN32

```

```

988         std::cout << error_str << std::endl;
989     #endif
990
991     throw std::runtime_error(error_str);
992
993     break;
994 }
995 }
996
997 return production_kW;
998 } /* __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]

```

bool is_cycle_charging)

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.

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.

```

1041 {
1042     // 1. get minimal Combustion dispatch
1043     double target_production_kW = 1.2 * net_load_kW;
1044     double total_capacity_kW = 0;
1045
1046     std::map<double, std::vector<bool>::iterator> iter = this->combustion_map.begin();
1047     while (iter != std::prev(this->combustion_map.end(), 1)) {
1048         if (target_production_kW <= total_capacity_kW) {
1049             break;
1050         }
1051         iter++;
1052         total_capacity_kW = iter->first;
1053     }
1054
1055     // 2. share load proportionally (by rated capacity) over active Combustion assets
1056     Combustion* combustion_ptr;
1057     double production_kW = 0;
1058     double request_kW = 0;
1059     double _net_load_kW = net_load_kW;
1060
1061     for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
1062         combustion_ptr = combustion_ptr_vec_ptr->at(i);
1063         if (total_capacity_kW > 0) {

```

```

1066         request_kW =
1067             int(this->combustion_map[total_capacity_kW][i]) *
1068             net_load_kW *
1069             (combustion_ptr->capacity_kW / total_capacity_kW);
1070     }
1071
1072     else {
1073         request_kW = 0;
1074     }
1075
1076     if (is_cycle_charging and request_kW > 0) {
1077         if (request_kW < combustion_ptr->cycle_charging_setpoint * combustion_ptr->capacity_kW) {
1078             request_kW = combustion_ptr->cycle_charging_setpoint * combustion_ptr->capacity_kW;
1079         }
1080     }
1081
1082     production_kW = combustion_ptr->requestProductionkW(
1083         timestep,
1084         dt_hrs,
1085         request_kW
1086     );
1087
1088     _net_load_kW = combustion_ptr->commit(
1089         timestep,
1090         dt_hrs,
1091         production_kW,
1092         _net_load_kW
1093     );
1094 }
1095
1096 return _net_load_kW;
1097 } /* __handleCombustionDispatch() */

```

4.3.3.9 __handleNoncombustionDispatch()

```

double Controller::__handleNoncombustionDispatch (
    int timestep,
    double dt_hrs,
    double net_load_kW,
    std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
    Resources * resources_ptr ) [private]
{
1138 {
1139     Noncombustion* noncombustion_ptr;
1140     double production_kW = 0;
1141
1142     for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
1143         noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
1144
1145         switch (noncombustion_ptr->type) {
1146             case (NoncombustionType :: HYDRO): {
1147                 double resource_value = 0;
1148
1149                 if (not noncombustion_ptr->normalized_production_series_given) {
1150                     resource_value =
1151                         resources_ptr->resource_map_1D[noncombustion_ptr->resource_key][timestep];
1152                 }
1153
1154                 production_kW = noncombustion_ptr->requestProductionkW(
1155                     timestep,
1156                     dt_hrs,
1157                     net_load_kW,
1158                     resource_value
1159                 );
1160
1161                 net_load_kW = noncombustion_ptr->commit(
1162                     timestep,
1163                     dt_hrs,
1164                     production_kW,
1165                     net_load_kW,
1166                     resource_value
1167                 );
1168
1169                 break;
1170             }
1171
1172             default: {
1173                 production_kW = noncombustion_ptr->requestProductionkW(

```

```

1174         timestep,
1175         dt_hrs,
1176         net_load_kW
1177     );
1178
1179     net_load_kW = noncombustion_ptr->commit (
1180         timestep,
1181         dt_hrs,
1182         production_kW,
1183         net_load_kW
1184     );
1185
1186     break;
1187 }
1188 }
1189 }
1190
1191 return net_load_kW;
1192 } /* __handleNoncombustionDispatch() */

```

4.3.3.10 __handleStorageCharging() [1/2]

```

void Controller::__handleStorageCharging (
    int timestep,
    double dt_hrs,
    std::list< Storage * > storage_ptr_list,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
    std::vector< Renewable * > * renewable_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_list</i>	A list of pointers to the Storage assets that are to be charged.
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .

```

658 {
659     double acceptable_kW = 0;
660     double curtailment_kW = 0;
661
662     Storage* storage_ptr;
663     Combustion* combustion_ptr;
664     Noncombustion* noncombustion_ptr;
665     Renewable* renewable_ptr;
666
667     std::list<Storage*>::iterator iter;
668     for (
669         iter = storage_ptr_list.begin();
670         iter != storage_ptr_list.end();
671         iter++
672     ){
673         storage_ptr = (*iter);
674
675         // 1. attempt to charge from Combustion curtailment first
676         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
677             combustion_ptr = combustion_ptr_vec_ptr->at(i);
678             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
679
680             if (curtailment_kW <= 0) {
681                 continue;
682             }
683
684             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

```

```

685
686         if (acceptable_kW > curtailment_kW) {
687             acceptable_kW = curtailment_kW;
688         }
689
690         combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
691         combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
692         storage_ptr->power_kW += acceptable_kW;
693     }
694
695     // 2. attempt to charge from Noncombustion curtailment second
696     for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
697         noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
698         curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
699
700         if (curtailment_kW <= 0) {
701             continue;
702         }
703
704         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
705
706         if (acceptable_kW > curtailment_kW) {
707             acceptable_kW = curtailment_kW;
708         }
709
710         noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
711         noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
712         storage_ptr->power_kW += acceptable_kW;
713     }
714
715     // 3. attempt to charge from Renewable curtailment third
716     for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
717         renewable_ptr = renewable_ptr_vec_ptr->at(i);
718         curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
719
720         if (curtailment_kW <= 0) {
721             continue;
722         }
723
724         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
725
726         if (acceptable_kW > curtailment_kW) {
727             acceptable_kW = curtailment_kW;
728         }
729
730         renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
731         renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
732         storage_ptr->power_kW += acceptable_kW;
733     }
734
735     // 4. commit charge
736     storage_ptr->commitCharge(
737         timestep,
738         dt_hrs,
739         storage_ptr->power_kW
740     );
741 }
742
743 return;
744 } /* __handleStorageCharging() */

```

4.3.3.11 __handleStorageCharging() [2/2]

```

void Controller::__handleStorageCharging (
    int timestep,
    double dt_hrs,
    std::vector< Storage * > * storage_ptr_vec_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
    std::vector< Renewable * > * renewable_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.
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion pointer vector of the Model .
<i>renewable_ptr_vec_ptr</i>	A pointer to the Renewable pointer vector of the Model .

```

787 {
788     double acceptable_kW = 0;
789     double curtailment_kW = 0;
790
791     Storage* storage_ptr;
792     Combustion* combustion_ptr;
793     Noncombustion* noncombustion_ptr;
794     Renewable* renewable_ptr;
795
796     for (size_t j = 0; j < storage_ptr_vec_ptr->size(); j++) {
797         storage_ptr = storage_ptr_vec_ptr->at(j);
798
799         // 1. attempt to charge from Combustion curtailment first
800         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
801             combustion_ptr = combustion_ptr_vec_ptr->at(i);
802             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
803
804             if (curtailment_kW <= 0) {
805                 continue;
806             }
807
808             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
809
810             if (acceptable_kW > curtailment_kW) {
811                 acceptable_kW = curtailment_kW;
812             }
813
814             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
815             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
816             storage_ptr->power_kW += acceptable_kW;
817         }
818
819         // 2. attempt to charge from Noncombustion curtailment second
820         for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
821             noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
822             curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
823
824             if (curtailment_kW <= 0) {
825                 continue;
826             }
827
828             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
829
830             if (acceptable_kW > curtailment_kW) {
831                 acceptable_kW = curtailment_kW;
832             }
833
834             noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
835             noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
836             storage_ptr->power_kW += acceptable_kW;
837         }
838
839         // 3. attempt to charge from Renewable curtailment third
840         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
841             renewable_ptr = renewable_ptr_vec_ptr->at(i);
842             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
843
844             if (curtailment_kW <= 0) {
845                 continue;
846             }
847
848             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
849
850             if (acceptable_kW > curtailment_kW) {
851                 acceptable_kW = curtailment_kW;
852             }
853
854             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
855             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
856             storage_ptr->power_kW += acceptable_kW;
857         }
858     }

```

```

859         // 4. commit charge
860         storage_ptr->commitCharge(
861             timestep,
862             dt_hrs,
863             storage_ptr->power_kW
864         );
865     }
866
867     return;
868 } /* __handleStorageCharging() */

```

4.3.3.12 __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.

```

1226 {
1227     double discharging_kW = 0;
1228
1229     Storage* storage_ptr;
1230
1231     std::list<Storage*>::iterator iter;
1232     for (
1233         iter = storage_ptr_list.begin();
1234         iter != storage_ptr_list.end();
1235         iter++
1236     ){
1237         storage_ptr = (*iter);
1238
1239         discharging_kW = storage_ptr->getAvailablekW(dt_hrs);
1240
1241         if (discharging_kW > net_load_kW) {
1242             discharging_kW = net_load_kW;
1243         }
1244
1245         net_load_kW = storage_ptr->commitDischarge(
1246             timestep,
1247             dt_hrs,
1248             discharging_kW,
1249             net_load_kW
1250         );
1251     }
1252
1253     return net_load_kW;
1254 } /* __handleStorageDischarging() */

```

4.3.3.13 applyDispatchControl()

```
void Controller::applyDispatchControl (
    ElectricalLoad * electrical_load_ptr,
    Resources * resources_ptr,
    std::vector< Combustion * > * combustion_ptr_vec_ptr,
    std::vector< Noncombustion * > * noncombustion_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>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 .
<i>noncombustion_ptr_vec_ptr</i>	A pointer to the Noncombustion 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 .

```
1406 {
1407     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
1408         switch (this->control_mode) {
1409             case (ControlMode :: LOAD_FOLLOWING): {
1410                 if (this->net_load_vec_kW[i] <= 0) {
1411                     this->__applyLoadFollowingControl_CHARGING(
1412                         i,
1413                         electrical_load_ptr,
1414                         resources_ptr,
1415                         combustion_ptr_vec_ptr,
1416                         noncombustion_ptr_vec_ptr,
1417                         renewable_ptr_vec_ptr,
1418                         storage_ptr_vec_ptr
1419                     );
1420                 }
1421             }
1422             else {
1423                 this->__applyLoadFollowingControl_DISCHARGING(
1424                     i,
1425                     electrical_load_ptr,
1426                     resources_ptr,
1427                     combustion_ptr_vec_ptr,
1428                     noncombustion_ptr_vec_ptr,
1429                     renewable_ptr_vec_ptr,
1430                     storage_ptr_vec_ptr
1431                 );
1432             }
1433             break;
1434         }
1435     }
1436
1437     case (ControlMode :: CYCLE_CHARGING): {
1438         if (this->net_load_vec_kW[i] <= 0) {
1439             this->__applyCycleChargingControl_CHARGING(
1440                 i,
1441                 electrical_load_ptr,
1442                 resources_ptr,
1443                 combustion_ptr_vec_ptr,
1444                 noncombustion_ptr_vec_ptr,
1445                 renewable_ptr_vec_ptr,
1446                 storage_ptr_vec_ptr
1447             );
1448         }
1449     }
1450     else {
1451         this->__applyCycleChargingControl_DISCHARGING(
1452             i,
1453             electrical_load_ptr,
1454             resources_ptr,
1455             combustion_ptr_vec_ptr,
1456             noncombustion_ptr_vec_ptr,
1457             renewable_ptr_vec_ptr,
1458             storage_ptr_vec_ptr
```



```

1459         );
1460     }
1461
1462     break;
1463 }
1464
1465     default: {
1466         std::string error_str = "ERROR: Controller :: applyDispatchControl(): ";
1467         error_str += "control mode ";
1468         error_str += std::to_string(this->control_mode);
1469         error_str += " not recognized";
1470
1471         #ifdef _WIN32
1472             std::cout << error_str << std::endl;
1473         #endif
1474
1475         throw std::runtime_error(error_str);
1476
1477         break;
1478     }
1479 }
1480 }
1481
1482     return;
1483 } /* applyDispatchControl() */

```

4.3.3.14 clear()

```

void Controller::clear (
    void )

```

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

```

1498 {
1499     this->net_load_vec_kW.clear();
1500     this->missed_load_vec_kW.clear();
1501     this->combustion_map.clear();
1502
1503     return;
1504 } /* clear() */

```

4.3.3.15 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 .

```

1356 {
1357     // 1. compute net load
1358     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);
1359 }

```

```

1360     // 2. construct Combustion table
1361     this->__constructCombustionMap(combustion_ptr_vec_ptr);
1362
1363     return;
1364 } /* init() */

```

4.3.3.16 setControlMode()

```

void Controller::setControlMode (
    ControlMode control_mode )

```

Parameters

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

```

1290 {
1291     this->control_mode = control_mode;
1292
1293     switch(control_mode) {
1294         case (ControlMode :: LOAD_FOLLOWING): {
1295             this->control_string = "LOAD_FOLLOWING";
1296
1297             break;
1298         }
1299
1300         case (ControlMode :: CYCLE_CHARGING): {
1301             this->control_string = "CYCLE_CHARGING";
1302
1303             break;
1304         }
1305
1306         default: {
1307             std::string error_str = "ERROR: Controller :: setControlMode(): ";
1308             error_str += "control mode ";
1309             error_str += std::to_string(control_mode);
1310             error_str += " not recognized";
1311
1312             #ifdef _WIN32
1313                 std::cout << error_str << std::endl;
1314             #endif
1315
1316             throw std::runtime_error(error_str);
1317
1318             break;
1319         }
1320     }
1321
1322     return;
1323 } /* 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:

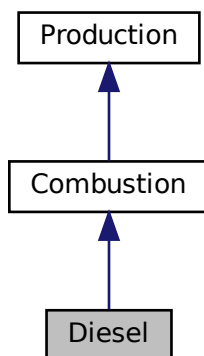
- header/[Controller.h](#)
- source/[Controller.cpp](#)

4.4 Diesel Class Reference

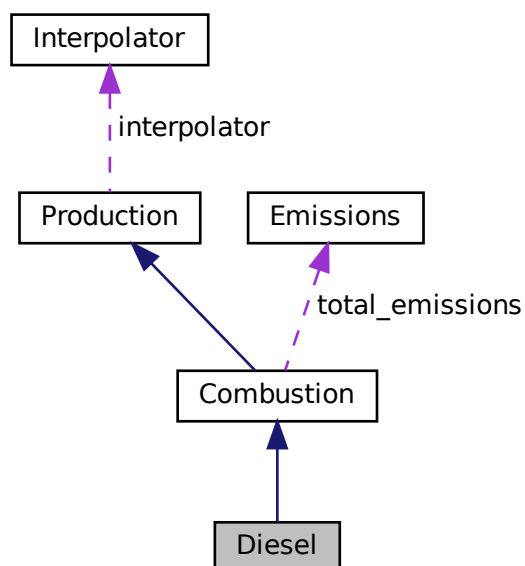
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

```
#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](#), std::vector< double > *)
Constructor (intended) for the [Diesel](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [requestProductionkW](#) (int, double, double)
Method which takes in production request, and then returns what the asset can deliver (subject to operating constraints, etc.).
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Diesel](#) (void)
Destructor for the [Diesel](#) class.

Public Attributes

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

Private Member Functions

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

4.4.1 Detailed Description

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

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Diesel() [1/2]

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

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

```
629 {
630     return;
631 } /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

```
Diesel::Diesel (
    int n_points,
    double n_years,
    DieselInputs diesel_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```
663 :
664 Combustion(
665     n_points,
666     n_years,
667     diesel_inputs.combustion_inputs,
668     time_vec_hrs_ptr
669 )
670 {
671     // 1. check inputs
672     this->__checkInputs(diesel_inputs);
673
674     // 2. set attributes
675     this->type = CombustionType :: DIESEL;
676     this->type_str = "DIESEL";
677
678     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
679
680     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
681
682     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
683     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
684     this->time_since_last_start_hrs = 0;
```

```

685
686     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
687     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
688     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
689     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
690     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
691     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
692
693     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
694         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
695     }
696     else {
697         this->linear_fuel_slope_LkWh = diesel_inputs.linear_fuel_slope_LkWh;
698     }
699
700     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
701         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
702     }
703     else {
704         this->linear_fuel_intercept_LkWh = diesel_inputs.linear_fuel_intercept_LkWh;
705     }
706
707     if (diesel_inputs.capital_cost < 0) {
708         this->capital_cost = this->__getGenericCapitalCost();
709     }
710     else {
711         this->capital_cost = diesel_inputs.capital_cost;
712     }
713
714     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
715         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
716     }
717     else {
718         this->operation_maintenance_cost_kWh =
719             diesel_inputs.operation_maintenance_cost_kWh;
720     }
721
722     if (not this->is_sunk) {
723         this->capital_cost_vec[0] = this->capital_cost;
724     }
725
726     // 3. construction print
727     if (this->print_flag) {
728         std::cout << "Diesel object constructed at " << this << std::endl;
729     }
730
731     return;
732 } /* Diesel() */

```

4.4.2.3 ~Diesel()

```

Diesel::~Diesel (
    void )

```

Destructor for the [Diesel](#) class.

```

894 {
895     // 1. destruction print
896     if (this->print_flag) {
897         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
898     }
899
900     return;
901 } /* ~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

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

```

64 {
65     // 1. check fuel_cost_L
66     if (diesel_inputs.fuel_cost_L < 0) {
67         std::string error_str = "ERROR: Diesel(): ";
68         error_str += "DieselInputs::fuel_cost_L 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     // 2. check CO2_emissions_intensity_kgL
78     if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
79         std::string error_str = "ERROR: Diesel(): ";
80         error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";
81
82         #ifdef _WIN32
83             std::cout << error_str << std::endl;
84         #endif
85
86         throw std::invalid_argument(error_str);
87     }
88
89     // 3. check CO_emissions_intensity_kgL
90     if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
91         std::string error_str = "ERROR: Diesel(): ";
92         error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";
93
94         #ifdef _WIN32
95             std::cout << error_str << std::endl;
96         #endif
97
98         throw std::invalid_argument(error_str);
99     }
100
101     // 4. check NOx_emissions_intensity_kgL
102     if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
103         std::string error_str = "ERROR: Diesel(): ";
104         error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";
105
106         #ifdef _WIN32
107             std::cout << error_str << std::endl;
108         #endif
109
110         throw std::invalid_argument(error_str);
111     }
112
113     // 5. check SOx_emissions_intensity_kgL
114     if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
115         std::string error_str = "ERROR: Diesel(): ";
116         error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";
117
118         #ifdef _WIN32
119             std::cout << error_str << std::endl;
120         #endif
121
122         throw std::invalid_argument(error_str);
123     }
124
125     // 6. check CH4_emissions_intensity_kgL
126     if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
127         std::string error_str = "ERROR: Diesel(): ";
128         error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";
129
130         #ifdef _WIN32
131             std::cout << error_str << std::endl;
132         #endif
133
134         throw std::invalid_argument(error_str);
135     }
136
137     // 7. check PM_emissions_intensity_kgL
138     if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
139         std::string error_str = "ERROR: Diesel(): ";
140         error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";
141
142         #ifdef _WIN32
143             std::cout << error_str << std::endl;
144         #endif
145

```



```

146         throw std::invalid_argument(error_str);
147     }
148
149     // 8. check minimum_load_ratio
150     if (diesel_inputs.minimum_load_ratio < 0) {
151         std::string error_str = "ERROR: Diesel(): ";
152         error_str += "DieselInputs::minimum_load_ratio must be >= 0";
153
154         #ifdef _WIN32
155             std::cout << error_str << std::endl;
156         #endif
157
158         throw std::invalid_argument(error_str);
159     }
160
161     // 9. check minimum_runtime_hrs
162     if (diesel_inputs.minimum_runtime_hrs < 0) {
163         std::string error_str = "ERROR: Diesel(): ";
164         error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";
165
166         #ifdef _WIN32
167             std::cout << error_str << std::endl;
168         #endif
169
170         throw std::invalid_argument(error_str);
171     }
172
173     // 10. check replace_running_hrs
174     if (diesel_inputs.replace_running_hrs <= 0) {
175         std::string error_str = "ERROR: Diesel(): ";
176         error_str += "DieselInputs::replace_running_hrs must be > 0";
177
178         #ifdef _WIN32
179             std::cout << error_str << std::endl;
180         #endif
181
182         throw std::invalid_argument(error_str);
183     }
184
185     return;
186 } /* __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].

```

263 {
264     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.425) + 800;
265
266     return capital_cost_per_kW * this->capacity_kW;
267 } /* __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].

```
238 {
239     double linear_fuel_intercept_LkWh = 0.0940 * pow(this->capacity_kW, -0.2735);
240
241     return linear_fuel_intercept_LkWh;
242 } /* __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].

```
210 {
211     double linear_fuel_slope_LkWh = 0.4234 * pow(this->capacity_kW, -0.1012);
212
213     return linear_fuel_slope_LkWh;
214 } /* __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].

```
291 {
292     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
293
294     return operation_maintenance_cost_kWh;
295 } /* __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.

```
325 {
326     /*
327     * Helper method (private) to handle the starting/stopping of the diesel
328     * generator. The minimum runtime constraint is enforced in this method.
329     */
330
331     if (this->is_running) {
332         // handle stopping
333         if (
334             production_kW <= 0 and
335             this->time_since_last_start_hrs >= this->minimum_runtime_hrs
336         ) {
337             this->is_running = false;
338         }
339     }
340
341     else {
342         // handle starting
343         if (production_kW > 0) {
344             this->is_running = true;
345             this->n_starts++;
346             this->time_since_last_start_hrs = 0;
347         }
348     }
349 }
```

```

350     return;
351 } /* __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](#).

```

370 {
371     // 1. create filestream
372     write_path += "summary_results.md";
373     std::ofstream ofs;
374     ofs.open(write_path, std::ofstream::out);
375
376     // 2. write to summary results (markdown)
377     ofs << "# ";
378     ofs << std::to_string(int(ceil(this->capacity_kW)));
379     ofs << " kW DIESEL Summary Results\n";
380     ofs << "\n-----\n\n";
381
382     // 2.1. Production attributes
383     ofs << "## Production Attributes\n";
384     ofs << "\n";
385
386     ofs << "Capacity: " << this->capacity_kW << " kW \n";
387     ofs << "\n";
388
389     ofs << "Production Override: (N = 0 / Y = 1): "
390         << this->normalized_production_series_given << " \n";
391     if (this->normalized_production_series_given) {
392         ofs << "Path to Normalized Production Time Series: "
393             << this->path_2_normalized_production_time_series << " \n";
394     }
395     ofs << "\n";
396
397     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
398     ofs << "Capital Cost: " << this->capital_cost << " \n";
399     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
400         << " per kWh produced \n";
401     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
402         << " \n";
403     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
404         << " \n";
405     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
406     ofs << "\n";
407
408     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
409     ofs << "\n-----\n\n";
410
411     // 2.2. Combustion attributes
412     ofs << "## Combustion Attributes\n";
413     ofs << "\n";
414
415     ofs << "Fuel Cost: " << this->fuel_cost_L << " per L \n";
416     ofs << "Nominal Fuel Escalation Rate (annual): "
417         << this->nominal_fuel_escalation_annual << " \n";
418     ofs << "Real Fuel Escalation Rate (annual): "
419         << this->real_fuel_escalation_annual << " \n";
420     ofs << "\n";
421
422     ofs << "Fuel Mode: " << this->fuel_mode_str << " \n";
423     switch (this->fuel_mode) {
424     case (FuelMode :: FUEL_MODE_LINEAR): {
425         ofs << "Linear Fuel Slope: " << this->linear_fuel_slope_LkWh

```

```

426         « " L/kWh \n";
427     ofs « "Linear Fuel Intercept Coefficient: "
428         « this->linear_fuel_intercept_LkWh « " L/kWh \n";
429     ofs « "\n";
430
431     break;
432 }
433
434 case (FuelMode :: FUEL_MODE_LOOKUP): {
435     ofs « "Fuel Consumption Data: " « this->interpolator.path_map_1D[0]
436         « " \n";
437
438     break;
439 }
440
441 default: {
442     // write nothing!
443
444     break;
445 }
446 }
447
448 ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
449     « this->CO2_emissions_intensity_kgL « " kg/L \n";
450
451 ofs « "Carbon Monoxide (CO) Emissions Intensity: "
452     « this->CO_emissions_intensity_kgL « " kg/L \n";
453
454 ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
455     « this->NOx_emissions_intensity_kgL « " kg/L \n";
456
457 ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
458     « this->SOx_emissions_intensity_kgL « " kg/L \n";
459
460 ofs « "Methane (CH4) Emissions Intensity: "
461     « this->CH4_emissions_intensity_kgL « " kg/L \n";
462
463 ofs « "Particulate Matter (PM) Emissions Intensity: "
464     « this->PM_emissions_intensity_kgL « " kg/L \n";
465
466 ofs « "\n-----\n\n";
467
468 // 2.3. Diesel attributes
469 ofs « "## Diesel Attributes\n";
470 ofs « "\n";
471
472 ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « " \n";
473 ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs \n";
474
475 ofs « "\n-----\n\n";
476
477 // 2.4. Diesel Results
478 ofs « "## Results\n";
479 ofs « "\n";
480
481 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
482 ofs « "\n";
483
484 ofs « "Total Dispatch: " « this->total_dispatch_kWh
485     « " kWh \n";
486
487 ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
488     « " per kWh dispatched \n";
489 ofs « "\n";
490
491 ofs « "Running Hours: " « this->running_hours « " \n";
492 ofs « "Starts: " « this->n_starts « " \n";
493 ofs « "Replacements: " « this->n_replacements « " \n";
494
495 ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
496     « "(Annual Average: " « this->total_fuel_consumed_L / this->n_years
497     « " L/yr) \n";
498 ofs « "\n";
499
500 ofs « "Total Carbon Dioxide (CO2) Emissions: " «
501     this->total_emissions.CO2_kg « " kg "
502     « "(Annual Average: " « this->total_emissions.CO2_kg / this->n_years
503     « " kg/yr) \n";
504
505 ofs « "Total Carbon Monoxide (CO) Emissions: " «
506     this->total_emissions.CO_kg « " kg "
507     « "(Annual Average: " « this->total_emissions.CO_kg / this->n_years
508     « " kg/yr) \n";
509
510 ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
511     this->total_emissions.NOx_kg « " kg "
512     « "(Annual Average: " « this->total_emissions.NOx_kg / this->n_years

```

```

513         « " kg/yr)  \n";
514
515     ofs « "Total Sulfur Oxides (SOx) Emissions: " «
516         this->total_emissions.SOx_kg « " kg "
517         « "(Annual Average: " « this->total_emissions.SOx_kg / this->n_years
518         « " kg/yr)  \n";
519
520     ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
521         « "(Annual Average: " « this->total_emissions.CH4_kg / this->n_years
522         « " kg/yr)  \n";
523
524     ofs « "Total Particulate Matter (PM) Emissions: " «
525         this->total_emissions.PM_kg « " kg "
526         « "(Annual Average: " « this->total_emissions.PM_kg / this->n_years
527         « " kg/yr)  \n";
528
529     ofs « "\n-----\n\n";
530
531     ofs.close();
532     return;
533 } /* __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 <i>time_vec_hrs</i> attribute of the ElectricalLoad .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Combustion](#).

```

563 {
564     // 1. create filestream
565     write_path += "time_series_results.csv";
566     std::ofstream ofs;
567     ofs.open(write_path, std::ofstream::out);
568
569     // 2. write time series results (comma separated value)
570     ofs « "Time (since start of data) [hrs],";
571     ofs « "Production [kW],";
572     ofs « "Dispatch [kW],";
573     ofs « "Storage [kW],";
574     ofs « "Curtailement [kW],";
575     ofs « "Is Running (N = 0 / Y = 1),";
576     ofs « "Fuel Consumption [L],";
577     ofs « "Fuel Cost (actual),";
578     ofs « "Carbon Dioxide (CO2) Emissions [kg],";
579     ofs « "Carbon Monoxide (CO) Emissions [kg],";
580     ofs « "Nitrogen Oxides (NOx) Emissions [kg],";
581     ofs « "Sulfur Oxides (SOx) Emissions [kg],";
582     ofs « "Methane (CH4) Emissions [kg],";
583     ofs « "Particulate Matter (PM) Emissions [kg],";
584     ofs « "Capital Cost (actual),";
585     ofs « "Operation and Maintenance Cost (actual),";
586     ofs « "\n";
587
588     for (int i = 0; i < max_lines; i++) {
589         ofs « time_vec_hrs_ptr->at(i) « ",";
590         ofs « this->production_vec_kW[i] « ",";
591         ofs « this->dispatch_vec_kW[i] « ",";
592         ofs « this->storage_vec_kW[i] « ",";
593         ofs « this->curtailement_vec_kW[i] « ",";

```

```

594         ofs « this->is_running_vec[i] « ", ";
595         ofs « this->fuel_consumption_vec_L[i] « ", ";
596         ofs « this->fuel_cost_vec[i] « ", ";
597         ofs « this->CO2_emissions_vec_kg[i] « ", ";
598         ofs « this->CO_emissions_vec_kg[i] « ", ";
599         ofs « this->NOx_emissions_vec_kg[i] « ", ";
600         ofs « this->SOx_emissions_vec_kg[i] « ", ";
601         ofs « this->CH4_emissions_vec_kg[i] « ", ";
602         ofs « this->PM_emissions_vec_kg[i] « ", ";
603         ofs « this->capital_cost_vec[i] « ", ";
604         ofs « this->operation_maintenance_cost_vec[i] « ", ";
605         ofs « "\n";
606     }
607
608     ofs.close();
609     return;
610 } /* __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](#).

```

852 {
853     // 1. handle start/stop, enforce minimum runtime constraint
854     this->__handleStartStop(timestep, dt_hrs, production_kW);
855
856     // 2. invoke base class method
857     load_kW = Combustion::commit(
858         timestep,
859         dt_hrs,
860         production_kW,
861         load_kW
862     );
863
864     if (this->is_running) {
865         // 3. log time since last start
866         this->time_since_last_start_hrs += dt_hrs;
867
868         // 4. correct operation and maintenance costs (should be non-zero if idling)
869         if (production_kW <= 0) {
870             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
871
872             double operation_maintenance_cost =
873                 this->operation_maintenance_cost_kWh * produced_kWh;
874             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
875         }
876     }
877 }

```

```

876     }
877
878     return load_kW;
879 } /* commit() */

```

4.4.3.10 handleReplacement()

```

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

```

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

Parameters

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

Reimplemented from [Combustion](#).

```

750 {
751     // 1. reset attributes
752     this->time_since_last_start_hrs = 0;
753
754     // 2. invoke base class method
755     Combustion :: handleReplacement (timestep);
756
757     return;
758 } /* __handleReplacement() */

```

4.4.3.11 requestProductionkW()

```

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

```

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

Parameters

<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](#).

```

790 {
791     // 0. given production time series override
792     if (this->normalized_production_series_given) {
793         double production_kW = Production :: getProductionkW(timestep);

```



```
794
795     return production_kW;
796 }
797
798 // 1. return on request of zero
799 if (request_kW <= 0) {
800     return 0;
801 }
802
803 double deliver_kW = request_kW;
804
805 // 2. enforce capacity constraint
806 if (deliver_kW > this->capacity_kW) {
807     deliver_kW = this->capacity_kW;
808 }
809
810 // 3. enforce minimum load ratio
811 if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
812     deliver_kW = this->minimum_load_ratio * this->capacity_kW;
813 }
814
815 return deliver_kW;
816 } /* requestProductionkW() */
```

4.4.4 Member Data Documentation

4.4.4.1 minimum_load_ratio

```
double Diesel::minimum_load_ratio
```

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

4.4.4.2 minimum_runtime_hrs

```
double Diesel::minimum_runtime_hrs
```

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

4.4.4.3 time_since_last_start_hrs

```
double Diesel::time_since_last_start_hrs
```

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

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

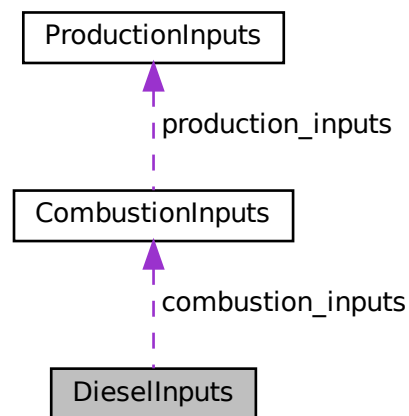
- header/Production/Combustion/[Diesel.h](#)
- source/Production/Combustion/[Diesel.cpp](#)

4.5 DieselInputs Struct Reference

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

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

Collaboration diagram for DieselInputs:



Public Attributes

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

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

- double `linear_fuel_intercept_LkWh` = -1

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

- double `CO2_emissions_intensity_kgL` = 2.7

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

- double `CO_emissions_intensity_kgL` = 0.0178

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

- double `NOx_emissions_intensity_kgL` = 0.0014

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

- double `SOx_emissions_intensity_kgL` = 0.0042

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

- double `CH4_emissions_intensity_kgL` = 0.0007

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

- double `PM_emissions_intensity_kgL` = 0.0001

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

4.5.1 Detailed Description

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

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

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

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

Ref: [NRCan \[2014\]](#)

Ref: [CIMAC \[2008\]](#)

4.5.2 Member Data Documentation

4.5.2.1 capital_cost

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

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

4.5.2.2 CH4_emissions_intensity_kgL

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

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

4.5.2.3 CO2_emissions_intensity_kgL

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

Carbon dioxide (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.

```
62 {
63     return;
64 } /* 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.
---	--

```
82 {
83     this->readLoadData(path_2_electrical_load_time_series);
84
85     return;
86 } /* ElectricalLoad() */
```

4.6.2.3 ~ElectricalLoad()

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

Destructor for the [ElectricalLoad](#) class.

```
209 {
210     this->clear();
211     return;
212 } /* ~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.

```
182 {
183     this->n_points = 0;
184     this->n_years = 0;
185     this->min_load_kW = 0;
186     this->mean_load_kW = 0;
187     this->max_load_kW = 0;
188
189     this->path_2_electrical_load_time_series.clear();
190     this->time_vec_hrs.clear();
191     this->dt_vec_hrs.clear();
192     this->load_vec_kW.clear();
193
194     return;
195 } /* 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.
---	--

```
104 {
105     // 1. clear
106     this->clear();
107
108     // 2. init CSV reader, record path
109     io::CSVReader<2> CSV(path_2_electrical_load_time_series);
110
111     CSV.read_header(
112         io::ignore_extra_column,
113         "Time (since start of data) [hrs]",
114         "Electrical Load [kW]"
115     );
116
117     this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;
118
119     // 3. read in time and load data, increment n_points, track min and max load
120     double time_hrs = 0;
121     double load_kW = 0;
122     double load_sum_kW = 0;
123
124     this->n_points = 0;
125
126     this->min_load_kW = std::numeric_limits<double>::infinity();
127     this->max_load_kW = -1 * std::numeric_limits<double>::infinity();
128
129     while (CSV.read_row(time_hrs, load_kW)) {
130         this->time_vec_hrs.push_back(time_hrs);
131         this->load_vec_kW.push_back(load_kW);
132
133         load_sum_kW += load_kW;
134
135         this->n_points++;
136
137         if (this->min_load_kW > load_kW) {
138             this->min_load_kW = load_kW;
139         }
140     }
```



```

141         if (this->max_load_kW < load_kW) {
142             this->max_load_kW = load_kW;
143         }
144     }
145
146     // 4. compute mean load
147     this->mean_load_kW = load_sum_kW / this->n_points;
148
149     // 5. set number of years (assuming 8,760 hours per year)
150     this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;
151
152     // 6. populate dt_vec_hrs
153     this->dt_vec_hrs.resize(n_points, 0);
154
155     for (int i = 0; i < n_points; i++) {
156         if (i == n_points - 1) {
157             this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
158         }
159         else {
160             double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];
161             this->dt_vec_hrs[i] = dt_hrs;
162         }
163     }
164
165     }
166
167     return;
168 } /* 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:

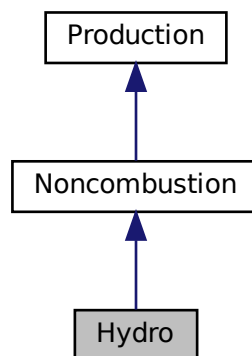
- [header/Production/Combustion/Combustion.h](#)

4.8 Hydro Class Reference

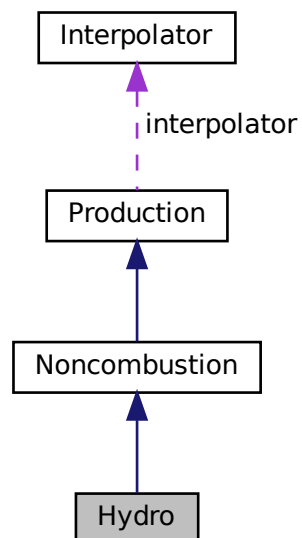
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

```
#include <Hydro.h>
```

Inheritance diagram for Hydro:



Collaboration diagram for Hydro:



Public Member Functions

- [Hydro](#) (void)
Constructor (dummy) for the [Hydro](#) class.
- [Hydro](#) (int, double, [HydroInputs](#), std::vector< double > *)
Constructor (intended) for the [Hydro](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [requestProductionkW](#) (int, double, 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, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Hydro](#) (void)
Destructor for the [Hydro](#) class.

Public Attributes

- [HydroTurbineType](#) turbine_type
The type of hydroelectric turbine model to use.
- double [fluid_density_kgm3](#)
The density [kg/m3] of the hydroelectric working fluid.
- double [net_head_m](#)
The net head [m] of the asset.
- double [reservoir_capacity_m3](#)
The capacity [m3] of the hydro reservoir.
- double [init_reservoir_state](#)
The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- double [stored_volume_m3](#)
The volume [m3] of stored fluid.
- double [minimum_power_kW](#)
The minimum power [kW] that the asset can produce. Corresponds to minimum productive flow.
- double [minimum_flow_m3hr](#)
The minimum required flow [m3/hr] for the asset to produce. Corresponds to minimum power.
- double [maximum_flow_m3hr](#)
The maximum productive flow [m3/hr] that the asset can support.
- std::vector< double > [turbine_flow_vec_m3hr](#)
A vector of the turbine flow [m3/hr] at each point in the modelling time series.
- std::vector< double > [spill_rate_vec_m3hr](#)
A vector of the spill rate [m3/hr] at each point in the modelling time series.
- std::vector< double > [stored_volume_vec_m3](#)
A vector of the stored volume [m3] in the reservoir at each point in the modelling time series.

Private Member Functions

- void [__checkInputs](#) ([HydroInputs](#))
Helper method to check inputs to the [Hydro](#) constructor.
- void [__initInterpolator](#) (void)
Helper method to set up turbine and generator efficiency interpolation.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic hydroelectric capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method (private) to generate a generic hydroelectric operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__getEfficiencyFactor](#) (double)
Helper method to compute the efficiency factor (product of turbine and generator efficiencies).
- double [__getMinimumFlowm3hr](#) (void)
Helper method to compute and return the minimum required flow for production, based on turbine type.
- double [__getMaximumFlowm3hr](#) (void)
Helper method to compute and return the maximum productive flow, based on turbine type.
- double [__flowToPower](#) (double)
Helper method to translate a given flow into a corresponding power output.
- double [__powerToFlow](#) (double)
Helper method to translate a given power output into a corresponding flow.
- double [__getAvailableFlow](#) (double, double)
Helper method to determine what flow is currently available to the turbine.
- double [__getAcceptableFlow](#) (double)
Helper method to determine what flow is currently acceptable by the reservoir.
- void [__updateState](#) (int, double, double, double)
Helper method to update and log flow and reservoir state.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Hydro](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, int=-1)
Helper method to write time series results for [Hydro](#).

4.8.1 Detailed Description

A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

4.8.2 Constructor & Destructor Documentation

4.8.2.1 [Hydro\(\)](#) [1/2]

```
Hydro::Hydro (
    void )
```

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

```
859 {
860     return;
861 } /* Hydro() */
```

4.8.2.2 Hydro() [2/2]

```
Hydro::Hydro (
    int n_points,
    double n_years,
    HydroInputs hydro_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the [Hydro](#) 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>hydro_inputs</i>	A structure of Hydro constructor inputs.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```
893 :
894 Noncombustion(
895     n_points,
896     n_years,
897     hydro_inputs.noncombustion_inputs,
898     time_vec_hrs_ptr
899 )
900 {
901     // 1. check inputs
902     this->__checkInputs(hydro_inputs);
903
904     // 2. set attributes
905     this->type = NoncombustionType :: HYDRO;
906     this->type_str = "HYDRO";
907
908     this->resource_key = hydro_inputs.resource_key;
909
910     this->turbine_type = hydro_inputs.turbine_type;
911
912     this->fluid_density_kgm3 = hydro_inputs.fluid_density_kgm3;
913     this->net_head_m = hydro_inputs.net_head_m;
914
915     this->reservoir_capacity_m3 = hydro_inputs.reservoir_capacity_m3;
916     this->init_reservoir_state = hydro_inputs.init_reservoir_state;
917     this->stored_volume_m3 =
918         hydro_inputs.init_reservoir_state * hydro_inputs.reservoir_capacity_m3;
919
920     this->minimum_power_kW = 0.1 * this->capacity_kW;    // <-- NEED TO DOUBLE CHECK THAT THIS MAKES
SENSE IN GENERAL
921
922     this->__initInterpolator();
923
924     this->minimum_flow_m3hr = this->__getMinimumFlowm3hr();
925     this->maximum_flow_m3hr = this->__getMaximumFlowm3hr();
926
927     this->turbine_flow_vec_m3hr.resize(this->n_points, 0);
928     this->spill_rate_vec_m3hr.resize(this->n_points, 0);
929     this->stored_volume_vec_m3.resize(this->n_points, 0);
930
931     if (hydro_inputs.capital_cost < 0) {
932         this->capital_cost = this->__getGenericCapitalCost();
933     }
934     else {
935         this->capital_cost = hydro_inputs.capital_cost;
936     }
937
938     if (hydro_inputs.operation_maintenance_cost_kWh < 0) {
939         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
940     }
941     else {
942         this->operation_maintenance_cost_kWh =
943             hydro_inputs.operation_maintenance_cost_kWh;
944     }
945
946     if (not this->is_sunk) {
947         this->capital_cost_vec[0] = this->capital_cost;
948     }
949
950     return;
951 } /* Hydro() */
```


4.8.2.3 ~Hydro()

```
Hydro::~~Hydro (
    void )
```

Destructor for the [Hydro](#) class.

```
1125 {
1126     // 1. destruction print
1127     if (this->print_flag) {
1128         std::cout << "Hydro object at " << this << " destroyed" << std::endl;
1129     }
1130
1131     return;
1132 } /* ~Hydro() */
```

4.8.3 Member Function Documentation

4.8.3.1 __checkInputs()

```
void Hydro::__checkInputs (
    HydroInputs hydro_inputs ) [private]
```

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

Parameters

<i>hydro_inputs</i>	A structure of Hydro constructor inputs.
---------------------	--

```
64 {
65     // 1. check fluid_density_kgm3
66     if (hydro_inputs.fluid_density_kgm3 <= 0) {
67         std::string error_str = "ERROR: Hydro(): fluid_density_kgm3 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     // 2. check net_head_m
77     if (hydro_inputs.net_head_m <= 0) {
78         std::string error_str = "ERROR: Hydro(): net_head_m must be > 0";
79
80         #ifdef _WIN32
81             std::cout << error_str << std::endl;
82         #endif
83
84         throw std::invalid_argument(error_str);
85     }
86
87     // 3. check reservoir_capacity_m3
88     if (hydro_inputs.reservoir_capacity_m3 < 0) {
89         std::string error_str = "ERROR: Hydro(): reservoir_capacity_m3 must be >= 0";
90
91         #ifdef _WIN32
92             std::cout << error_str << std::endl;
93         #endif
94
95         throw std::invalid_argument(error_str);
96     }
97 }
```

```

98     // 4. check init_reservoir_state
99     if (
100         hydro_inputs.init_reservoir_state < 0 or
101         hydro_inputs.init_reservoir_state > 1
102     ) {
103         std::string error_str = "ERROR: Hydro(): init_reservoir_state must be in ";
104         error_str += "the closed interval [0, 1]";
105
106         #ifdef _WIN32
107             std::cout << error_str << std::endl;
108         #endif
109
110         throw std::invalid_argument(error_str);
111     }
112
113     return;
114 } /* __checkInputs() */

```

4.8.3.2 __flowToPower()

```

double Hydro::__flowToPower (
    double flow_m3hr ) [private]

```

Helper method to translate a given flow into a corresponding power output.

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

Parameters

<i>flow_m3hr</i>	The flow [m3/hr] through the turbine.
------------------	---------------------------------------

Returns

The power output [kW] corresponding to a given flow [m3/hr].

```

452 {
453     // 1. return on less than minimum flow
454     if (flow_m3hr < this->minimum_flow_m3hr) {
455         return 0;
456     }
457
458     // 2. interpolate flow to power
459     double power_kW = this->interpolator.interp1D(
460         HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
461         flow_m3hr
462     );
463
464     return power_kW;
465 } /* __flowToPower() */

```

4.8.3.3 __getAcceptableFlow()

```

double Hydro::__getAcceptableFlow (
    double dt_hrs ) [private]

```

Helper method to determine what flow is currently acceptable by the reservoir.

Parameters

<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
---------------	--

Returns

The flow [m3/hr] currently acceptable by the reservoir.

```

554 {
555     // 1. if no reservoir, return
556     if (this->reservoir_capacity_m3 <= 0) {
557         return 0;
558     }
559
560     // 2. compute acceptable based on room in reservoir
561     double acceptable_m3hr = (this->reservoir_capacity_m3 - this->stored_volume_m3) /
562         dt_hrs;
563
564     return acceptable_m3hr;
565 } /* __getAcceptableFlow() */

```

4.8.3.4 __getAvailableFlow()

```

double Hydro::__getAvailableFlow (
    double dt_hrs,
    double hydro_resource_m3hr ) [private]

```

Helper method to determine what flow is currently available to the turbine.

Parameters

<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
<i>hydro_resource_m3hr</i>	The currently available hydro flow resource [m3/hr].

Returns

The flow [m3/hr] currently available through the turbine.

```

521 {
522     // 1. init to flow available from stored volume in reservoir
523     double flow_m3hr = this->stored_volume_m3 / dt_hrs;
524
525     // 2. add flow available from resource
526     flow_m3hr += hydro_resource_m3hr;
527
528     // 3. cap at maximum flow
529     if (flow_m3hr > this->maximum_flow_m3hr) {
530         flow_m3hr = this->maximum_flow_m3hr;
531     }
532
533     return flow_m3hr;
534 } /* __getAvailableFlow() */

```

4.8.3.5 __getEfficiencyFactor()

```

double Hydro::__getEfficiencyFactor (
    double power_kW ) [private]

```

Helper method to compute the efficiency factor (product of turbine and generator efficiencies).

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

Parameters

<code>power_kW</code>	The power requested of the hydro plant.
-----------------------	---

Returns

The product of the turbine and generator efficiencies.

```

350 {
351     // 1. return on zero
352     if (power_kW <= 0) {
353         return 0;
354     }
355
356     // 2. compute power ratio (clip to [0, 1])
357     double power_ratio = power_kW / this->capacity_kW;
358
359     if (power_ratio < 0) {
360         power_ratio = 0;
361     }
362
363     else if (power_ratio > 1) {
364         power_ratio = 1;
365     }
366
367
368     // 3. init efficiency factor to the turbine efficiency
369     double efficiency_factor = this->interpolator.interp1D(
370         HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
371         power_ratio
372     );
373
374     // 4. include generator efficiency
375     efficiency_factor *= this->interpolator.interp1D(
376         HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
377         power_ratio
378     );
379
380     return efficiency_factor;
381 } /* __getEfficiencyFactor() */

```

4.8.3.6 __getGenericCapitalCost()

```

double Hydro::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic hydroelectric capital cost.

This model was obtained by way of ...

Returns

A generic capital cost for the hydroelectric asset [CAD].

```

299 {
300     double capital_cost_per_kW = 1000; //<-- WIP: need something better here!
301
302     return capital_cost_per_kW * this->capacity_kW + 15000000; //<-- WIP: need something better here!
303 } /* __getGenericCapitalCost() */

```

4.8.3.7 `__getGenericOpMaintCost()`

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

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

This model was obtained by way of ...

Returns

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

```
324 {
325     double operation_maintenance_cost_kWh = 0.05;  //<-- WIP: need something better here!
326
327     return operation_maintenance_cost_kWh;
328 } /* __getGenericOpMaintCost() */
```

4.8.3.8 `__getMaximumFlowm3hr()`

```
double Hydro::__getMaximumFlowm3hr (
    void ) [private]
```

Helper method to compute and return the maximum productive flow, based on turbine type.

This helper method assumes that the maximum flow is that which is associated with a power ratio of 1.

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

Returns

The maximum productive flow [m3/hr].

```
429 {
430     return this->__powerToFlow(this->capacity_kW);
431 } /* __getMaximumFlowm3hr() */
```

4.8.3.9 `__getMinimumFlowm3hr()`

```
double Hydro::__getMinimumFlowm3hr (
    void ) [private]
```

Helper method to compute and return the minimum required flow for production, based on turbine type.

This helper method assumes that the minimum flow is that which is associated with a power ratio of 0.1. See constructor for initialization of `minimum_power_kW`.

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

Returns

The minimum required flow [m3/hr] for production.

```
404 {
405     return this->__powerToFlow(this->minimum_power_kW);
406 } /* __getMinimumFlowm3hr() */
```

4.8.3.10 __initInterpolator()

```
void Hydro::__initInterpolator (
    void ) [private]
```

Helper method to set up turbine and generator efficiency interpolation.

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

```
131 {
132     // 1. set up generator efficiency interpolation
133     InterpolatorStruct1D generator_interp_struct_1D;
134
135     generator_interp_struct_1D.n_points = 12;
136
137     generator_interp_struct_1D.x_vec = {
138         0, 0.1, 0.2, 0.3, 0.4, 0.5,
139         0.6, 0.7, 0.75, 0.8, 0.9, 1
140     };
141
142     generator_interp_struct_1D.min_x = 0;
143     generator_interp_struct_1D.max_x = 1;
144
145     generator_interp_struct_1D.y_vec = {
146         0.000, 0.800, 0.900, 0.913,
147         0.925, 0.943, 0.947, 0.950,
148         0.953, 0.954, 0.956, 0.958
149     };
150
151     this->interpolator.interp_map_1D.insert(
152         std::pair<int, InterpolatorStruct1D>{
153             HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
154             generator_interp_struct_1D
155         }
156     );
157
158     // 2. set up turbine efficiency interpolation
159     InterpolatorStruct1D turbine_interp_struct_1D;
160
161     turbine_interp_struct_1D.n_points = 11;
162
163     turbine_interp_struct_1D.x_vec = {
164         0, 0.1, 0.2, 0.3, 0.4,
165         0.5, 0.6, 0.7, 0.8, 0.9,
166         1
167     };
168
169     turbine_interp_struct_1D.min_x = 0;
170     turbine_interp_struct_1D.max_x = 1;
171
172     std::vector<double> efficiency_vec;
173
174     switch (this->turbine_type) {
175     case (HydroTurbineType :: HYDRO_TURBINE_PELTON): {
176         efficiency_vec = {
177             0.000, 0.780, 0.855, 0.875, 0.890,
178             0.900, 0.908, 0.913, 0.918, 0.908,
179             0.880
180         };
181         break;
182     }
183
184     case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
185         efficiency_vec = {
186             0.000, 0.400, 0.625, 0.745, 0.810,
187             0.845, 0.880, 0.900, 0.910, 0.900,
188             0.850
189         };
190         break;
191     }
192
193     case (HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
194         efficiency_vec = {
195             0.000, 0.265, 0.460, 0.550, 0.650,
196             0.740, 0.805, 0.845, 0.900, 0.880,
197             0.850
198         };
199         break;
200     }
201
202     }
203 }
```

```

204
205     default: {
206         std::string error_str = "ERROR: Hydro(): turbine type ";
207         error_str += std::to_string(this->turbine_type);
208         error_str += " not recognized";
209
210         #ifdef _WIN32
211             std::cout << error_str << std::endl;
212         #endif
213
214         throw std::runtime_error(error_str);
215
216         break;
217     }
218 }
219
220 turbine_interp_struct_1D.y_vec = efficiency_vec;
221
222 this->interpolator.interp_map_1D.insert(
223     std::pair<int, InterpolatorStruct1D>(
224         HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
225         turbine_interp_struct_1D
226     )
227 );
228
229 // 3. set up flow to power interpolation
230 InterpolatorStruct1D flow_to_power_interp_struct_1D;
231
232 double power_ratio = 0.1;
233 std::vector<double> power_ratio_vec (91, 0);
234
235 for (size_t i = 0; i < power_ratio_vec.size(); i++) {
236     power_ratio_vec[i] = power_ratio;
237
238     power_ratio += 0.01;
239
240     if (power_ratio < 0) {
241         power_ratio = 0;
242     }
243
244     else if (power_ratio > 1) {
245         power_ratio = 1;
246     }
247 }
248
249 flow_to_power_interp_struct_1D.n_points = power_ratio_vec.size();
250
251 std::vector<double> flow_vec_m3hr;
252 std::vector<double> power_vec_kW;
253 flow_vec_m3hr.resize(power_ratio_vec.size(), 0);
254 power_vec_kW.resize(power_ratio_vec.size(), 0);
255
256 for (size_t i = 0; i < power_ratio_vec.size(); i++) {
257     flow_vec_m3hr[i] = this->__powerToFlow(power_ratio_vec[i] * this->capacity_kW);
258     power_vec_kW[i] = power_ratio_vec[i] * this->capacity_kW;
259     /*
260     std::cout << flow_vec_m3hr[i] << "\t" << power_vec_kW[i] << " (" <<
261         power_ratio_vec[i] << ")" << std::endl;
262     */
263 }
264
265 flow_to_power_interp_struct_1D.x_vec = flow_vec_m3hr;
266
267 flow_to_power_interp_struct_1D.min_x = flow_vec_m3hr[0];
268 flow_to_power_interp_struct_1D.max_x = flow_vec_m3hr[flow_vec_m3hr.size() - 1];
269
270 flow_to_power_interp_struct_1D.y_vec = power_vec_kW;
271
272 this->interpolator.interp_map_1D.insert(
273     std::pair<int, InterpolatorStruct1D>(
274         HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
275         flow_to_power_interp_struct_1D
276     )
277 );
278
279 return;
280 } /* __initInterpolator() */

```

4.8.3.11 __powerToFlow()

```

double Hydro::__powerToFlow (
    double power_kW ) [private]

```

Helper method to translate a given power output into a corresponding flow.

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

Parameters

<i>power_kW</i>	The power output [kW] of the hydroelectric generator.
-----------------	---

Returns

```

486 {
487     // 1. return on zero power
488     if (power_kW <= 0) {
489         return 0;
490     }
491
492     // 2. get efficiency factor
493     double efficiency_factor = this->__getEfficiencyFactor(power_kW);
494
495     // 3. compute flow
496     double flow_m3hr = 3600 * 1000 * power_kW;
497     flow_m3hr /= efficiency_factor * this->fluid_density_kgm3 * 9.81 * this->net_head_m;
498
499     return flow_m3hr;
500 } /* __powerToFlow() */

```

4.8.3.12 __updateState()

```

void Hydro::__updateState (
    int timestep,
    double dt_hrs,
    double production_kW,
    double hydro_resource_m3hr ) [private]

```

Helper method to update and log flow and reservoir state.

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>hydro_resource_m3hr</i>	The currently available hydro flow resource [m3/hr].

```

598 {
599     // 1. get turbine flow, log
600     double flow_m3hr = 0;
601
602     if (production_kW >= this->minimum_power_kW) {
603         flow_m3hr = this->__powerToFlow(production_kW);
604     }
605
606     double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);
607
608     if (flow_m3hr > available_flow_m3hr) {
609         flow_m3hr = available_flow_m3hr;
610     }
611
612     this->turbine_flow_vec_m3hr[timestep] = flow_m3hr;
613
614     // 3. compute net reservoir flow

```



```

615     double net_flow_m3hr = hydro_resource_m3hr - flow_m3hr;
616
617     // 4. compute flow acceptable by reservoir
618     double acceptable_flow_m3hr = this->__getAcceptableFlow(dt_hrs);
619
620     // 5. compute spill, update net flow (if applicable), log
621     double spill_m3hr = 0;
622
623     if (acceptable_flow_m3hr < net_flow_m3hr) {
624         spill_m3hr = net_flow_m3hr - acceptable_flow_m3hr;
625         net_flow_m3hr = acceptable_flow_m3hr;
626     }
627
628     this->spill_rate_vec_m3hr[timestep] = spill_m3hr;
629
630     // 6. update reservoir state, log
631     this->stored_volume_m3 += net_flow_m3hr * dt_hrs;
632     this->stored_volume_vec_m3[timestep] = this->stored_volume_m3;
633
634     return;
635 } /* __updateState() */

```

4.8.3.13 __writeSummary()

```

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

```

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

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 [Noncombustion](#).

```

653 {
654     // 1. create filestream
655     write_path += "summary_results.md";
656     std::ofstream ofs;
657     ofs.open(write_path, std::ofstream::out);
658
659     // 2. write to summary results (markdown)
660     ofs << "# ";
661     ofs << std::to_string(int(ceil(this->capacity_kW)));
662     ofs << " kW HYDRO Summary Results\n";
663     ofs << "\n-----\n\n";
664
665     // 2.1. Production attributes
666     ofs << "## Production Attributes\n";
667     ofs << "\n";
668
669     ofs << "Capacity: " << this->capacity_kW << " kW \n";
670     ofs << "\n";
671
672     ofs << "Production Override: (N = 0 / Y = 1): "
673         << this->normalized_production_series_given << " \n";
674     if (this->normalized_production_series_given) {
675         ofs << "Path to Normalized Production Time Series: "
676             << this->path_2_normalized_production_time_series << " \n";
677     }
678     ofs << "\n";
679
680     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
681     ofs << "Capital Cost: " << this->capital_cost << " \n";
682     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
683         << " per kWh produced \n";
684     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
685         << " \n";
686     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
687         << " \n";
688     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
689     ofs << "\n";

```

```

690
691 ofs « "Replacement Running Hours: " « this->replace_running_hrs « " \n";
692 ofs « "\n-----\n\n";
693
694 // 2.2. Noncombustion attributes
695 ofs « "## Noncombustion Attributes\n";
696 ofs « "\n";
697
698 //...
699
700 ofs « "\n-----\n\n";
701
702 // 2.3. Hydro attributes
703 ofs « "## Hydro Attributes\n";
704 ofs « "\n";
705
706 ofs « "Fluid Density: " « this->fluid_density_kgm3 « " kg/m3 \n";
707 ofs « "Net Head: " « this->net_head_m « " m \n";
708 ofs « "\n";
709
710 ofs « "Reservoir Volume: " « this->reservoir_capacity_m3 « " m3 \n";
711 ofs « "Reservoir Initial State: " « this->init_reservoir_state « " \n";
712 ofs « "\n";
713
714 ofs « "Turbine Type: ";
715 switch(this->turbine_type) {
716     case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
717         ofs « "PELTON";
718
719         break;
720     }
721
722     case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
723         ofs « "FRANCIS";
724
725         break;
726     }
727
728     case(HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
729         ofs « "KAPLAN";
730
731         break;
732     }
733
734     default: {
735         // write nothing!
736
737         break;
738     }
739 }
740 ofs « " \n";
741 ofs « "\n";
742 ofs « "Minimum Flow: " « this->minimum_flow_m3hr « " m3/hr \n";
743 ofs « "Maximum Flow: " « this->maximum_flow_m3hr « " m3/hr \n";
744 ofs « "\n";
745 ofs « "Minimum Production: " « this->minimum_power_kW « " kW \n";
746 ofs « "\n";
747
748 ofs « "\n-----\n\n";
749
750 // 2.4. Hydro Results
751 ofs « "## Results\n";
752 ofs « "\n";
753
754 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
755 ofs « "\n";
756
757 ofs « "Total Dispatch: " « this->total_dispatch_kWh
758     « " kWh \n";
759
760 ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
761     « " per kWh dispatched \n";
762 ofs « "\n";
763
764 ofs « "Running Hours: " « this->running_hours « " \n";
765 ofs « "Replacements: " « this->n_replacements « " \n";
766
767 //...
768
769 ofs « "\n-----\n\n";
770
771 ofs.close();
772 return;
773 } /* __writeSummary() */

```

4.8.3.14 __writeTimeSeries()

```
void Hydro::__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 [Hydro](#).

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>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Noncombustion](#).

```
803 {
804     // 1. create filestream
805     write_path += "time_series_results.csv";
806     std::ofstream ofs;
807     ofs.open(write_path, std::ofstream::out);
808
809     // 2. write time series results (comma separated value)
810     ofs << "Time (since start of data) [hrs],";
811     ofs << "Production [kW],";
812     ofs << "Dispatch [kW],";
813     ofs << "Storage [kW],";
814     ofs << "Curtailement [kW],";
815     ofs << "Is Running (N = 0 / Y = 1),";
816     ofs << "Turbine Flow [m3/hr],";
817     ofs << "Spill Rate [m3/hr],";
818     ofs << "Stored Volume [m3],";
819     ofs << "Capital Cost (actual),";
820     ofs << "Operation and Maintenance Cost (actual),";
821     ofs << "\n";
822
823     for (int i = 0; i < max_lines; i++) {
824         ofs << time_vec_hrs_ptr->at(i) << ",";
825         ofs << this->production_vec_kW[i] << ",";
826         ofs << this->dispatch_vec_kW[i] << ",";
827         ofs << this->storage_vec_kW[i] << ",";
828         ofs << this->curtailment_vec_kW[i] << ",";
829         ofs << this->is_running_vec[i] << ",";
830         ofs << this->turbine_flow_vec_m3hr[i] << ",";
831         ofs << this->spill_rate_vec_m3hr[i] << ",";
832         ofs << this->stored_volume_vec_m3[i] << ",";
833         ofs << this->capital_cost_vec[i] << ",";
834         ofs << this->operation_maintenance_cost_vec[i] << ",";
835         ofs << "\n";
836     }
837
838     ofs.close();
839     return;
840 } /* __writeTimeSeries() */
```

4.8.3.15 commit()

```
double Hydro::commit (
    int timestep,
    double dt_hrs,
    double production_kW,
    double load_kW,
    double hydro_resource_m3hr ) [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 [Noncombustion](#).

```

1092 {
1093     // 1. invoke base class method
1094     load_kW = Noncombustion :: commit(
1095         timestep,
1096         dt_hrs,
1097         production_kW,
1098         load_kW
1099     );
1100
1101     // 2. update state and record
1102     this->__updateState(
1103         timestep,
1104         dt_hrs,
1105         production_kW,
1106         hydro_resource_m3hr
1107     );
1108
1109     return load_kW;
1110 } /* commit() */

```

4.8.3.16 handleReplacement()

```

void Hydro::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Noncombustion](#).

```

969 {
970     // 1. reset attributes
971     //...
972
973     // 2. invoke base class method
974     Noncombustion :: handleReplacement(timestep);
975
976     return;
977 } /* __handleReplacement() */

```

4.8.3.17 requestProductionkW()

```

double Hydro::requestProductionkW (
    int timestep,

```

```
double dt_hrs,
double request_kW,
double hydro_resource_m3hr ) [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].
<i>hydro_resource_m3hr</i>	The currently available hydro flow resource [m3/hr].

Returns

The production [kW] delivered by the hydro generator.

Reimplemented from [Noncombustion](#).

```
1013 {
1014     // 0. given production time series override
1015     if (this->normalized_production_series_given) {
1016         double production_kW = Production::getProductionkW(timestep);
1017
1018         return production_kW;
1019     }
1020
1021     // 1. return on request of zero
1022     if (request_kW <= 0) {
1023         return 0;
1024     }
1025
1026     // 2. if request is less than minimum power, set to minimum power
1027     if (request_kW < this->minimum_power_kW) {
1028         request_kW = this->minimum_power_kW;
1029     }
1030
1031     // 3. check available flow, return if less than minimum flow
1032     double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);
1033
1034     if (available_flow_m3hr < this->minimum_flow_m3hr) {
1035         return 0;
1036     }
1037
1038     // 4. init production to request, enforce capacity constraint (which also accounts
1039     //     for maximum flow constraint).
1040     double production_kW = request_kW;
1041
1042     if (production_kW > this->capacity_kW) {
1043         production_kW = this->capacity_kW;
1044     }
1045
1046     // 5. map production to flow
1047     double flow_m3hr = this->__powerToFlow(production_kW);
1048
1049     // 6. if flow is in excess of available, then adjust production accordingly
1050     if (flow_m3hr > available_flow_m3hr) {
1051         production_kW = this->__flowToPower(available_flow_m3hr);
1052     }
1053
1054     return production_kW;
1055 } /* requestProductionkW() */
```

4.8.4 Member Data Documentation

4.8.4.1 fluid_density_kgm3

```
double Hydro::fluid_density_kgm3
```

The density [kg/m3] of the hydroelectric working fluid.

4.8.4.2 init_reservoir_state

```
double Hydro::init_reservoir_state
```

The initial state of the reservoir (where state is volume of stored fluid divided by capacity).

4.8.4.3 maximum_flow_m3hr

```
double Hydro::maximum_flow_m3hr
```

The maximum productive flow [m3/hr] that the asset can support.

4.8.4.4 minimum_flow_m3hr

```
double Hydro::minimum_flow_m3hr
```

The minimum required flow [m3/hr] for the asset to produce. Corresponds to minimum power.

4.8.4.5 minimum_power_kW

```
double Hydro::minimum_power_kW
```

The minimum power [kW] that the asset can produce. Corresponds to minimum productive flow.

4.8.4.6 net_head_m

```
double Hydro::net_head_m
```

The net head [m] of the asset.

4.8.4.7 reservoir_capacity_m3

```
double Hydro::reservoir_capacity_m3
```

The capacity [m3] of the hydro reservoir.

4.8.4.8 spill_rate_vec_m3hr

```
std::vector<double> Hydro::spill_rate_vec_m3hr
```

A vector of the spill rate [m3/hr] at each point in the modelling time series.

4.8.4.9 stored_volume_m3

```
double Hydro::stored_volume_m3
```

The volume [m3] of stored fluid.

4.8.4.10 stored_volume_vec_m3

```
std::vector<double> Hydro::stored_volume_vec_m3
```

A vector of the stored volume [m3] in the reservoir at each point in the modelling time series.

4.8.4.11 turbine_flow_vec_m3hr

```
std::vector<double> Hydro::turbine_flow_vec_m3hr
```

A vector of the turbine flow [m3/hr] at each point in the modelling time series.

4.8.4.12 turbine_type

```
HydroTurbineType Hydro::turbine_type
```

The type of hydroelectric turbine model to use.

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

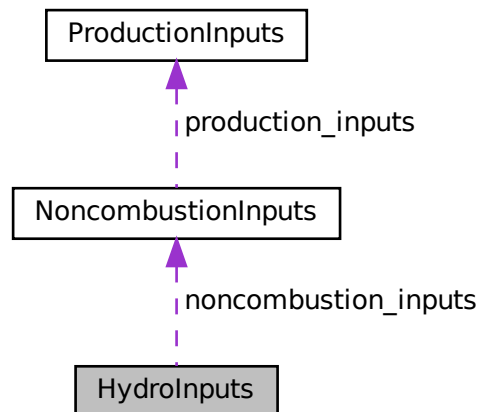
- [header/Production/Noncombustion/Hydro.h](#)
- [source/Production/Noncombustion/Hydro.cpp](#)

4.9 HydroInputs Struct Reference

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

```
#include <Hydro.h>
```

Collaboration diagram for HydroInputs:



Public Attributes

- [NoncombustionInputs](#) `noncombustion_inputs`
An encapsulated [NoncombustionInputs](#) 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 `fluid_density_kgm3` = 1000
The density [kg/m3] of the hydroelectric working fluid.
- double `net_head_m` = 500
The net head [m] of the asset.
- double `reservoir_capacity_m3` = 0
The capacity [m3] of the hydro reservoir.
- double `init_reservoir_state` = 0
The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- [HydroTurbineType](#) `turbine_type` = [HydroTurbineType](#) :: `HYDRO_TURBINE_PELTON`
The type of hydroelectric turbine model to use.

4.9.1 Detailed Description

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

4.9.2 Member Data Documentation

4.9.2.1 capital_cost

```
double HydroInputs::capital_cost = -1
```

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

4.9.2.2 fluid_density_kgm3

```
double HydroInputs::fluid_density_kgm3 = 1000
```

The density [kg/m3] of the hydroelectric working fluid.

4.9.2.3 init_reservoir_state

```
double HydroInputs::init_reservoir_state = 0
```

The initial state of the reservoir (where state is volume of stored fluid divided by capacity).

4.9.2.4 net_head_m

```
double HydroInputs::net_head_m = 500
```

The net head [m] of the asset.

4.9.2.5 noncombustion_inputs

```
NoncombustionInputs HydroInputs::noncombustion_inputs
```

An encapsulated [NoncombustionInputs](#) instance.

4.9.2.6 operation_maintenance_cost_kWh

```
double HydroInputs::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].

4.9.2.7 reservoir_capacity_m3

```
double HydroInputs::reservoir_capacity_m3 = 0
```

The capacity [m3] of the hydro reservoir.

4.9.2.8 resource_key

```
int HydroInputs::resource_key = 0
```

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

4.9.2.9 turbine_type

```
HydroTurbineType HydroInputs::turbine_type = HydroTurbineType :: HYDRO_TURBINE_PELTON
```

The type of hydroelectric turbine model to use.

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

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

4.10 Interpolator Class Reference

A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

```
#include <Interpolator.h>
```

Public Member Functions

- [Interpolator](#) (void)
Constructor for the [Interpolator](#) class.
- void [addData1D](#) (int, std::string)
Method to add 1D interpolation data to the [Interpolator](#).
- void [addData2D](#) (int, std::string)
Method to add 2D interpolation data to the [Interpolator](#).
- double [interp1D](#) (int, double)
Method to perform a 1D interpolation.
- double [interp2D](#) (int, double, double)
Method to perform a 2D interpolation.
- [~Interpolator](#) (void)
Destructor for the [Interpolator](#) class.

Public Attributes

- std::map< int, [InterpolatorStruct1D](#) > [interp_map_1D](#)
A map <int, [InterpolatorStruct1D](#)> of given 1D interpolation data.
- std::map< int, std::string > [path_map_1D](#)
A map <int, string> of the paths (either relative or absolute) to the given 1D interpolation data.
- std::map< int, [InterpolatorStruct2D](#) > [interp_map_2D](#)
A map <int, [InterpolatorStruct2D](#)> of given 2D interpolation data.
- std::map< int, std::string > [path_map_2D](#)
A map <int, string> of the paths (either relative or absolute) to the given 2D interpolation data.

Private Member Functions

- void [__checkDataKey1D](#) (int)
Helper method to check if given data key (1D) is already in use.
- void [__checkDataKey2D](#) (int)
Helper method to check if given data key (2D) is already in use.
- void [__checkBounds1D](#) (int, double)
Helper method to check that the given 1D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void [__checkBounds2D](#) (int, double, double)
Helper method to check that the given 2D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void [__throwReadError](#) (std::string, int)
Helper method to throw a read error whenever non-numeric data is encountered where only numeric data should be.
- bool [__isNonNumeric](#) (std::string)
Helper method to determine if given string is non-numeric (i.e., contains.
- int [__getInterpolationIndex](#) (double, std::vector< double > *)
Helper method to get appropriate interpolation index into given vector.
- std::vector< std::string > [__splitCommaSeparatedString](#) (std::string, std::string="|")
Helper method to split a comma-separated string into a vector of substrings.
- std::vector< std::vector< std::string > > [__getDataStringMatrix](#) (std::string)
- void [__readData1D](#) (int, std::string)
Helper method to read the given 1D interpolation data into [Interpolator](#).
- void [__readData2D](#) (int, std::string)
Helper method to read the given 2D interpolation data into [Interpolator](#).

4.10.1 Detailed Description

A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 Interpolator()

```
Interpolator::Interpolator (
    void )
```

Constructor for the [Interpolator](#) class.

```
707 {
708     //...
709
710     return;
711 } /* Interpolator() */
```

4.10.2.2 ~Interpolator()

```
Interpolator::~Interpolator (
    void )
```

Destructor for the [Interpolator](#) class.

```
893 {
894     //...
895
896     return;
897 } /* ~Interpolator() */
```

4.10.3 Member Function Documentation

4.10.3.1 __checkBounds1D()

```
void Interpolator::__checkBounds1D (
    int data_key,
    double interp_x ) [private]
```

Helper method to check that the given 1D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.

Parameters

<i>data_key</i>	A key associated with the given interpolation data.
<i>interp_↔</i>	The query value to be interpolated.
<i>_x</i>	

```

133 {
134     // 1. key error
135     if (this->interp_map_1D.count(data_key) == 0) {
136         std::string error_str = "ERROR: Interpolator::interp1D() ";
137         error_str += "data key ";
138         error_str += std::to_string(data_key);
139         error_str += " has not been registered";
140
141         #ifdef _WIN32
142             std::cout << error_str << std::endl;
143         #endif
144
145         throw std::invalid_argument(error_str);
146     }
147
148     // 2. bounds error
149     if (
150         interp_x < this->interp_map_1D[data_key].min_x or
151         interp_x > this->interp_map_1D[data_key].max_x
152     ) {
153         std::string error_str = "ERROR: Interpolator::interp1D() ";
154         error_str += "interpolation value ";
155         error_str += std::to_string(interp_x);
156         error_str += " is outside of the given interpolation data domain [";
157         error_str += std::to_string(this->interp_map_1D[data_key].min_x);
158         error_str += " , ";
159         error_str += std::to_string(this->interp_map_1D[data_key].max_x);
160         error_str += " ]";
161
162         #ifdef _WIN32
163             std::cout << error_str << std::endl;
164         #endif
165
166         throw std::invalid_argument(error_str);
167     }
168
169     return;
170 } /* __checkBounds1D() */

```

4.10.3.2 __checkBounds2D()

```

void Interpolator::__checkBounds2D (
    int data_key,
    double interp_x,
    double interp_y ) [private]

```

Helper method to check that the given 2D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.

Parameters

<i>data_key</i>	A key associated with the given interpolation data.
<i>interp_x</i>	The first query value to be interpolated.
<i>interp_y</i>	The second query value to be interpolated.

```

193 {
194     // 1. key error
195     if (this->interp_map_2D.count(data_key) == 0) {
196         std::string error_str = "ERROR: Interpolator::interp2D() ";
197         error_str += "data key ";
198         error_str += std::to_string(data_key);
199         error_str += " has not been registered";
200
201         #ifdef _WIN32
202             std::cout << error_str << std::endl;
203         #endif
204
205         throw std::invalid_argument(error_str);
206     }

```

```

207
208 // 2. bounds error (x_interp)
209 if (
210     interp_x < this->interp_map_2D[data_key].min_x or
211     interp_x > this->interp_map_2D[data_key].max_x
212 ) {
213     std::string error_str = "ERROR: Interpolator::interp2D() ";
214     error_str += "interpolation value interp_x = ";
215     error_str += std::to_string(interp_x);
216     error_str += " is outside of the given interpolation data domain [";
217     error_str += std::to_string(this->interp_map_2D[data_key].min_x);
218     error_str += " , ";
219     error_str += std::to_string(this->interp_map_2D[data_key].max_x);
220     error_str += "]\n";
221
222     #ifdef _WIN32
223         std::cout << error_str << std::endl;
224     #endif
225
226     throw std::invalid_argument(error_str);
227 }
228
229 // 2. bounds error (y_interp)
230 if (
231     interp_y < this->interp_map_2D[data_key].min_y or
232     interp_y > this->interp_map_2D[data_key].max_y
233 ) {
234     std::string error_str = "ERROR: Interpolator::interp2D() ";
235     error_str += "interpolation value interp_y = ";
236     error_str += std::to_string(interp_y);
237     error_str += " is outside of the given interpolation data domain [";
238     error_str += std::to_string(this->interp_map_2D[data_key].min_y);
239     error_str += " , ";
240     error_str += std::to_string(this->interp_map_2D[data_key].max_y);
241     error_str += "]\n";
242
243     #ifdef _WIN32
244         std::cout << error_str << std::endl;
245     #endif
246
247     throw std::invalid_argument(error_str);
248 }
249
250 return;
251 } /* __checkBounds2D() */

```

4.10.3.3 __checkDataKey1D()

```

void Interpolator::__checkDataKey1D (
    int data_key ) [private]

```

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

Parameters

<i>data_key</i>	The key associated with the given 1D interpolation data.
-----------------	--

```

65 {
66     if (this->interp_map_1D.count(data_key) > 0) {
67         std::string error_str = "ERROR: Interpolator::addData1D() ";
68         error_str += "data key (1D) ";
69         error_str += std::to_string(data_key);
70         error_str += " is already in use";
71
72         #ifdef _WIN32
73             std::cout << error_str << std::endl;
74         #endif
75
76         throw std::invalid_argument(error_str);
77     }
78
79     return;
80 } /* __checkDataKey1D() */

```

4.10.3.4 `__checkDataKey2D()`

```
void Interpolator::__checkDataKey2D (
    int data_key ) [private]
```

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

Parameters

<code>data_key</code>	The key associated with the given 2D interpolation data.
-----------------------	--

```
97 {
98     if (this->interp_map_2D.count(data_key) > 0) {
99         std::string error_str = "ERROR: Interpolator::addData2D() ";
100         error_str += "data key (2D) ";
101         error_str += std::to_string(data_key);
102         error_str += " is already in use";
103
104         #ifdef _WIN32
105             std::cout << error_str << std::endl;
106         #endif
107
108         throw std::invalid_argument(error_str);
109     }
110
111     return;
112 } /* __checkDataKey2D() */
```

4.10.3.5 `__getDataStringMatrix()`

```
std::vector< std::vector< std::string > > Interpolator::__getDataStringMatrix (
    std::string path_2_data ) [private]
```

```
426 {
427     // 1. create input file stream
428     std::ifstream ifs;
429     ifs.open(path_2_data);
430
431     // 2. check that open() worked
432     if (not ifs.is_open()) {
433         std::string error_str = "ERROR: Interpolator::__getDataStringMatrix() ";
434         error_str += " failed to open ";
435         error_str += path_2_data;
436
437         #ifdef _WIN32
438             std::cout << error_str << std::endl;
439         #endif
440
441         throw std::invalid_argument(error_str);
442     }
443
444     // 3. read file line by line
445     bool is_header = true;
446     std::string line;
447     std::vector<std::string> line_split_vec;
448     std::vector<std::vector<std::string>> string_matrix;
449
450     while (not ifs.eof()) {
451         std::getline(ifs, line);
452
453         if (is_header) {
454             is_header = false;
455             continue;
456         }
457
458         line_split_vec = this->__splitCommaSeparatedString(line);
459
460         if (not line_split_vec.empty()) {
461             string_matrix.push_back(line_split_vec);
462         }
463     }
464
465     ifs.close();
466     return string_matrix;
467 } /* __getDataStringMatrix() */
```

4.10.3.6 `__getInterpolationIndex()`

```
int Interpolator::__getInterpolationIndex (
    double interp_x,
    std::vector< double > * x_vec_ptr ) [private]
```

Helper method to get appropriate interpolation index into given vector.

Parameters

<i>interp_x</i>	The query value to be interpolated.
<i>x_vec_ptr</i>	A pointer to the given vector of interpolation data.

Returns

The appropriate interpolation index into the given vector.

```
343 {
344     int idx = 0;
345     while (
346         not (interp_x >= x_vec_ptr->at(idx) and interp_x <= x_vec_ptr->at(idx + 1))
347     ) {
348         idx++;
349     }
350
351     return idx;
352 } /* __getInterpolationIndex() */
```

4.10.3.7 `__isNonNumeric()`

```
bool Interpolator::__isNonNumeric (
    std::string str ) [private]
```

Helper method to determine if given string is non-numeric (i.e., contains.

Parameters

<i>str</i>	The string being tested.
------------	--------------------------

Returns

A boolean indicating if the given string is non-numeric.

```
308 {
309     for (size_t i = 0; i < str.size(); i++) {;
310         if (isalpha(str[i])) {
311             return true;
312         }
313     }
314
315     return false;
316 } /* __isAlpha() */
```


4.10.3.8 `__readData1D()`

```
void Interpolator::__readData1D (
    int data_key,
    std::string path_2_data ) [private]
```

Helper method to read the given 1D interpolation data into [Interpolator](#).

Parameters

<i>data_key</i>	A key associated with the given interpolation data.
<i>path_2_data</i>	The path (either relative or absolute) to the given interpolation data.

```
487 {
488     // 1. get string matrix
489     std::vector<std::vector<std::string>> string_matrix =
490         this->__getDataStringMatrix(path_2_data);
491
492     // 2. read string matrix contents into 1D interpolation struct
493     InterpolatorStruct1D interp_struct_1D;
494
495     interp_struct_1D.n_points = string_matrix.size();
496     interp_struct_1D.x_vec.resize(interp_struct_1D.n_points, 0);
497     interp_struct_1D.y_vec.resize(interp_struct_1D.n_points, 0);
498
499     for (int i = 0; i < interp_struct_1D.n_points; i++) {
500         try {
501             interp_struct_1D.x_vec[i] = std::stod(string_matrix[i][0]);
502             interp_struct_1D.y_vec[i] = std::stod(string_matrix[i][1]);
503         }
504
505         catch (...) {
506             this->__throwReadError(path_2_data, 1);
507         }
508     }
509
510     interp_struct_1D.min_x = interp_struct_1D.x_vec[0];
511     interp_struct_1D.max_x = interp_struct_1D.x_vec[interp_struct_1D.n_points - 1];
512
513     // 3. write struct to map
514     this->interp_map_1D.insert(
515         std::pair<int, InterpolatorStruct1D>(data_key, interp_struct_1D)
516     );
517
518     /*
519     // ==== TEST PRINT ==== //
520     std::cout << std::endl;
521     std::cout << path_2_data << std::endl;
522     std::cout << "-----" << std::endl;
523
524     std::cout << "n_points: " << this->interp_map_1D[data_key].n_points << std::endl;
525
526     std::cout << "x_vec: [";
527     for (
528         int i = 0;
529         i < this->interp_map_1D[data_key].n_points;
530         i++
531     ) {
532         std::cout << this->interp_map_1D[data_key].x_vec[i] << ", ";
533     }
534     std::cout << "]" << std::endl;
535
536     std::cout << "y_vec: [";
537     for (
538         int i = 0;
539         i < this->interp_map_1D[data_key].n_points;
540         i++
541     ) {
542         std::cout << this->interp_map_1D[data_key].y_vec[i] << ", ";
543     }
544     std::cout << "]" << std::endl;
545
546     std::cout << std::endl;
547     // ==== END TEST PRINT ==== //
548     /**/
549
550     return;
551 } /* __readData1D() */
```

4.10.3.9 __readData2D()

```
void Interpolator::__readData2D (
    int data_key,
    std::string path_2_data ) [private]
```

Helper method to read the given 2D interpolation data into [Interpolator](#).

Parameters

<i>data_key</i>	A key associated with the given interpolation data.
<i>path_2_data</i>	The path (either relative or absolute) to the given interpolation data.

```
571 {
572     // 1. get string matrix
573     std::vector<std::vector<std::string>> string_matrix =
574         this->__getDataStringMatrix(path_2_data);
575
576     // 2. read string matrix contents into 2D interpolation map
577     InterpolatorStruct2D interp_struct_2D;
578
579     interp_struct_2D.n_rows = string_matrix.size() - 1;
580     interp_struct_2D.n_cols = string_matrix[0].size() - 1;
581
582     interp_struct_2D.x_vec.resize(interp_struct_2D.n_cols, 0);
583     interp_struct_2D.y_vec.resize(interp_struct_2D.n_rows, 0);
584
585     interp_struct_2D.z_matrix.resize(interp_struct_2D.n_rows, {});
586
587     for (int i = 0; i < interp_struct_2D.n_rows; i++) {
588         interp_struct_2D.z_matrix[i].resize(interp_struct_2D.n_cols, 0);
589     }
590
591     for (size_t i = 1; i < string_matrix[0].size(); i++) {
592         try {
593             interp_struct_2D.x_vec[i - 1] = std::stod(string_matrix[0][i]);
594         }
595         catch (...) {
596             this->__throwReadError(path_2_data, 2);
597         }
598     }
599
600     interp_struct_2D.min_x = interp_struct_2D.x_vec[0];
601     interp_struct_2D.max_x = interp_struct_2D.x_vec[interp_struct_2D.n_cols - 1];
602
603     for (size_t i = 1; i < string_matrix.size(); i++) {
604         try {
605             interp_struct_2D.y_vec[i - 1] = std::stod(string_matrix[i][0]);
606         }
607         catch (...) {
608             this->__throwReadError(path_2_data, 2);
609         }
610     }
611
612     interp_struct_2D.min_y = interp_struct_2D.y_vec[0];
613     interp_struct_2D.max_y = interp_struct_2D.y_vec[interp_struct_2D.n_rows - 1];
614
615     for (size_t i = 1; i < string_matrix.size(); i++) {
616         for (size_t j = 1; j < string_matrix[0].size(); j++) {
617             try {
618                 interp_struct_2D.z_matrix[i - 1][j - 1] = std::stod(string_matrix[i][j]);
619             }
620             catch (...) {
621                 this->__throwReadError(path_2_data, 2);
622             }
623         }
624     }
625
626     // 3. write struct to map
627     this->interp_map_2D.insert(
628         std::pair<int, InterpolatorStruct2D>(data_key, interp_struct_2D)
629     );
630
631     /*
632     // ==== TEST PRINT ==== //
633     std::cout << std::endl;
634     std::cout << path_2_data << std::endl;
```

```

638     std::cout << "-----" << std::endl;
639
640     std::cout << "n_rows: " << this->interp_map_2D[data_key].n_rows << std::endl;
641     std::cout << "n_cols: " << this->interp_map_2D[data_key].n_cols << std::endl;
642
643     std::cout << "x_vec: [";
644     for (
645         int i = 0;
646         i < this->interp_map_2D[data_key].n_cols;
647         i++
648     ) {
649         std::cout << this->interp_map_2D[data_key].x_vec[i] << ", ";
650     }
651     std::cout << "]" << std::endl;
652
653     std::cout << "y_vec: [";
654     for (
655         int i = 0;
656         i < this->interp_map_2D[data_key].n_rows;
657         i++
658     ) {
659         std::cout << this->interp_map_2D[data_key].y_vec[i] << ", ";
660     }
661     std::cout << "]" << std::endl;
662
663     std::cout << "z_matrix:" << std::endl;
664     for (
665         int i = 0;
666         i < this->interp_map_2D[data_key].n_rows;
667         i++
668     ) {
669         std::cout << "\t[";
670
671         for (
672             int j = 0;
673             j < this->interp_map_2D[data_key].n_cols;
674             j++
675         ) {
676             std::cout << this->interp_map_2D[data_key].z_matrix[i][j] << ", ";
677         }
678
679         std::cout << "]" << std::endl;
680     }
681     std::cout << std::endl;
682
683     std::cout << std::endl;
684     // ==== END TEST PRINT ==== //
685     /**/
686
687     return;
688 } /* __readData2D() */

```

4.10.3.10 __splitCommaSeparatedString()

```

std::vector< std::string > Interpolator::__splitCommaSeparatedString (
    std::string str,
    std::string break_str = "|" ) [private]

```

Helper method to split a comma-separated string into a vector of substrings.

Parameters

<i>str</i>	The string to be split.
<i>break_str</i>	A string which triggers the function to break. What has been split up to the point of the break is then returned.

Returns

A vector of substrings, which follows from splitting the given string in a comma separated manner.

```

381 {
382     std::vector<std::string> str_split_vec;
383
384     size_t idx = 0;
385     std::string substr;
386
387     while ((idx = str.find(',', ' ')) != std::string::npos) {
388         substr = str.substr(0, idx);
389
390         if (substr == break_str) {
391             break;
392         }
393
394         str_split_vec.push_back(substr);
395
396         str.erase(0, idx + 1);
397     }
398
399     return str_split_vec;
400 } /* __splitCommaSeparatedString() */

```

4.10.3.11 __throwReadError()

```

void Interpolator::__throwReadError (
    std::string path_2_data,
    int dimensions ) [private]

```

Helper method to throw a read error whenever non-numeric data is encountered where only numeric data should be.

Parameters

<i>path_2_data</i>	The path (either relative or absolute) to the given interpolation data.
<i>dimensions</i>	The dimensionality of the data being read.

```

272 {
273     std::string error_str = "ERROR: Interpolator::addData";
274     error_str += std::to_string(dimensions);
275     error_str += "D() ";
276     error_str += " failed to read ";
277     error_str += path_2_data;
278     error_str += " (this is probably a std::stod() error; is there non-numeric ";
279     error_str += "data where only numeric data should be?)";
280
281     #ifdef _WIN32
282         std::cout << error_str << std::endl;
283     #endif
284
285     throw std::runtime_error(error_str);
286
287     return;
288 } /* __throwReadError() */

```

4.10.3.12 addData1D()

```

void Interpolator::addData1D (
    int data_key,
    std::string path_2_data )

```

Method to add 1D interpolation data to the [Interpolator](#).

Parameters

<i>data_key</i>	A key used to index into the Interpolator .
<i>path_2_data</i>	A path (either relative or absolute) to the given 1D interpolation data.

```

731 {
732     // 1. check key
733     this->__checkDataKey1D(data_key);
734
735     // 2. read data into map
736     this->__readData1D(data_key, path_2_data);
737
738     // 3. record path
739     this->path_map_1D.insert(std::pair<int, std::string>(data_key, path_2_data));
740
741     return;
742 } /* addData1D() */

```

4.10.3.13 addData2D()

```

void Interpolator::addData2D (
    int data_key,
    std::string path_2_data )

```

Method to add 2D interpolation data to the [Interpolator](#).

Parameters

<i>data_key</i>	A key used to index into the Interpolator .
<i>path_2_data</i>	A path (either relative or absolute) to the given 2D interpolation data.

```

762 {
763     // 1. check key
764     this->__checkDataKey2D(data_key);
765
766     // 2. read data into map
767     this->__readData2D(data_key, path_2_data);
768
769     // 3. record path
770     this->path_map_2D.insert(std::pair<int, std::string>(data_key, path_2_data));
771
772     return;
773 } /* addData2D() */

```

4.10.3.14 interp1D()

```

double Interpolator::interp1D (
    int data_key,
    double interp_x )

```

Method to perform a 1D interpolation.

Parameters

<i>data_key</i>	A key used to index into the Interpolator .
<i>interp_x</i>	The query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.

Returns

An interpolation of the given query value.

```

795 {
796     // 1. check bounds
797     this->__checkBounds1D(data_key, interp_x);
798
799     // 2. get interpolation index
800     int idx = this->__getInterpolationIndex(
801         interp_x,
802         &(this->interp_map_1D[data_key].x_vec)
803     );
804
805     // 3. perform interpolation
806     double x_0 = this->interp_map_1D[data_key].x_vec[idx];
807     double x_1 = this->interp_map_1D[data_key].x_vec[idx + 1];
808
809     double y_0 = this->interp_map_1D[data_key].y_vec[idx];
810     double y_1 = this->interp_map_1D[data_key].y_vec[idx + 1];
811
812     double interp_y = ((y_1 - y_0) / (x_1 - x_0)) * (interp_x - x_0) + y_0;
813
814     return interp_y;
815 } /* interp1D() */

```

4.10.3.15 interp2D()

```

double Interpolator::interp2D (
    int data_key,
    double interp_x,
    double interp_y )

```

Method to perform a 2D interpolation.

Parameters

<i>data_key</i>	A key used to index into the Interpolator .
<i>interp_x</i>	The first query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.
<i>interp_y</i>	The second query value to be interpolated. If this value is outside the domain of the associated interpolation data, then an error will occur.

Returns

An interpolation of the given query values.

```

840 {
841     // 1. check bounds
842     this->__checkBounds2D(data_key, interp_x, interp_y);
843
844     // 2. get interpolation indices
845     int idx_x = this->__getInterpolationIndex(
846         interp_x,
847         &(this->interp_map_2D[data_key].x_vec)
848     );
849
850     int idx_y = this->__getInterpolationIndex(
851         interp_y,
852         &(this->interp_map_2D[data_key].y_vec)
853     );
854
855     // 3. perform first horizontal interpolation
856     double x_0 = this->interp_map_2D[data_key].x_vec[idx_x];
857     double x_1 = this->interp_map_2D[data_key].x_vec[idx_x + 1];
858
859     double z_0 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x];
860     double z_1 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x + 1];

```

```

861
862     double interp_z_0 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
863
864     // 4. perform second horizontal interpolation
865     z_0 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x];
866     z_1 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x + 1];
867
868     double interp_z_1 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
869
870     // 5. perform vertical interpolation
871     double y_0 = this->interp_map_2D[data_key].y_vec[idx_y];
872     double y_1 = this->interp_map_2D[data_key].y_vec[idx_y + 1];
873
874     double interp_z =
875         ((interp_z_1 - interp_z_0) / (y_1 - y_0)) * (interp_y - y_0) + interp_z_0;
876
877     return interp_z;
878 } /* interp2D() */

```

4.10.4 Member Data Documentation

4.10.4.1 interp_map_1D

`std::map<int, InterpolatorStruct1D> Interpolator::interp_map_1D`

A map <int, [InterpolatorStruct1D](#)> of given 1D interpolation data.

4.10.4.2 interp_map_2D

`std::map<int, InterpolatorStruct2D> Interpolator::interp_map_2D`

A map <int, [InterpolatorStruct2D](#)> of given 2D interpolation data.

4.10.4.3 path_map_1D

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

A map <int, string> of the paths (either relative or absolute) to the given 1D interpolation data.

4.10.4.4 path_map_2D

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

A map <int, string> of the paths (either relative or absolute) to the given 2D interpolation data.

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

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

4.11 InterpolatorStruct1D Struct Reference

A struct which holds two parallel vectors for use in 1D interpolation.

```
#include <Interpolator.h>
```

Public Attributes

- int `n_points` = 0
The number of data points in each parallel vector.
- `std::vector< double > x_vec` = {}
A vector of independent data.
- double `min_x` = 0
The minimum (i.e., first) element of `x_vec`.
- double `max_x` = 0
The maximum (i.e., last) element of `x_vec`.
- `std::vector< double > y_vec` = {}
A vector of dependent data.

4.11.1 Detailed Description

A struct which holds two parallel vectors for use in 1D interpolation.

4.11.2 Member Data Documentation

4.11.2.1 `max_x`

```
double InterpolatorStruct1D::max_x = 0
```

The maximum (i.e., last) element of `x_vec`.

4.11.2.2 `min_x`

```
double InterpolatorStruct1D::min_x = 0
```

The minimum (i.e., first) element of `x_vec`.

4.11.2.3 n_points

```
int InterpolatorStruct1D::n_points = 0
```

The number of data points in each parallel vector.

4.11.2.4 x_vec

```
std::vector<double> InterpolatorStruct1D::x_vec = {}
```

A vector of independent data.

4.11.2.5 y_vec

```
std::vector<double> InterpolatorStruct1D::y_vec = {}
```

A vector of dependent data.

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

- header/[Interpolator.h](#)

4.12 InterpolatorStruct2D Struct Reference

A struct which holds two parallel vectors and a matrix for use in 2D interpolation.

```
#include <Interpolator.h>
```

Public Attributes

- int [n_rows](#) = 0
The number of rows in the matrix (also the length of y_vec)
- int [n_cols](#) = 0
The number of cols in the matrix (also the length of x_vec)
- std::vector< double > [x_vec](#) = {}
A vector of independent data (columns).
- double [min_x](#) = 0
The minimum (i.e., first) element of x_vec.
- double [max_x](#) = 0
The maximum (i.e., last) element of x_vec.
- std::vector< double > [y_vec](#) = {}
A vector of independent data (rows).
- double [min_y](#) = 0
The minimum (i.e., first) element of y_vec.
- double [max_y](#) = 0
The maximum (i.e., last) element of y_vec.
- std::vector< std::vector< double > > [z_matrix](#) = {}
A matrix of dependent data.

4.12.1 Detailed Description

A struct which holds two parallel vectors and a matrix for use in 2D interpolation.

4.12.2 Member Data Documentation

4.12.2.1 max_x

```
double InterpolatorStruct2D::max_x = 0
```

The maximum (i.e., last) element of x_vec.

4.12.2.2 max_y

```
double InterpolatorStruct2D::max_y = 0
```

The maximum (i.e., last) element of y_vec.

4.12.2.3 min_x

```
double InterpolatorStruct2D::min_x = 0
```

The minimum (i.e., first) element of x_vec.

4.12.2.4 min_y

```
double InterpolatorStruct2D::min_y = 0
```

The minimum (i.e., first) element of y_vec.

4.12.2.5 n_cols

```
int InterpolatorStruct2D::n_cols = 0
```

The number of cols in the matrix (also the length of x_vec)

4.12.2.6 n_rows

```
int InterpolatorStruct2D::n_rows = 0
```

The number of rows in the matrix (also the length of y_vec)

4.12.2.7 x_vec

```
std::vector<double> InterpolatorStruct2D::x_vec = {}
```

A vector of independent data (columns).

4.12.2.8 y_vec

```
std::vector<double> InterpolatorStruct2D::y_vec = {}
```

A vector of independent data (rows).

4.12.2.9 z_matrix

```
std::vector<std::vector<double> > InterpolatorStruct2D::z_matrix = {}
```

A matrix of dependent data.

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

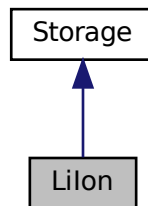
- header/[Interpolator.h](#)

4.13 Lilon Class Reference

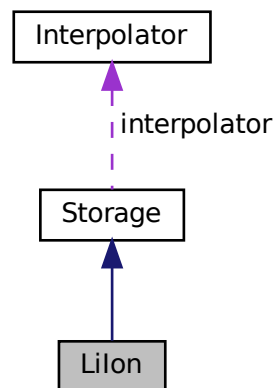
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

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

Inheritance diagram for Lilon:



Collaboration diagram for Lilon:



Public Member Functions

- [Lilon](#) (void)
Constructor (dummy) for the [Lilon](#) class.
- [Lilon](#) (int, double, [LilonInputs](#))
Constructor (intended) for the [Lilon](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [getAvailablekW](#) (double)

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

 - double `getAcceptablekW` (double)
- Method to get the charge power currently acceptable by the asset.*

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

 - double `commitDischarge` (int, double, double, double)
- Method which takes in the discharging power for the current timestep and records. Returns the load remaining after discharge.*

 - `~Lilon` (void)

Destructor for the `Lilon` class.

Public Attributes

- bool `power_degradation_flag`

A flag which indicates whether or not power degradation should be modelled.
- double `dynamic_energy_capacity_kWh`

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

The dynamic (i.e. degrading) power capacity [kW] of the asset.
- double `SOH`

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

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

A dimensionless acceleration coefficient used in modelling energy capacity degradation.
- double `degradation_beta`

A dimensionless acceleration exponent used in modelling energy capacity degradation.
- double `degradation_B_hat_cal_0`

A reference (or base) pre-exponential factor [$1/\sqrt{\text{hrs}}$] used in modelling energy capacity degradation.
- double `degradation_r_cal`

A dimensionless constant used in modelling energy capacity degradation.
- double `degradation_Ea_cal_0`

A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.
- double `degradation_a_cal`

A pre-exponential factor [J/mol] used in modelling energy capacity degradation.
- double `degradation_s_cal`

A dimensionless constant used in modelling energy capacity degradation.
- double `gas_constant_JmolK`

The universal gas constant [J/mol.K].
- double `temperature_K`

The absolute environmental temperature [K] of the lithium ion battery energy storage system.
- double `init_SOC`

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

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

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

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

- *The charging efficiency of the asset.*
double [discharging_efficiency](#)
- *The discharging efficiency of the asset.*
std::vector< double > [SOH_vec](#)
- *A vector of the state of health of the asset at each point in the modelling time series.*

Private Member Functions

- void [__checkInputs](#) ([LilonInputs](#))
Helper method to check inputs to the [Lilon](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic lithium ion battery energy storage system capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic lithium ion battery energy storage system operation and maintenance cost. This is a cost incurred per unit energy charged/discharged.
- void [__toggleDepleted](#) (void)
Helper method to toggle the is_depleted attribute of [Lilon](#).
- void [__handleDegradation](#) (int, double, double)
Helper method to apply degradation modelling and update attributes.
- void [__modelDegradation](#) (double, double)
Helper method to model energy capacity degradation as a function of operating state.
- double [__getBcal](#) (double)
Helper method to compute and return the base pre-exponential factor for a given state of charge.
- double [__getEacal](#) (double)
Helper method to compute and return the activation energy value for a given state of charge.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Lilon](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, int=-1)
Helper method to write time series results for [Lilon](#).

4.13.1 Detailed Description

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

4.13.2 Constructor & Destructor Documentation

4.13.2.1 [Lilon\(\)](#) [1/2]

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

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

```
674 {
675     return;
676 } /* LiIon() */
```

4.13.2.2 Lilon() [2/2]

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

Constructor (intended) for the [LiIon](#) 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>liion_inputs</i>	A structure of LiIon constructor inputs.

```
704 :
705 Storage(
706     n_points,
707     n_years,
708     liion_inputs.storage_inputs
709 )
710 {
711     // 1. check inputs
712     this->__checkInputs(liion_inputs);
713
714     // 2. set attributes
715     this->type = StorageType::LIION;
716     this->type_str = "LIION";
717
718     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
719     this->dynamic_power_capacity_kW = this->power_capacity_kW;
720
721     this->SOH = 1;
722     this->power_degradation_flag = liion_inputs.power_degradation_flag;
723     this->replace_SOH = liion_inputs.replace_SOH;
724
725     this->degradation_alpha = liion_inputs.degradation_alpha;
726     this->degradation_beta = liion_inputs.degradation_beta;
727     this->degradation_B_hat_cal_0 = liion_inputs.degradation_B_hat_cal_0;
728     this->degradation_r_cal = liion_inputs.degradation_r_cal;
729     this->degradation_Ea_cal_0 = liion_inputs.degradation_Ea_cal_0;
730     this->degradation_a_cal = liion_inputs.degradation_a_cal;
731     this->degradation_s_cal = liion_inputs.degradation_s_cal;
732     this->gas_constant_JmolK = liion_inputs.gas_constant_JmolK;
733     this->temperature_K = liion_inputs.temperature_K;
734
735     this->init_SOC = liion_inputs.init_SOC;
736     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
737
738     this->min_SOC = liion_inputs.min_SOC;
739     this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
740     this->max_SOC = liion_inputs.max_SOC;
741
742     this->charging_efficiency = liion_inputs.charging_efficiency;
743     this->discharging_efficiency = liion_inputs.discharging_efficiency;
744
745     if (liion_inputs.capital_cost < 0) {
746         this->capital_cost = this->__getGenericCapitalCost();
747     }
748     else {
749         this->capital_cost = liion_inputs.capital_cost;
750     }
751
752     if (liion_inputs.operation_maintenance_cost_kWh < 0) {
753         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
754     }
755     else {
756         this->operation_maintenance_cost_kWh =
757             liion_inputs.operation_maintenance_cost_kWh;
758     }
759
760     if (not this->is_sunk) {
761         this->capital_cost_vec[0] = this->capital_cost;
762     }
763
764     this->SOH_vec.resize(this->n_points, 0);
765
766     // 3. construction print
```

```

767     if (this->print_flag) {
768         std::cout << "LiIon object constructed at " << this << std::endl;
769     }
770
771     return;
772 } /* LiIon() */

```

4.13.2.3 ~LiIon()

```

LiIon::~~LiIon (
    void )

```

Destructor for the [LiIon](#) class.

```

1029 {
1030     // 1. destruction print
1031     if (this->print_flag) {
1032         std::cout << "LiIon object at " << this << " destroyed" << std::endl;
1033     }
1034
1035     return;
1036 } /* ~LiIon() */

```

4.13.3 Member Function Documentation

4.13.3.1 __checkInputs()

```

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

```

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

Parameters

<i>liion_inputs</i>	A structure of LiIon constructor inputs.
---------------------	--

```

64 {
65     // 1. check replace_SOH
66     if (liion_inputs.replace_SOH < 0 or liion_inputs.replace_SOH > 1) {
67         std::string error_str = "ERROR: LiIon(): replace_SOH must be in the closed ";
68         error_str += "interval [0, 1]";
69
70         #ifdef _WIN32
71             std::cout << error_str << std::endl;
72         #endif
73
74         throw std::invalid_argument(error_str);
75     }
76
77     // 2. check init_SOC
78     if (liion_inputs.init_SOC < 0 or liion_inputs.init_SOC > 1) {
79         std::string error_str = "ERROR: LiIon(): init_SOC must be in the closed ";
80         error_str += "interval [0, 1]";
81
82         #ifdef _WIN32
83             std::cout << error_str << std::endl;
84         #endif
85
86         throw std::invalid_argument(error_str);
87     }
88
89     // 3. check min_SOC

```



```

90     if (lilion_inputs.min_SOC < 0 or lilion_inputs.min_SOC > 1) {
91         std::string error_str = "ERROR: LiIon(): min_SOC must be in the closed ";
92         error_str += "interval [0, 1]";
93
94         #ifdef _WIN32
95             std::cout << error_str << std::endl;
96         #endif
97
98         throw std::invalid_argument(error_str);
99     }
100
101     // 4. check hysteresis_SOC
102     if (lilion_inputs.hysteresis_SOC < 0 or lilion_inputs.hysteresis_SOC > 1) {
103         std::string error_str = "ERROR: LiIon(): hysteresis_SOC must be in the closed ";
104         error_str += "interval [0, 1]";
105
106         #ifdef _WIN32
107             std::cout << error_str << std::endl;
108         #endif
109
110         throw std::invalid_argument(error_str);
111     }
112
113     // 5. check max_SOC
114     if (lilion_inputs.max_SOC < 0 or lilion_inputs.max_SOC > 1) {
115         std::string error_str = "ERROR: LiIon(): max_SOC must be in the closed ";
116         error_str += "interval [0, 1]";
117
118         #ifdef _WIN32
119             std::cout << error_str << std::endl;
120         #endif
121
122         throw std::invalid_argument(error_str);
123     }
124
125     // 6. check charging_efficiency
126     if (lilion_inputs.charging_efficiency <= 0 or lilion_inputs.charging_efficiency > 1) {
127         std::string error_str = "ERROR: LiIon(): charging_efficiency must be in the ";
128         error_str += "half-open interval (0, 1]";
129
130         #ifdef _WIN32
131             std::cout << error_str << std::endl;
132         #endif
133
134         throw std::invalid_argument(error_str);
135     }
136
137     // 7. check discharging_efficiency
138     if (
139         lilion_inputs.discharging_efficiency <= 0 or
140         lilion_inputs.discharging_efficiency > 1
141     ) {
142         std::string error_str = "ERROR: LiIon(): discharging_efficiency must be in the ";
143         error_str += "half-open interval (0, 1]";
144
145         #ifdef _WIN32
146             std::cout << error_str << std::endl;
147         #endif
148
149         throw std::invalid_argument(error_str);
150     }
151
152     // 8. check degradation_alpha
153     if (lilion_inputs.degradation_alpha <= 0) {
154         std::string error_str = "ERROR: LiIon(): degradation_alpha must be > 0";
155
156         #ifdef _WIN32
157             std::cout << error_str << std::endl;
158         #endif
159
160         throw std::invalid_argument(error_str);
161     }
162
163     // 9. check degradation_beta
164     if (lilion_inputs.degradation_beta <= 0) {
165         std::string error_str = "ERROR: LiIon(): degradation_beta must be > 0";
166
167         #ifdef _WIN32
168             std::cout << error_str << std::endl;
169         #endif
170
171         throw std::invalid_argument(error_str);
172     }
173
174     // 10. check degradation_B_hat_cal_0
175     if (lilion_inputs.degradation_B_hat_cal_0 <= 0) {
176         std::string error_str = "ERROR: LiIon(): degradation_B_hat_cal_0 must be > 0";

```

```

177
178     #ifdef _WIN32
179         std::cout << error_str << std::endl;
180     #endif
181
182     throw std::invalid_argument(error_str);
183 }
184
185 // 11. check degradation_r_cal
186 if (liion_inputs.degradation_r_cal < 0) {
187     std::string error_str = "ERROR: LiIon(): degradation_r_cal must be >= 0";
188
189     #ifdef _WIN32
190         std::cout << error_str << std::endl;
191     #endif
192
193     throw std::invalid_argument(error_str);
194 }
195
196 // 12. check degradation_Ea_cal_0
197 if (liion_inputs.degradation_Ea_cal_0 <= 0) {
198     std::string error_str = "ERROR: LiIon(): degradation_Ea_cal_0 must be > 0";
199
200     #ifdef _WIN32
201         std::cout << error_str << std::endl;
202     #endif
203
204     throw std::invalid_argument(error_str);
205 }
206
207 // 13. check degradation_a_cal
208 if (liion_inputs.degradation_a_cal < 0) {
209     std::string error_str = "ERROR: LiIon(): degradation_a_cal must be >= 0";
210
211     #ifdef _WIN32
212         std::cout << error_str << std::endl;
213     #endif
214
215     throw std::invalid_argument(error_str);
216 }
217
218 // 14. check degradation_s_cal
219 if (liion_inputs.degradation_s_cal < 0) {
220     std::string error_str = "ERROR: LiIon(): degradation_s_cal must be >= 0";
221
222     #ifdef _WIN32
223         std::cout << error_str << std::endl;
224     #endif
225
226     throw std::invalid_argument(error_str);
227 }
228
229 // 15. check gas_constant_JmolK
230 if (liion_inputs.gas_constant_JmolK <= 0) {
231     std::string error_str = "ERROR: LiIon(): gas_constant_JmolK must be > 0";
232
233     #ifdef _WIN32
234         std::cout << error_str << std::endl;
235     #endif
236
237     throw std::invalid_argument(error_str);
238 }
239
240 // 16. check temperature_K
241 if (liion_inputs.temperature_K < 0) {
242     std::string error_str = "ERROR: LiIon(): temperature_K must be >= 0";
243
244     #ifdef _WIN32
245         std::cout << error_str << std::endl;
246     #endif
247
248     throw std::invalid_argument(error_str);
249 }
250
251 return;
252 } /* __checkInputs() */

```

4.13.3.2 __getBcal()

```

double LiIon::__getBcal (
    double SOC ) [private]

```

Helper method to compute and return the base pre-exponential factor for a given state of charge.

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

Parameters

SOC	The current state of charge of the asset.
------------	---

Returns

The base pre-exponential factor for the given state of charge.

```

456 {
457     double B_cal = this->degradation_B_hat_cal_0 *
458         exp(this->degradation_r_cal * SOC);
459
460     return B_cal;
461 } /* __getBcal() */

```

4.13.3.3 __getEacal()

```

double LiIon::__getEacal (
    double SOC ) [private]

```

Helper method to compute and return the activation energy value for a given state of charge.

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

Parameters

SOC	The current state of charge of the asset.
------------	---

Returns

The activation energy value for the given state of charge.

```

483 {
484     double Ea_cal = this->degradation_Ea_cal_0;
485
486     Ea_cal -= this->degradation_a_cal *
487         (exp(this->degradation_s_cal * SOC) - 1);
488
489     return Ea_cal;
490 } /* __getEacal() */

```

4.13.3.4 __getGenericCapitalCost()

```

double LiIon::__getGenericCapitalCost (
    void ) [private]

```

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

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

Returns

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

```

275 {
276     double capital_cost_per_kWh = 250 * pow(this->energy_capacity_kWh, -0.15) + 650;
277
278     return capital_cost_per_kWh * this->energy_capacity_kWh;
279 } /* __getGenericCapitalCost() */

```

4.13.3.5 __getGenericOpMaintCost()

```

double LiIon::__getGenericOpMaintCost (
    void ) [private]

```

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

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

Returns

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

```

303 {
304     return 0.01;
305 } /* __getGenericOpMaintCost() */

```

4.13.3.6 __handleDegradation()

```

void LiIon::__handleDegradation (
    int timestep,
    double dt_hrs,
    double charging_discharging_kW ) [private]

```

Helper method to apply degradation modelling and update attributes.

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>charging_discharging_kW</i>	The charging/discharging power [kW] being sent to the asset.

```

373 {
374     // 1. model degradation
375     this->__modelDegradation(dt_hrs, charging_discharging_kW);
376
377     // 2. update and record
378     this->SOH_vec[timestep] = this->SOH;
379     this->dynamic_energy_capacity_kWh = this->SOH * this->energy_capacity_kWh;
380
381     if (this->power_degradation_flag) {
382         this->dynamic_power_capacity_kW = this->SOH * this->power_capacity_kW;
383     }

```

```

384
385     return;
386 } /* __handleDegradation() */

```

4.13.3.7 __modelDegradation()

```

void LiIon::__modelDegradation (
    double dt_hrs,
    double charging_discharging_kW ) [private]

```

Helper method to model energy capacity degradation as a function of operating state.

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

Parameters

<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
<i>charging_discharging_kW</i>	The charging/discharging power [kW] being sent to the asset.

```

409 {
410     // 1. compute SOC
411     double SOC = this->charge_kWh / this->energy_capacity_kWh;
412
413     // 2. compute C-rate and corresponding acceleration factor
414     double C_rate = charging_discharging_kW / this->power_capacity_kW;
415
416     double C_acceleration_factor =
417         1 + this->degradation_alpha * pow(C_rate, this->degradation_beta);
418
419     // 3. compute dSOH / dt
420     double B_cal = __getBcal(SOC);
421     double Ea_cal = __getEacal(SOC);
422
423     double dSOH_dt = B_cal *
424         exp((-1 * Ea_cal) / (this->gas_constant_JmolK * this->temperature_K));
425
426     dSOH_dt *= dSOH_dt;
427     dSOH_dt *= 1 / (2 * this->SOH);
428     dSOH_dt *= C_acceleration_factor;
429
430     // 4. update state of health
431     this->SOH -= dSOH_dt * dt_hrs;
432
433     return;
434 } /* __modelDegradation() */

```

4.13.3.8 __toggleDepleted()

```

void LiIon::__toggleDepleted (
    void ) [private]

```

Helper method to toggle the `is_depleted` attribute of `Lilon`.

```

320 {
321     if (this->is_depleted) {
322         double hysteresis_charge_kWh = this->hysteresis_SOC * this->energy_capacity_kWh;
323
324         if (hysteresis_charge_kWh > this->dynamic_energy_capacity_kWh) {
325             hysteresis_charge_kWh = this->dynamic_energy_capacity_kWh;
326         }
327
328         if (this->charge_kWh >= hysteresis_charge_kWh) {
329             this->is_depleted = false;

```

```

330     }
331 }
332
333 else {
334     double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
335
336     if (this->charge_kWh <= min_charge_kWh) {
337         this->is_depleted = true;
338     }
339 }
340
341 return;
342 } /* __toggleDepleted() */

```

4.13.3.9 __writeSummary()

```

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

```

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

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 [Storage](#).

```

508 {
509     // 1. create filestream
510     write_path += "summary_results.md";
511     std::ofstream ofs;
512     ofs.open(write_path, std::ofstream::out);
513
514     // 2. write summary results (markdown)
515     ofs << "# ";
516     ofs << std::to_string(int(ceil(this->power_capacity_kW)));
517     ofs << " kW ";
518     ofs << std::to_string(int(ceil(this->energy_capacity_kWh)));
519     ofs << " kWh LIION Summary Results\n";
520     ofs << "\n-----\n\n";
521
522     // 2.1. Storage attributes
523     ofs << "## Storage Attributes\n";
524     ofs << "\n";
525     ofs << "Power Capacity: " << this->power_capacity_kW << " kW \n";
526     ofs << "Energy Capacity: " << this->energy_capacity_kWh << " kWh \n";
527     ofs << "\n";
528
529     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
530     ofs << "Capital Cost: " << this->capital_cost << " \n";
531     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
532     << " per kWh charged/discharged \n";
533     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
534     << " \n";
535     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
536     << " \n";
537     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
538
539     ofs << "\n-----\n\n";
540
541     // 2.2. LiIon attributes
542     ofs << "## LiIon Attributes\n";
543     ofs << "\n";
544
545     ofs << "Charging Efficiency: " << this->charging_efficiency << " \n";
546     ofs << "Discharging Efficiency: " << this->discharging_efficiency << " \n";
547     ofs << "\n";
548
549     ofs << "Initial State of Charge: " << this->init_SOC << " \n";
550     ofs << "Minimum State of Charge: " << this->min_SOC << " \n";
551     ofs << "Hysteresis State of Charge: " << this->hysteresis_SOC << " \n";
552     ofs << "Maximum State of Charge: " << this->max_SOC << " \n";

```

```

553     ofs << "\n";
554
555     ofs << "Replacement State of Health: " << this->replace_SOH << " \n";
556     ofs << "\n";
557
558     ofs << "Degradation Acceleration Coeff.: " << this->degradation_alpha << " \n";
559     ofs << "Degradation Acceleration Exp.: " << this->degradation_beta << " \n";
560     ofs << "Degradation Base Pre-Exponential Factor: "
561         << this->degradation_B_hat_cal_0 << " 1/sqrt(hrs) \n";
562     ofs << "Degradation Dimensionless Constant (r_cal): "
563         << this->degradation_r_cal << " \n";
564     ofs << "Degradation Base Activation Energy: "
565         << this->degradation_Ea_cal_0 << " J/mol \n";
566     ofs << "Degradation Pre-Exponential Factor: "
567         << this->degradation_a_cal << " J/mol \n";
568     ofs << "Degradation Dimensionless Constant (s_cal): "
569         << this->degradation_s_cal << " \n";
570     ofs << "Universal Gas Constant: " << this->gas_constant_JmolK
571         << " J/mol.K \n";
572     ofs << "Absolute Environmental Temperature: " << this->temperature_K << " K \n";
573
574     ofs << "\n-----\n\n";
575
576     // 2.3. LiIon Results
577     ofs << "## Results\n";
578     ofs << "\n";
579
580     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
581     ofs << "\n";
582
583     ofs << "Total Discharge: " << this->total_discharge_kWh
584         << " kWh \n";
585
586     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
587         << " per kWh dispatched \n";
588     ofs << "\n";
589
590     ofs << "Replacements: " << this->n_replacements << " \n";
591
592     ofs << "\n-----\n\n";
593     ofs.close();
594     return;
595 } /* __writeSummary() */

```

4.13.3.10 __writeTimeSeries()

```

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

```

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

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 [Storage](#).

```

626 {
627     // 1. create filestream
628     write_path += "time_series_results.csv";
629     std::ofstream ofs;
630     ofs.open(write_path, std::ofstream::out);
631
632     // 2. write time series results (comma separated value)
633     ofs << "Time (since start of data) [hrs],";
634     ofs << "Charging Power [kW],";

```

```

635     ofs << "Discharging Power [kW],";
636     ofs << "Charge (at end of timestep) [kWh],";
637     ofs << "State of Health (at end of timestep) [ ],";
638     ofs << "Capital Cost (actual),";
639     ofs << "Operation and Maintenance Cost (actual),";
640     ofs << "\n";
641
642     for (int i = 0; i < max_lines; i++) {
643         ofs << time_vec_hrs_ptr->at(i) << ", ";
644         ofs << this->charging_power_vec_kW[i] << ", ";
645         ofs << this->discharging_power_vec_kW[i] << ", ";
646         ofs << this->charge_vec_kWh[i] << ", ";
647         ofs << this->SOH_vec[i] << ", ";
648         ofs << this->capital_cost_vec[i] << ", ";
649         ofs << this->operation_maintenance_cost_vec[i] << ", ";
650         ofs << "\n";
651     }
652
653     ofs.close();
654     return;
655 } /* __writeTimeSeries() */

```

4.13.3.11 commitCharge()

```

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

```

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

Parameters

<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>charging_kW</i>	The charging power [kw] being sent to the asset.

Reimplemented from [Storage](#).

```

920 {
921     // 1. record charging power
922     this->charging_power_vec_kW[timestep] = charging_kW;
923
924     // 2. update charge and record
925     this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
926     this->charge_vec_kWh[timestep] = this->charge_kWh;
927
928     // 3. toggle depleted flag (if applicable)
929     this->__toggleDepleted();
930
931     // 4. model degradation
932     this->__handleDegradation(timestep, dt_hrs, charging_kW);
933
934     // 5. trigger replacement (if applicable)
935     if (this->SOH <= this->replace_SOH) {
936         this->handleReplacement(timestep);
937     }
938
939     // 6. capture operation and maintenance costs (if applicable)
940     if (charging_kW > 0) {
941         this->operation_maintenance_cost_vec[timestep] = charging_kW * dt_hrs *
942             this->operation_maintenance_cost_kWh;
943     }
944
945     this->power_kW = 0;
946     return;
947 } /* commitCharge() */

```


4.13.3.12 commitDischarge()

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

Method which takes in the discharging power for the current timestep and records. Returns the load remaining after discharge.

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>discharging_kW</i>	The discharging power [kW] being drawn from the asset.
<i>load_kW</i>	The load [kW] passed to the asset in this timestep.

Returns

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

Reimplemented from [Storage](#).

```
983 {
984     // 1. record discharging power, update total
985     this->discharging_power_vec_kW[timestep] = discharging_kW;
986     this->total_discharge_kWh += discharging_kW * dt_hrs;
987
988     // 2. update charge and record
989     this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
990     this->charge_vec_kWh[timestep] = this->charge_kWh;
991
992     // 3. update load
993     load_kW -= discharging_kW;
994
995     // 4. toggle depleted flag (if applicable)
996     this->__toggleDepleted();
997
998     // 5. model degradation
999     this->__handleDegradation(timestep, dt_hrs, discharging_kW);
1000
1001     // 6. trigger replacement (if applicable)
1002     if (this->SOH <= this->replace_SOH) {
1003         this->handleReplacement(timestep);
1004     }
1005
1006     // 7. capture operation and maintenance costs (if applicable)
1007     if (discharging_kW > 0) {
1008         this->operation_maintenance_cost_vec[timestep] = discharging_kW * dt_hrs *
1009             this->operation_maintenance_cost_kWh;
1010     }
1011
1012     this->power_kW = 0;
1013     return load_kW;
1014 } /* commitDischarge() */
```

4.13.3.13 getAcceptablekW()

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

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

Parameters

<code>dt_hrs</code>	The interval of time [hrs] associated with the timestep.
---------------------	--

Returns

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

Reimplemented from [Storage](#).

```

864 {
865     // 1. get max charge
866     double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;
867
868     if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
869         max_charge_kWh = this->dynamic_energy_capacity_kWh;
870     }
871
872     // 2. compute acceptable power
873     // (accounting for the power currently being charged/discharged by the asset)
874     double acceptable_kW =
875         (max_charge_kWh - this->charge_kWh) /
876         (this->charging_efficiency * dt_hrs);
877
878     acceptable_kW -= this->power_kW;
879
880     if (acceptable_kW <= 0) {
881         return 0;
882     }
883
884     // 3. apply power constraint
885     if (acceptable_kW > this->dynamic_power_capacity_kW) {
886         acceptable_kW = this->dynamic_power_capacity_kW;
887     }
888
889     return acceptable_kW;
890 } /* getAcceptablekW( */

```

4.13.3.14 getAvailablekW()

```

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

```

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

Parameters

<code>dt_hrs</code>	The interval of time [hrs] associated with the timestep.
---------------------	--

Returns

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

Reimplemented from [Storage](#).

```

823 {
824     // 1. get min charge
825     double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
826
827     // 2. compute available power
828     // (accounting for the power currently being charged/discharged by the asset)
829     double available_kW =
830         ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
831         dt_hrs;

```

```

832
833     available_kW -= this->power_kW;
834
835     if (available_kW <= 0) {
836         return 0;
837     }
838
839     // 3. apply power constraint
840     if (available_kW > this->dynamic_power_capacity_kW) {
841         available_kW = this->dynamic_power_capacity_kW;
842     }
843
844     return available_kW;
845 } /* getAvailablekW() */

```

4.13.3.15 handleReplacement()

```

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

```

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

Parameters

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

Reimplemented from [Storage](#).

```

790 {
791     // 1. reset attributes
792     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
793     this->dynamic_power_capacity_kW = this->power_capacity_kW;
794     this->SOH = 1;
795
796     // 2. invoke base class method
797     Storage::handleReplacement(timestep);
798
799     // 3. correct attributes
800     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
801     this->is_depleted = false;
802
803     return;
804 } /* __handleReplacement() */

```

4.13.4 Member Data Documentation

4.13.4.1 charging_efficiency

```
double LiIon::charging_efficiency
```

The charging efficiency of the asset.

4.13.4.2 degradation_a_cal

```
double LiIon::degradation_a_cal
```

A pre-exponential factor [J/mol] used in modelling energy capacity degradation.

4.13.4.3 degradation_alpha

```
double LiIon::degradation_alpha
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.13.4.4 degradation_B_hat_cal_0

```
double LiIon::degradation_B_hat_cal_0
```

A reference (or base) pre-exponential factor [$1/\sqrt{\text{hrs}}$] used in modelling energy capacity degradation.

4.13.4.5 degradation_beta

```
double LiIon::degradation_beta
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.13.4.6 degradation_Ea_cal_0

```
double LiIon::degradation_Ea_cal_0
```

A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.

4.13.4.7 degradation_r_cal

```
double LiIon::degradation_r_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.13.4.8 degradation_s_cal

```
double LiIon::degradation_s_cal
```

A dimensionless constant used in modelling energy capacity degradation.

4.13.4.9 discharging_efficiency

```
double LiIon::discharging_efficiency
```

The discharging efficiency of the asset.

4.13.4.10 dynamic_energy_capacity_kWh

```
double LiIon::dynamic_energy_capacity_kWh
```

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

4.13.4.11 dynamic_power_capacity_kW

```
double LiIon::dynamic_power_capacity_kW
```

The dynamic (i.e. degrading) power capacity [kW] of the asset.

4.13.4.12 gas_constant_JmolK

```
double LiIon::gas_constant_JmolK
```

The universal gas constant [J/mol.K].

4.13.4.13 hysteresis_SOC

```
double LiIon::hysteresis_SOC
```

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

4.13.4.14 init_SOC

```
double LiIon::init_SOC
```

The initial state of charge of the asset.

4.13.4.15 max_SOC

```
double LiIon::max_SOC
```

The maximum state of charge of the asset.

4.13.4.16 min_SOC

```
double LiIon::min_SOC
```

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

4.13.4.17 power_degradation_flag

```
bool LiIon::power_degradation_flag
```

A flag which indicates whether or not power degradation should be modelled.

4.13.4.18 replace_SOH

```
double LiIon::replace_SOH
```

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

4.13.4.19 SOH

```
double LiIon::SOH
```

The state of health of the asset.

4.13.4.20 SOH_vec

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

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

4.13.4.21 temperature_K

```
double LiIon::temperature_K
```

The absolute environmental temperature [K] of the lithium ion battery energy storage system.

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

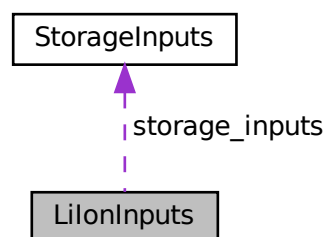
- [header/Storage/Lilon.h](#)
- [source/Storage/Lilon.cpp](#)

4.14 LilonInputs Struct Reference

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

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

Collaboration diagram for LilonInputs:



Public Attributes

- [StorageInputs storage_inputs](#)
An encapsulated [StorageInputs](#) instance.
- double [capital_cost](#) = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double [operation_maintenance_cost_kWh](#) = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double [init_SOC](#) = 0.5
The initial state of charge of the asset.
- double [min_SOC](#) = 0.15
The minimum state of charge of the asset. Will toggle `is_depleted` when reached.
- double [hysteresis_SOC](#) = 0.5
The state of charge the asset must achieve to toggle `is_depleted`.
- double [max_SOC](#) = 0.9
The maximum state of charge of the asset.
- double [charging_efficiency](#) = 0.9
The charging efficiency of the asset.
- double [discharging_efficiency](#) = 0.9
The discharging efficiency of the asset.
- double [replace_SOH](#) = 0.8
The state of health at which the asset is considered "dead" and must be replaced.
- bool [power_degradation_flag](#) = false
A flag which indicates whether or not power degradation should be modelled.
- double [degradation_alpha](#) = 8.935
A dimensionless acceleration coefficient used in modelling energy capacity degradation.
- double [degradation_beta](#) = 1
A dimensionless acceleration exponent used in modelling energy capacity degradation.
- double [degradation_B_hat_cal_0](#) = 5.22226e6
A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.
- double [degradation_r_cal](#) = 0.4361
A dimensionless constant used in modelling energy capacity degradation.
- double [degradation_Ea_cal_0](#) = 5.279e4
A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.
- double [degradation_a_cal](#) = 100
A pre-exponential factor [J/mol] used in modelling energy capacity degradation.
- double [degradation_s_cal](#) = 2
A dimensionless constant used in modelling energy capacity degradation.
- double [gas_constant_JmolK](#) = 8.31446
The universal gas constant [J/mol.K].
- double [temperature_K](#) = 273 + 20
The absolute environmental temperature [K] of the lithium ion battery energy storage system.

4.14.1 Detailed Description

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

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

4.14.2 Member Data Documentation

4.14.2.1 capital_cost

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

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

4.14.2.2 charging_efficiency

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

The charging efficiency of the asset.

4.14.2.3 degradation_a_cal

```
double LiIonInputs::degradation_a_cal = 100
```

A pre-exponential factor [J/mol] used in modelling energy capacity degradation.

4.14.2.4 degradation_alpha

```
double LiIonInputs::degradation_alpha = 8.935
```

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.14.2.5 degradation_B_hat_cal_0

```
double LiIonInputs::degradation_B_hat_cal_0 = 5.22226e6
```

A reference (or base) pre-exponential factor [1/sqrt(hrs)] used in modelling energy capacity degradation.

4.14.2.6 degradation_beta

```
double LiIonInputs::degradation_beta = 1
```

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.14.2.7 degradation_Ea_cal_0

```
double LiIonInputs::degradation_Ea_cal_0 = 5.279e4
```

A reference (or base) activation energy [J/mol] used in modelling energy capacity degradation.

4.14.2.8 degradation_r_cal

```
double LiIonInputs::degradation_r_cal = 0.4361
```

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.9 degradation_s_cal

```
double LiIonInputs::degradation_s_cal = 2
```

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.10 discharging_efficiency

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

The discharging efficiency of the asset.

4.14.2.11 gas_constant_JmolK

```
double LiIonInputs::gas_constant_JmolK = 8.31446
```

The universal gas constant [J/mol.K].

4.14.2.12 hysteresis_SOC

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

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

4.14.2.13 init_SOC

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

The initial state of charge of the asset.

4.14.2.14 max_SOC

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

The maximum state of charge of the asset.

4.14.2.15 min_SOC

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

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

4.14.2.16 operation_maintenance_cost_kWh

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

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

4.14.2.17 power_degradation_flag

```
bool LiIonInputs::power_degradation_flag = false
```

A flag which indicates whether or not power degradation should be modelled.

4.14.2.18 replace_SOH

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

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

4.14.2.19 storage_inputs

```
StorageInputs LiIonInputs::storage_inputs
```

An encapsulated [StorageInputs](#) instance.

4.14.2.20 temperature_K

```
double LiIonInputs::temperature_K = 273 + 20
```

The absolute environmental temperature [K] of the lithium ion battery energy storage system.

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

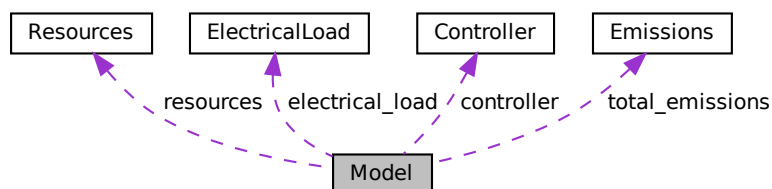
- header/Storage/[Lilon.h](#)

4.15 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](#) ([NoncombustionType](#), std::string, int)
A method to add a renewable resource time series to the [Model](#).
- void [addResource](#) ([RenewableType](#), std::string, int)
A method to add a renewable resource time series to the [Model](#).
- void [addHydro](#) ([HydroInputs](#))
Method to add a [Hydro](#) asset 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 [addLilon](#) ([LilonInputs](#))
Method to add a [Lilon](#) 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_renewable_dispatch_kWh](#)
The total energy dispatched [kWh] by all renewable assets over the [Model](#) run.
- 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< Noncombustion * > noncombustion_ptr_vec`
A vector of pointers to the various [Noncombustion](#) 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. Also tallies up total dispatch and discharge.
- `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](#).

4.15.1 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.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Model() [1/2]

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

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

```
598 {
599     return;
600 } /* Model() */
```

4.15.2.2 Model() [2/2]

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

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

Parameters

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

```
617 {
618     // 1. check inputs
619     this->__checkInputs(model_inputs);
620
621     // 2. read in electrical load data
622     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
623
624     // 3. set control mode
625     this->controller.setControlMode(model_inputs.control_mode);
626
627     // 4. set public attributes
628     this->total_fuel_consumed_L = 0;
629     this->net_present_cost = 0;
630     this->total_dispatch_discharge_kWh = 0;
631     this->total_renewable_dispatch_kWh = 0;
632     this->levellized_cost_of_energy_kWh = 0;
633
634     return;
635 } /* Model() */
```

4.15.2.3 ~Model()

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

Destructor for the [Model](#) class.

```
1154 {
1155     this->clear();
1156     return;
1157 } /* ~Model() */
```

4.15.3 Member Function Documentation

4.15.3.1 __checkInputs()

```
void Model::__checkInputs (
    ModelInputs model_inputs ) [private]
```

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

Parameters

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

```
65 {
66     // 1. check path_2_electrical_load_time_series
67     if (model_inputs.path_2_electrical_load_time_series.empty()) {
68         std::string error_str = "ERROR: Model() path_2_electrical_load_time_series ";
69         error_str += "cannot be empty";
70
71         #ifdef WIN32
72             std::cout << error_str << std::endl;
73         #endif
74
75         throw std::invalid_argument(error_str);
76     }
77
78     return;
79 } /* __checkInputs() */
```

4.15.3.2 __computeEconomics()

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

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

```
265 {
266     this->__computeNetPresentCost();
267     this->__computeLevellizedCostOfEnergy();
268
269     return;
270 } /* __computeEconomics() */
```

4.15.3.3 __computeFuelAndEmissions()

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

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

```
95 {
96     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
97         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
98
99         this->total_fuel_consumed_L +=
100             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
101
102         this->total_emissions.CO2_kg +=
103             this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
104
105         this->total_emissions.CO_kg +=
106             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
107
108         this->total_emissions.NOx_kg +=
109             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
110
111         this->total_emissions.SOx_kg +=
```



```

112         this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
113
114         this->total_emissions.CH4_kg +=
115         this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
116
117         this->total_emissions.PM_kg +=
118         this->combustion_ptr_vec[i]->total_emissions.PM_kg;
119     }
120
121     return;
122 } /* __computeFuelAndEmissions() */

```

4.15.3.4 __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.

```

212 {
213     // 1. account for Combustion economics in levellized cost of energy
214     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
215         this->levellized_cost_of_energy_kWh +=
216         (
217             this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
218             this->combustion_ptr_vec[i]->total_dispatch_kWh
219         ) / this->total_dispatch_discharge_kWh;
220     }
221
222     // 2. account for Noncombustion economics in levellized cost of energy
223     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
224         this->levellized_cost_of_energy_kWh +=
225         (
226             this->noncombustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
227             this->noncombustion_ptr_vec[i]->total_dispatch_kWh
228         ) / this->total_dispatch_discharge_kWh;
229     }
230
231     // 3. account for Renewable economics in levellized cost of energy
232     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
233         this->levellized_cost_of_energy_kWh +=
234         (
235             this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
236             this->renewable_ptr_vec[i]->total_dispatch_kWh
237         ) / this->total_dispatch_discharge_kWh;
238     }
239
240     // 4. account for Storage economics in levellized cost of energy
241     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
242         this->levellized_cost_of_energy_kWh +=
243         (
244             this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
245             this->storage_ptr_vec[i]->total_discharge_kWh
246         ) / this->total_dispatch_discharge_kWh;
247     }
248
249     return;
250 } /* __computeLevellizedCostOfEnergy() */

```

4.15.3.5 __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. Also tallies up total dispatch and discharge.

```

139 {
140     // 1. account for Combustion economics in net present cost

```

```

141 // increment total dispatch
142 for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
143     this->combustion_ptr_vec[i]->computeEconomics(
144         &(this->electrical_load.time_vec_hrs)
145     );
146
147     this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
148
149     this->total_dispatch_discharge_kWh +=
150         this->combustion_ptr_vec[i]->total_dispatch_kWh;
151 }
152
153 // 2. account for Noncombustion economics in net present cost
154 // increment total dispatch
155 for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
156     this->noncombustion_ptr_vec[i]->computeEconomics(
157         &(this->electrical_load.time_vec_hrs)
158     );
159
160     this->net_present_cost += this->noncombustion_ptr_vec[i]->net_present_cost;
161
162     this->total_dispatch_discharge_kWh +=
163         this->noncombustion_ptr_vec[i]->total_dispatch_kWh;
164 }
165
166 // 3. account for Renewable economics in net present cost,
167 // increment total dispatch
168 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
169     this->renewable_ptr_vec[i]->computeEconomics(
170         &(this->electrical_load.time_vec_hrs)
171     );
172
173     this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
174
175     this->total_dispatch_discharge_kWh +=
176         this->renewable_ptr_vec[i]->total_dispatch_kWh;
177
178     this->total_renewable_dispatch_kWh +=
179         this->renewable_ptr_vec[i]->total_dispatch_kWh;
180 }
181
182 // 4. account for Storage economics in net present cost
183 // increment total dispatch
184 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
185     this->storage_ptr_vec[i]->computeEconomics(
186         &(this->electrical_load.time_vec_hrs)
187     );
188
189     this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
190
191     this->total_dispatch_discharge_kWh +=
192         this->storage_ptr_vec[i]->total_discharge_kWh;
193 }
194
195 return;
196 } /* __computeNetPresentCost() */

```

4.15.3.6 __writeSummary()

```

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

```

Helper method to write summary 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.
-------------------	--

```

288 {
289     // 1. create subdirectory
290     write_path += "Model/";
291     std::filesystem::create_directory(write_path);
292 }

```

```

293 // 2. create filestream
294 write_path += "summary_results.md";
295 std::ofstream ofs;
296 ofs.open(write_path, std::ofstream::out);
297
298 // 3. write summary results (markdown)
299 ofs << "# Model Summary Results\n";
300 ofs << "\n-----\n\n";
301
302 // 3.1. ElectricalLoad
303 ofs << "## Electrical Load\n";
304 ofs << "\n";
305 ofs << "Path: " <<
306     this->electrical_load.path_2_electrical_load_time_series << " \n";
307 ofs << "Data Points: " << this->electrical_load.n_points << " \n";
308 ofs << "Years: " << this->electrical_load.n_years << " \n";
309 ofs << "Min: " << this->electrical_load.min_load_kW << " kW \n";
310 ofs << "Mean: " << this->electrical_load.mean_load_kW << " kW \n";
311 ofs << "Max: " << this->electrical_load.max_load_kW << " kW \n";
312 ofs << "\n-----\n\n";
313
314 // 3.2. Controller
315 ofs << "## Controller\n";
316 ofs << "\n";
317 ofs << "Control Mode: " << this->controller.control_string << " \n";
318 ofs << "\n-----\n\n";
319
320 // 3.3. Resources (1D)
321 ofs << "## 1D Renewable Resources\n";
322 ofs << "\n";
323
324 std::map<int, std::string>::iterator string_map_1D_iter =
325     this->resources.string_map_1D.begin();
326 std::map<int, std::string>::iterator path_map_1D_iter =
327     this->resources.path_map_1D.begin();
328
329 while (
330     string_map_1D_iter != this->resources.string_map_1D.end() and
331     path_map_1D_iter != this->resources.path_map_1D.end()
332 ) {
333     ofs << "Resource Key: " << string_map_1D_iter->first << " \n";
334     ofs << "Type: " << string_map_1D_iter->second << " \n";
335     ofs << "Path: " << path_map_1D_iter->second << " \n";
336     ofs << "\n";
337
338     string_map_1D_iter++;
339     path_map_1D_iter++;
340 }
341
342 ofs << "\n-----\n\n";
343
344 // 3.4. Resources (2D)
345 ofs << "## 2D Renewable Resources\n";
346 ofs << "\n";
347
348 std::map<int, std::string>::iterator string_map_2D_iter =
349     this->resources.string_map_2D.begin();
350 std::map<int, std::string>::iterator path_map_2D_iter =
351     this->resources.path_map_2D.begin();
352
353 while (
354     string_map_2D_iter != this->resources.string_map_2D.end() and
355     path_map_2D_iter != this->resources.path_map_2D.end()
356 ) {
357     ofs << "Resource Key: " << string_map_2D_iter->first << " \n";
358     ofs << "Type: " << string_map_2D_iter->second << " \n";
359     ofs << "Path: " << path_map_2D_iter->second << " \n";
360     ofs << "\n";
361
362     string_map_2D_iter++;
363     path_map_2D_iter++;
364 }
365
366 ofs << "\n-----\n\n";
367
368 // 3.5. Combustion
369 ofs << "## Combustion Assets\n";
370 ofs << "\n";
371
372 for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
373     ofs << "Asset Index: " << i << " \n";
374     ofs << "Type: " << this->combustion_ptr_vec[i]->type_str << " \n";
375     ofs << "Capacity: " << this->combustion_ptr_vec[i]->capacity_kW << " kW \n";
376     ofs << "\n";
377 }
378
379 ofs << "\n-----\n\n";

```

```

380
381 // 3.6. Noncombustion
382 ofs « "## Noncombustion Assets\n";
383 ofs « "\n";
384
385 for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
386     ofs « "Asset Index: " « i « " \n";
387     ofs « "Type: " « this->noncombustion_ptr_vec[i]->type_str « " \n";
388     ofs « "Capacity: " « this->noncombustion_ptr_vec[i]->capacity_kW « " kW \n";
389
390     if (this->noncombustion_ptr_vec[i]->type == NoncombustionType::HYDRO) {
391         ofs « "Reservoir Capacity: " «
392             ((Hydro*) (this->noncombustion_ptr_vec[i]))->reservoir_capacity_m3 «
393             " m3 \n";
394     }
395
396     ofs « "\n";
397 }
398
399 ofs « "\n-----\n\n";
400
401 // 3.7. Renewable
402 ofs « "## Renewable Assets\n";
403 ofs « "\n";
404
405 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
406     ofs « "Asset Index: " « i « " \n";
407     ofs « "Type: " « this->renewable_ptr_vec[i]->type_str « " \n";
408     ofs « "Capacity: " « this->renewable_ptr_vec[i]->capacity_kW « " kW \n";
409     ofs « "\n";
410 }
411
412 ofs « "\n-----\n\n";
413
414 // 3.8. Storage
415 ofs « "## Storage Assets\n";
416 ofs « "\n";
417
418 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
419     ofs « "Asset Index: " « i « " \n";
420     ofs « "Type: " « this->storage_ptr_vec[i]->type_str « " \n";
421     ofs « "Power Capacity: " « this->storage_ptr_vec[i]->power_capacity_kW
422         « " kW \n";
423     ofs « "Energy Capacity: " « this->storage_ptr_vec[i]->energy_capacity_kWh
424         « " kWh \n";
425     ofs « "\n";
426 }
427
428 ofs « "\n-----\n\n";
429
430 // 3.9. Model Results
431 ofs « "## Results\n";
432 ofs « "\n";
433
434 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
435 ofs « "\n";
436
437 ofs « "Total Dispatch + Discharge: " « this->total_dispatch_discharge_kWh
438     « " kWh \n";
439
440 ofs « "Renewable Penetration: "
441     « this->total_renewable_dispatch_kWh / this->total_dispatch_discharge_kWh
442     « " \n";
443 ofs « "\n";
444
445 ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
446     « " per kWh dispatched/discharged \n";
447 ofs « "\n";
448
449 ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
450     « "(Annual Average: " «
451         this->total_fuel_consumed_L / this->electrical_load.n_years
452         « " L/yr) \n";
453 ofs « "\n";
454
455 ofs « "Total Carbon Dioxide (CO2) Emissions: " «
456     this->total_emissions.CO2_kg « " kg "
457     « "(Annual Average: " «
458         this->total_emissions.CO2_kg / this->electrical_load.n_years
459         « " kg/yr) \n";
460
461 ofs « "Total Carbon Monoxide (CO) Emissions: " «
462     this->total_emissions.CO_kg « " kg "
463     « "(Annual Average: " «
464         this->total_emissions.CO_kg / this->electrical_load.n_years
465         « " kg/yr) \n";
466

```

```

467 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
468     this->total_emissions.NOx_kg << " kg "
469     << "(Annual Average: " <<
470         this->total_emissions.NOx_kg / this->electrical_load.n_years
471         << " kg/yr) \n";
472
473 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
474     this->total_emissions.SOx_kg << " kg "
475     << "(Annual Average: " <<
476         this->total_emissions.SOx_kg / this->electrical_load.n_years
477         << " kg/yr) \n";
478
479 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
480     << "(Annual Average: " <<
481         this->total_emissions.CH4_kg / this->electrical_load.n_years
482         << " kg/yr) \n";
483
484 ofs << "Total Particulate Matter (PM) Emissions: " <<
485     this->total_emissions.PM_kg << " kg "
486     << "(Annual Average: " <<
487         this->total_emissions.PM_kg / this->electrical_load.n_years
488         << " kg/yr) \n";
489
490 ofs << "\n-----\n\n";
491
492 ofs.close();
493 return;
494 } /* __writeSummary() */

```

4.15.3.7 __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.

```

514 {
515     // 1. create filestream
516     write_path += "Model/time_series_results.csv";
517     std::ofstream ofs;
518     ofs.open(write_path, std::ofstream::out);
519
520     // 2. write time series results header (comma separated value)
521     ofs << "Time (since start of data) [hrs],";
522     ofs << "Electrical Load [kW],";
523     ofs << "Net Load [kW],";
524     ofs << "Missed Load [kW],";
525
526     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
527         ofs << this->renewable_ptr_vec[i]->capacity_kW << " kW "
528             << this->renewable_ptr_vec[i]->type_str << " Dispatch [kW],";
529     }
530
531     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
532         ofs << this->storage_ptr_vec[i]->power_capacity_kW << " kW "
533             << this->storage_ptr_vec[i]->energy_capacity_kWh << " kWh "
534             << this->storage_ptr_vec[i]->type_str << " Discharge [kW],";
535     }
536
537     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
538         ofs << this->noncombustion_ptr_vec[i]->capacity_kW << " kW "
539             << this->noncombustion_ptr_vec[i]->type_str << " Dispatch [kW],";
540     }
541
542     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
543         ofs << this->combustion_ptr_vec[i]->capacity_kW << " kW "

```

```

544         « this->combustion_ptr_vec[i]->type_str « " Dispatch [kW],";
545     }
546
547     ofs « "\n";
548
549     // 3. write time series results values (comma separated value)
550     for (int i = 0; i < max_lines; i++) {
551         // 3.1. load values
552         ofs « this->electrical_load.time_vec_hrs[i] « ",";
553         ofs « this->electrical_load.load_vec_kW[i] « ",";
554         ofs « this->controller.net_load_vec_kW[i] « ",";
555         ofs « this->controller.missed_load_vec_kW[i] « ",";
556
557         // 3.2. asset-wise dispatch/discharge
558         for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
559             ofs « this->renewable_ptr_vec[j]->dispatch_vec_kW[i] « ",";
560         }
561
562         for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
563             ofs « this->storage_ptr_vec[j]->discharging_power_vec_kW[i] « ",";
564         }
565
566         for (size_t j = 0; j < this->noncombustion_ptr_vec.size(); j++) {
567             ofs « this->noncombustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
568         }
569
570         for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
571             ofs « this->combustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
572         }
573
574         ofs « "\n";
575     }
576
577     ofs.close();
578     return;
579 } /* __writeTimeSeries() */

```

4.15.3.8 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.
----------------------	---

```

652 {
653     Combustion* diesel_ptr = new Diesel(
654         this->electrical_load.n_points,
655         this->electrical_load.n_years,
656         diesel_inputs,
657         &(this->electrical_load.time_vec_hrs)
658     );
659
660     this->combustion_ptr_vec.push_back(diesel_ptr);
661
662     return;
663 } /* addDiesel() */

```

4.15.3.9 addHydro()

```

void Model::addHydro (
    HydroInputs hydro_inputs )

```

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

Parameters

<i>hydro_inputs</i>	A structure of Hydro constructor inputs.
---------------------	--

```

756 {
757     Noncombustion* hydro_ptr = new Hydro(
758         this->electrical_load.n_points,
759         this->electrical_load.n_years,
760         hydro_inputs,
761         &(this->electrical_load.time_vec_hrs)
762     );
763
764     this->noncombustion_ptr_vec.push_back(hydro_ptr);
765
766     return;
767 } /* addHydro() */

```

4.15.3.10 addLiIon()

```

void Model::addLiIon (
    LiIonInputs liion_inputs )

```

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

Parameters

<i>liion_inputs</i>	A structure of LiIon constructor inputs.
---------------------	--

```

896 {
897     Storage* liion_ptr = new LiIon(
898         this->electrical_load.n_points,
899         this->electrical_load.n_years,
900         liion_inputs
901     );
902
903     this->storage_ptr_vec.push_back(liion_ptr);
904
905     return;
906 } /* addLiIon() */

```

4.15.3.11 addResource() [1/2]

```

void Model::addResource (
    NoncombustionType noncombustion_type,
    std::string path_2_resource_data,
    int resource_key )

```

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

Parameters

<i>noncombustion_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.

```

692 {
693     resources.addResource(
694         noncombustion_type,
695         path_2_resource_data,
696         resource_key,
697         &(this->electrical_load)
698     );
699
700     return;
701 } /* addResource() */

```

4.15.3.12 addResource() [2/2]

```

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.

```

730 {
731     resources.addResource(
732         renewable_type,
733         path_2_resource_data,
734         resource_key,
735         &(this->electrical_load)
736     );
737
738     return;
739 } /* addResource() */

```

4.15.3.13 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.
---------------------	--

```

784 {
785     Renewable* solar_ptr = new Solar(
786         this->electrical_load.n_points,
787         this->electrical_load.n_years,
788         solar_inputs,
789         &(this->electrical_load.time_vec_hrs)
790     );
791
792     this->renewable_ptr_vec.push_back(solar_ptr);

```



```

793
794     return;
795 } /* addSolar() */

```

4.15.3.14 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.
---------------------	--

```

812 {
813     Renewable* tidal_ptr = new Tidal(
814         this->electrical_load.n_points,
815         this->electrical_load.n_years,
816         tidal_inputs,
817         &(this->electrical_load.time_vec_hrs)
818     );
819
820     this->renewable_ptr_vec.push_back(tidal_ptr);
821
822     return;
823 } /* addTidal() */

```

4.15.3.15 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.
--------------------	---

```

840 {
841     Renewable* wave_ptr = new Wave(
842         this->electrical_load.n_points,
843         this->electrical_load.n_years,
844         wave_inputs,
845         &(this->electrical_load.time_vec_hrs)
846     );
847
848     this->renewable_ptr_vec.push_back(wave_ptr);
849
850     return;
851 } /* addWave() */

```

4.15.3.16 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.
--------------------------	---

```

868 {
869     Renewable* wind_ptr = new Wind(
870         this->electrical_load.n_points,
871         this->electrical_load.n_years,
872         wind_inputs,
873         &(this->electrical_load.time_vec_hrs)
874     );
875
876     this->renewable_ptr_vec.push_back(wind_ptr);
877
878     return;
879 } /* addWind() */

```

4.15.3.17 clear()

```

void Model::clear (
    void )

```

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

```

1023 {
1024     // 1. reset
1025     this->reset();
1026
1027     // 2. clear components
1028     controller.clear();
1029     electrical_load.clear();
1030     resources.clear();
1031
1032     return;
1033 } /* clear() */

```

4.15.3.18 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.

```

965 {
966     // 1. clear combustion_ptr_vec
967     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
968         delete this->combustion_ptr_vec[i];
969     }
970     this->combustion_ptr_vec.clear();
971
972     // 2. clear noncombustion_ptr_vec
973     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
974         delete this->noncombustion_ptr_vec[i];
975     }
976     this->noncombustion_ptr_vec.clear();
977
978     // 3. clear renewable_ptr_vec
979     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
980         delete this->renewable_ptr_vec[i];
981     }
982     this->renewable_ptr_vec.clear();
983
984     // 4. clear storage_ptr_vec

```

```

985     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
986         delete this->storage_ptr_vec[i];
987     }
988     this->storage_ptr_vec.clear();
989
990     // 5. reset components and attributes
991     this->controller.clear();
992
993     this->total_fuel_consumed_L = 0;
994
995     this->total_emissions.CO2_kg = 0;
996     this->total_emissions.CO_kg = 0;
997     this->total_emissions.NOx_kg = 0;
998     this->total_emissions.SOx_kg = 0;
999     this->total_emissions.CH4_kg = 0;
1000     this->total_emissions.PM_kg = 0;
1001
1002     this->net_present_cost = 0;
1003     this->total_dispatch_discharge_kWh = 0;
1004     this->total_renewable_dispatch_kWh = 0;
1005     this->levellized_cost_of_energy_kWh = 0;
1006
1007     return;
1008 } /* reset() */

```

4.15.3.19 run()

```

void Model::run (
    void )

```

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

```

921 {
922     // 1. init Controller
923     this->controller.init(
924         &(this->electrical_load),
925         &(this->renewable_ptr_vec),
926         &(this->resources),
927         &(this->combustion_ptr_vec)
928     );
929
930     // 2. apply dispatch control
931     this->controller.applyDispatchControl(
932         &(this->electrical_load),
933         &(this->resources),
934         &(this->combustion_ptr_vec),
935         &(this->noncombustion_ptr_vec),
936         &(this->renewable_ptr_vec),
937         &(this->storage_ptr_vec)
938     );
939
940     // 3. compute total fuel consumption and emissions
941     this->__computeFuelAndEmissions();
942
943     // 4. compute key economic metrics
944     this->__computeEconomics();
945
946     return;
947 } /* run() */

```

4.15.3.20 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.

```

1061 {
1062     // 1. handle sentinel
1063     if (max_lines < 0) {
1064         max_lines = this->electrical_load.n_points;
1065     }
1066
1067     // 2. check for pre-existing, warn (and remove), then create
1068     if (write_path.back() != '/') {
1069         write_path += '/';
1070     }
1071
1072     if (std::filesystem::is_directory(write_path)) {
1073         std::string warning_str = "WARNING: Model::writeResults(): ";
1074         warning_str += write_path;
1075         warning_str += " already exists, contents will be overwritten!";
1076
1077         std::cout << warning_str << std::endl;
1078
1079         std::filesystem::remove_all(write_path);
1080     }
1081
1082     std::filesystem::create_directory(write_path);
1083
1084     // 3. write summary
1085     this->__writeSummary(write_path);
1086
1087     // 4. write time series
1088     if (max_lines > this->electrical_load.n_points) {
1089         max_lines = this->electrical_load.n_points;
1090     }
1091
1092     if (max_lines > 0) {
1093         this->__writeTimeSeries(write_path, max_lines);
1094     }
1095
1096     // 5. call out to Combustion :: writeResults()
1097     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
1098         this->combustion_ptr_vec[i]->writeResults(
1099             write_path,
1100             &(this->electrical_load.time_vec_hrs),
1101             i,
1102             max_lines
1103         );
1104     }
1105
1106     // 6. call out to Noncombustion :: writeResults()
1107     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
1108         this->noncombustion_ptr_vec[i]->writeResults(
1109             write_path,
1110             &(this->electrical_load.time_vec_hrs),
1111             i,
1112             max_lines
1113         );
1114     }
1115
1116     // 7. call out to Renewable :: writeResults()
1117     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
1118         this->renewable_ptr_vec[i]->writeResults(
1119             write_path,
1120             &(this->electrical_load.time_vec_hrs),
1121             &(this->resources.resource_map_1D),
1122             &(this->resources.resource_map_2D),
1123             i,
1124             max_lines
1125         );
1126     }
1127
1128     // 8. call out to Storage :: writeResults()
1129     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
1130         this->storage_ptr_vec[i]->writeResults(
1131             write_path,
1132             &(this->electrical_load.time_vec_hrs),
1133             i,
1134             max_lines
1135         );
1136     }
1137

```

```
1138     return;  
1139 } /* writeResults() */
```

4.15.4 Member Data Documentation

4.15.4.1 combustion_ptr_vec

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

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

4.15.4.2 controller

```
Controller Model::controller
```

[Controller](#) component of [Model](#).

4.15.4.3 electrical_load

```
ElectricalLoad Model::electrical_load
```

[ElectricalLoad](#) component of [Model](#).

4.15.4.4 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).

4.15.4.5 net_present_cost

```
double Model::net_present_cost
```

The net present cost of the [Model](#) (undefined currency).

4.15.4.6 noncombustion_ptr_vec

```
std::vector<Noncombustion*> Model::noncombustion_ptr_vec
```

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

4.15.4.7 renewable_ptr_vec

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

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

4.15.4.8 resources

```
Resources Model::resources
```

[Resources](#) component of [Model](#).

4.15.4.9 storage_ptr_vec

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

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

4.15.4.10 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

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

4.15.4.11 total_emissions

```
Emissions Model::total_emissions
```

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

4.15.4.12 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

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

4.15.4.13 total_renewable_dispatch_kWh

```
double Model::total_renewable_dispatch_kWh
```

The total energy dispatched [kWh] by all renewable assets over the [Model](#) run.

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

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

4.16 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.

4.16.1 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).

4.16.2 Member Data Documentation

4.16.2.1 control_mode

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

The control mode to be applied by the [Controller](#) object.

4.16.2.2 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:

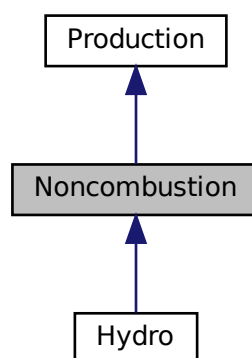
- [header/Model.h](#)

4.17 Noncombustion Class Reference

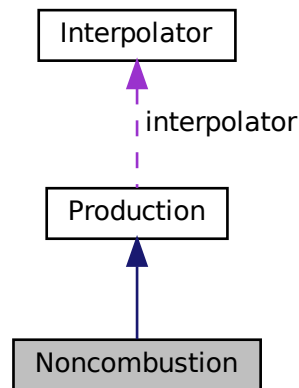
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

```
#include <Noncombustion.h>
```

Inheritance diagram for Noncombustion:



Collaboration diagram for Noncombustion:



Public Member Functions

- [Noncombustion](#) (void)
Constructor (dummy) for the [Noncombustion](#) class.
- [Noncombustion](#) (int, double, [NoncombustionInputs](#), std::vector< double > *)
Constructor (intended) for the [Noncombustion](#) class.
- virtual void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [requestProductionkW](#) (int, double, double)
- virtual double [requestProductionkW](#) (int, double, double, double)
- virtual double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual double [commit](#) (int, double, double, double, double)
- void [writeResults](#) (std::string, std::vector< double > *, int, int=-1)
Method which writes [Noncombustion](#) results to an output directory.
- virtual [~Noncombustion](#) (void)
Destructor for the [Noncombustion](#) class.

Public Attributes

- [NoncombustionType](#) type
The type ([NoncombustionType](#)) 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` ([NoncombustionInputs](#))
Helper method to check inputs to the [Noncombustion](#) constructor.
- void `__handleStartStop` (int, double, double)
Helper method to handle the starting/stopping of the [Noncombustion](#) asset.
- virtual void `__writeSummary` (std::string)
- virtual void `__writeTimeSeries` (std::string, std::vector< double > *, int=-1)

4.17.1 Detailed Description

The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

4.17.2 Constructor & Destructor Documentation

4.17.2.1 `Noncombustion()` [1/2]

```
Noncombustion::Noncombustion (
    void )
```

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

```
127 {
128     return;
129 } /* Noncombustion() */
```

4.17.2.2 `Noncombustion()` [2/2]

```
Noncombustion::Noncombustion (
    int n_points,
    double n_years,
    NoncombustionInputs noncombustion_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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

Parameters

<code>n_points</code>	The number of points in the modelling time series.
<code>n_years</code>	The number of years being modelled.
<code>noncombustion_inputs</code>	A structure of Noncombustion constructor inputs.
<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.

```
161 :
162 Production(
163     n_points,
```

```

164     n_years,
165     noncombustion_inputs.production_inputs,
166     time_vec_hrs_ptr
167 )
168 {
169     // 1. check inputs
170     this->__checkInputs(noncombustion_inputs);
171
172     // 2. set attributes
173     //...
174
175     // 3. construction print
176     if (this->print_flag) {
177         std::cout << "Noncombustion object constructed at " << this << std::endl;
178     }
179
180     return;
181 } /* Noncombustion() */

```

4.17.2.3 ~Noncombustion()

```

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

```

Destructor for the [Noncombustion](#) class.

```

372 {
373     // 1. destruction print
374     if (this->print_flag) {
375         std::cout << "Noncombustion object at " << this << " destroyed" << std::endl;
376     }
377
378     return;
379 } /* ~Noncombustion() */

```

4.17.3 Member Function Documentation

4.17.3.1 __checkInputs()

```

void Noncombustion::__checkInputs (
    NoncombustionInputs noncombustion_inputs ) [private]

```

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

Parameters

<i>noncombustion_inputs</i>	A structure of Noncombustion constructor inputs.
-----------------------------	--

```

64 {
65     //...
66
67     return;
68 } /* __checkInputs() */

```

4.17.3.2 __handleStartStop()

```

void Noncombustion::__handleStartStop (

```

```

    int timestep,
    double dt_hrs,
    double production_kW ) [private]

```

Helper method to handle the starting/stopping of the [Noncombustion](#) asset.

```

91 {
92     if (this->is_running) {
93         // handle stopping
94         if (production_kW <= 0) {
95             this->is_running = false;
96         }
97     }
98
99     else {
100        // handle starting
101        if (production_kW > 0) {
102            this->is_running = true;
103            this->n_starts++;
104        }
105    }
106
107    return;
108 } /* __handleStartStop() */

```

4.17.3.3 __writeSummary()

```

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

```

Reimplemented in [Hydro](#).

```

95 {return;}

```

4.17.3.4 __writeTimeSeries()

```

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

```

Reimplemented in [Hydro](#).

```

100     {return;}

```

4.17.3.5 commit() [1/2]

```

double Noncombustion::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](#).

```

267 {
268     // 1. handle start/stop
269     this->__handleStartStop(timestep, dt_hrs, production_kW);
270
271     // 2. invoke base class method
272     load_kW = Production::commit(
273         timestep,
274         dt_hrs,
275         production_kW,
276         load_kW
277     );
278
279
280     //...
281
282     return load_kW;
283 } /* commit() */

```

4.17.3.6 commit() [2/2]

```

virtual double Noncombustion::commit (
    int ,
    double ,
    double ,
    double ,
    double ) [inline], [virtual]

```

Reimplemented in [Hydro](#).

```

121 {return 0;}

```

4.17.3.7 computeEconomics()

```

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

```

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

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

Parameters

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

Reimplemented from [Production](#).

```

226 {
227     // 1. invoke base class method
228     Production :: computeEconomics(time_vec_hrs_ptr);
229
230     return;
231 } /* computeEconomics() */

```

4.17.3.8 [handleReplacement\(\)](#)

```

void Noncombustion::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Production](#).

Reimplemented in [Hydro](#).

```

199 {
200     // 1. reset attributes
201     //...
202
203     // 2. invoke base class method
204     Production :: handleReplacement(timestep);
205
206     return;
207 } /* \_\_handleReplacement() */

```

4.17.3.9 [requestProductionkW\(\)](#) [1/2]

```

virtual double Noncombustion::requestProductionkW (
    int ,
    double ,
    double ) [inline], [virtual]
117 {return 0;}

```

4.17.3.10 [requestProductionkW\(\)](#) [2/2]

```

virtual double Noncombustion::requestProductionkW (
    int ,
    double ,
    double ,
    double ) [inline], [virtual]

```

Reimplemented in [Hydro](#).

```

118 {return 0;}

```

4.17.3.11 writeResults()

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

Method which writes [Noncombustion](#) 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>noncombustion_index</i>	An integer which corresponds to the index of the Noncombustion 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.

```
319 {
320     // 1. handle sentinel
321     if (max_lines < 0) {
322         max_lines = this->n_points;
323     }
324
325     // 2. create subdirectories
326     write_path += "Production/";
327     if (not std::filesystem::is_directory(write_path)) {
328         std::filesystem::create_directory(write_path);
329     }
330
331     write_path += "Noncombustion/";
332     if (not std::filesystem::is_directory(write_path)) {
333         std::filesystem::create_directory(write_path);
334     }
335
336     write_path += this->type_str;
337     write_path += "_";
338     write_path += std::to_string(int(ceil(this->capacity_kW)));
339     write_path += "kW_idx";
340     write_path += std::to_string(combustion_index);
341     write_path += "/";
342     std::filesystem::create_directory(write_path);
343
344     // 3. write summary
345     this->__writeSummary(write_path);
346
347     // 4. write time series
348     if (max_lines > this->n_points) {
349         max_lines = this->n_points;
350     }
351
352     if (max_lines > 0) {
353         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
354     }
355
356     return;
357 } /* writeResults() */
```

4.17.4 Member Data Documentation

4.17.4.1 resource_key

```
int Noncombustion::resource_key
```

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

4.17.4.2 type

`NoncombustionType` `Noncombustion::type`

The type (`NoncombustionType`) of the asset.

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

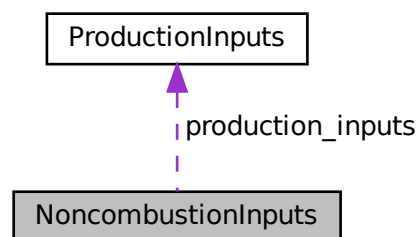
- [header/Production/Noncombustion/Noncombustion.h](#)
- [source/Production/Noncombustion/Noncombustion.cpp](#)

4.18 NoncombustionInputs Struct Reference

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

```
#include <Noncombustion.h>
```

Collaboration diagram for `NoncombustionInputs`:



Public Attributes

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

4.18.1 Detailed Description

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

4.18.2 Member Data Documentation

4.18.2.1 production_inputs

`ProductionInputs` `NoncombustionInputs::production_inputs`

An encapsulated `ProductionInputs` instance.

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

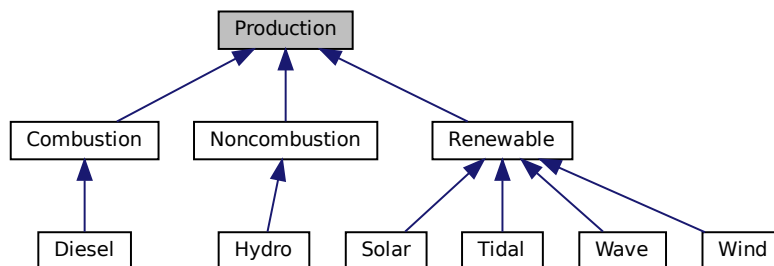
- `header/Production/Noncombustion/Noncombustion.h`

4.19 Production Class Reference

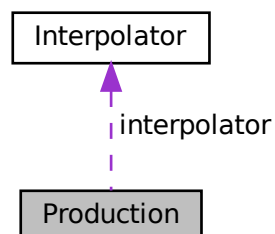
The base class of the `Production` hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

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

Inheritance diagram for `Production`:



Collaboration diagram for `Production`:



Public Member Functions

- [Production](#) (void)
Constructor (dummy) for the [Production](#) class.
- [Production](#) (int, double, [ProductionInputs](#), std::vector< double > *)
Constructor (intended) for the [Production](#) class.
- virtual void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeRealDiscountAnnual](#) (double, double)
Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- double [getProductionkW](#) (int)
A method to simply fetch the normalized production at a particular point in the given normalized production time series, multiply by the rated capacity of the asset, and return.
- 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

- [Interpolator](#) [interpolator](#)
[Interpolator](#) component of [Production](#).
- 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).
- bool [normalized_production_series_given](#)
A boolean which indicates whether or not a normalized production time series is given.
- 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 dispatch.
- std::string [type_str](#)
A string describing the type of the asset.
- std::string [path_2_normalized_production_time_series](#)
A string defining the path (either relative or absolute) to the given normalized production time series.
- 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 > [normalized_production_vec](#)
A vector of normalized production [] at each point in the modelling time series.
- std::vector< double > [production_vec_kW](#)
A vector of production [kW] at each point in the modelling time series.
- std::vector< double > [dispatch_vec_kW](#)
A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.
- std::vector< double > [storage_vec_kW](#)
A vector of storage [kW] at each point in the modelling time series. [Storage](#) is the amount of production that is sent to storage.
- std::vector< double > [curtailment_vec_kW](#)
A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.
- std::vector< double > [capital_cost_vec](#)
A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > [operation_maintenance_cost_vec](#)
A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- void [__checkInputs](#) (int, double, [ProductionInputs](#))
Helper method to check inputs to the [Production](#) constructor.
- void [__checkTimePoint](#) (double, double)
Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.
- void [__throwLengthError](#) (void)
Helper method to throw data length error (if not the same as the given electrical load time series).
- void [__checkNormalizedProduction](#) (double)

Helper method to check that given data values are everywhere contained in the closed interval $[0, 1]$. A normalized production time series is expected, so this must be true everywhere.

- void `__readNormalizedProductionData` (std::vector< double > *)

Helper method to read in a given time series of normalized production.

4.19.1 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.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 `Production()` [1/2]

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

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

```
307 {
308     return;
309 } /* Production() */
```

4.19.2.2 `Production()` [2/2]

```
Production::Production (
    int n_points,
    double n_years,
    ProductionInputs production_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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

Parameters

<code>n_points</code>	The number of points in the modelling time series.
<code>n_years</code>	The number of years being modelled.
<code>production_inputs</code>	A structure of Production constructor inputs.
<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.

```
342 {
343     // 1. check inputs
344     this->__checkInputs(n_points, n_years, production_inputs);
345
346     // 2. set attributes
347     this->print_flag = production_inputs.print_flag;
348     this->is_running = false;
349     this->is_sunk = production_inputs.is_sunk;
350     this->normalized_production_series_given = false;
351 }
```

```

352     this->n_points = n_points;
353     this->n_starts = 0;
354     this->n_replacements = 0;
355
356     this->n_years = n_years;
357
358     this->running_hours = 0;
359     this->replace_running_hrs = production_inputs.replace_running_hrs;
360
361     this->capacity_kW = production_inputs.capacity_kW;
362
363     this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
364     this->nominal_discount_annual = production_inputs.nominal_discount_annual;
365
366     this->real_discount_annual = this->computeRealDiscountAnnual(
367         production_inputs.nominal_inflation_annual,
368         production_inputs.nominal_discount_annual
369     );
370
371     this->capital_cost = 0;
372     this->operation_maintenance_cost_kWh = 0;
373     this->net_present_cost = 0;
374     this->total_dispatch_kWh = 0;
375     this->levellized_cost_of_energy_kWh = 0;
376
377     this->path_2_normalized_production_time_series = "";
378
379     this->is_running_vec.resize(this->n_points, 0);
380
381     this->normalized_production_vec.resize(this->n_points, 0);
382     this->production_vec_kW.resize(this->n_points, 0);
383     this->dispatch_vec_kW.resize(this->n_points, 0);
384     this->storage_vec_kW.resize(this->n_points, 0);
385     this->curtailment_vec_kW.resize(this->n_points, 0);
386
387     this->capital_cost_vec.resize(this->n_points, 0);
388     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
389
390     // 3. read in normalized production time series (if given)
391     if (not production_inputs.path_2_normalized_production_time_series.empty()) {
392         this->normalized_production_series_given = true;
393
394         this->path_2_normalized_production_time_series =
395             production_inputs.path_2_normalized_production_time_series;
396
397         this->__readNormalizedProductionData(time_vec_hrs_ptr);
398     }
399
400     // 4. construction print
401     if (this->print_flag) {
402         std::cout << "Production object constructed at " << this << std::endl;
403     }
404
405     return;
406 } /* Production() */

```

4.19.2.3 ~Production()

```

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

```

Destructor for the [Production](#) class.

```

655 {
656     // 1. destruction print
657     if (this->print_flag) {
658         std::cout << "Production object at " << this << " destroyed" << std::endl;
659     }
660
661     return;
662 } /* ~Production() */

```

4.19.3 Member Function Documentation

4.19.3.1 `__checkInputs()`

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

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

Parameters

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

```
70 {
71     // 1. check n_points
72     if (n_points <= 0) {
73         std::string error_str = "ERROR: Production(): n_points must be > 0";
74
75         #ifdef _WIN32
76             std::cout << error_str << std::endl;
77         #endif
78
79         throw std::invalid_argument(error_str);
80     }
81
82     // 2. check n_years
83     if (n_years <= 0) {
84         std::string error_str = "ERROR: Production(): n_years must be > 0";
85
86         #ifdef _WIN32
87             std::cout << error_str << std::endl;
88         #endif
89
90         throw std::invalid_argument(error_str);
91     }
92
93     // 3. check capacity_kW
94     if (production_inputs.capacity_kW <= 0) {
95         std::string error_str = "ERROR: Production(): ";
96         error_str += "ProductionInputs::capacity_kW must be > 0";
97
98         #ifdef _WIN32
99             std::cout << error_str << std::endl;
100        #endif
101
102        throw std::invalid_argument(error_str);
103    }
104
105    // 4. check replace_running_hrs
106    if (production_inputs.replace_running_hrs <= 0) {
107        std::string error_str = "ERROR: Production(): ";
108        error_str += "ProductionInputs::replace_running_hrs must be > 0";
109
110        #ifdef _WIN32
111            std::cout << error_str << std::endl;
112        #endif
113
114        throw std::invalid_argument(error_str);
115    }
116
117    return;
118 } /* __checkInputs() */
```

4.19.3.2 `__checkNormalizedProduction()`

```
void Production::__checkNormalizedProduction (
    double normalized_production ) [private]
```

Helper method to check that given data values are everywhere contained in the closed interval [0, 1]. A normalized production time series is expected, so this must be true everywhere.

Parameters

<i>normalized_production</i>	The normalized production value to check
------------------------------	--

```

210 {
211     if (normalized_production < 0 or normalized_production > 1) {
212         std::string error_str = "ERROR: Production(): ";
213         error_str += "the given normalized production time series at ";
214         error_str += this->path_2_normalized_production_time_series;
215         error_str += " contains normalized production values outside the closed ";
216         error_str += "interval [0, 1]";
217
218         #ifdef _WIN32
219             std::cout << error_str << std::endl;
220         #endif
221
222         throw std::runtime_error(error_str);
223     }
224
225     return;
226 } /* __throwValueError() */

```

4.19.3.3 __checkTimePoint()

```

void Production::__checkTimePoint (
    double time_received_hrs,
    double time_expected_hrs ) [private]

```

Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.

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).

```

146 {
147     if (time_received_hrs != time_expected_hrs) {
148         std::string error_str = "ERROR: Production(): ";
149         error_str += "the given normalized production time series at ";
150         error_str += this->path_2_normalized_production_time_series;
151         error_str += " does not align with the ";
152         error_str += "previously given electrical load time series";
153
154         #ifdef _WIN32
155             std::cout << error_str << std::endl;
156         #endif
157
158         throw std::runtime_error(error_str);
159     }
160
161     return;
162 } /* __checkTimePoint() */

```

4.19.3.4 __readNormalizedProductionData()

```

void Production::__readNormalizedProductionData (
    std::vector< double > * time_vec_hrs_ptr ) [private]

```

Helper method to read in a given time series of normalized production.

Parameters

<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.
-------------------------------	---

```

247 {
248     // 1. init CSV reader
249     io::CSVReader<2> CSV(this->path_2_normalized_production_time_series);
250
251     CSV.read_header(
252         io::ignore_extra_column,
253         "Time (since start of data) [hrs]",
254         "Normalized Production [ ]"
255     );
256
257     // 2. read in normalized performance data,
258     //     check values and check against time series (point-wise and length)
259     int n_points = 0;
260     double time_hrs = 0;
261     double time_expected_hrs = 0;
262     double normalized_production = 0;
263
264     while (CSV.read_row(time_hrs, normalized_production)) {
265         // 2.1. check length of data
266         if (n_points > this->n_points) {
267             this->__throwLengthError();
268         }
269
270         // 2.2. check normalized production value
271         this->__checkNormalizedProduction(normalized_production);
272
273         // 2.3. check time point
274         time_expected_hrs = time_vec_hrs_ptr->at(n_points);
275         this->__checkTimePoint(time_hrs, time_expected_hrs);
276
277         // 2.4. write to normalized production vector, increment n_points
278         this->normalized_production_vec[n_points] = normalized_production;
279         n_points++;
280     }
281
282     // 3. check length of data
283     if (n_points != this->n_points) {
284         this->__throwLengthError();
285     }
286
287     return;
288 } /* __readNormalizedProductionData() */

```

4.19.3.5 __throwLengthError()

```

void Production::__throwLengthError (
    void ) [private]

```

Helper method to throw data length error (if not the same as the given electrical load time series).

```

177 {
178     std::string error_str = "ERROR: Production(): ";
179     error_str += "the given normalized production time series at ";
180     error_str += this->path_2_normalized_production_time_series;
181     error_str += " is not the same length as the previously given electrical";
182     error_str += " load time series";
183
184     #ifdef _WIN32
185         std::cout << error_str << std::endl;
186     #endif
187
188     throw std::runtime_error(error_str);
189
190     return;
191 } /* __throwLengthError() */

```


4.19.3.6 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](#), [Noncombustion](#), [Diesel](#), and [Combustion](#).

```
596 {
597     // 1. record production
598     this->production_vec_kW[timestep] = production_kW;
599
600     // 2. compute and record dispatch and curtailment
601     double dispatch_kW = 0;
602     double curtailment_kW = 0;
603
604     if (production_kW > load_kW) {
605         dispatch_kW = load_kW;
606         curtailment_kW = production_kW - dispatch_kW;
607     }
608
609     else {
610         dispatch_kW = production_kW;
611     }
612
613     this->dispatch_vec_kW[timestep] = dispatch_kW;
614     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
615     this->curtailment_vec_kW[timestep] = curtailment_kW;
616
617     // 3. update load
618     load_kW -= dispatch_kW;
619
620     // 4. update and log running attributes
621     if (this->is_running) {
622         // 4.1. log running state, running hours
623         this->is_running_vec[timestep] = this->is_running;
624         this->running_hours += dt_hrs;
625
626         // 4.2. incur operation and maintenance costs
627         double produced_kWh = production_kW * dt_hrs;
628
629         double operation_maintenance_cost =
630             this->operation_maintenance_cost_kWh * produced_kWh;
631         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
632     }
633
634     // 5. trigger replacement, if applicable
635     if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
636         this->handleReplacement(timestep);
637     }
638
639     return load_kW;
640 } /* commit() */
```

4.19.3.7 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](#), [Noncombustion](#), and [Combustion](#).

```
494 {
495     // 1. compute net present cost
496     double t_hrs = 0;
497     double real_discount_scalar = 0;
498
499     for (int i = 0; i < this->n_points; i++) {
500         t_hrs = time_vec_hrs_ptr->at(i);
501
502         real_discount_scalar = 1.0 / pow(
503             1 + this->real_discount_annual,
504             t_hrs / 8760
505         );
506
507         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
508
509         this->net_present_cost +=
510             real_discount_scalar * this->operation_maintenance_cost_vec[i];
511     }
512
513     // assuming 8,760 hours per year
514     if (this->total_dispatch_kWh <= 0) {
515         this->levellized_cost_of_energy_kWh = this->net_present_cost;
516     }
517
518     else {
519         double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
520
521         double capital_recovery_factor =
522             (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
523             (pow(1 + this->real_discount_annual, n_years) - 1);
524
525         double total_annualized_cost = capital_recovery_factor *
526             this->net_present_cost;
527
528         this->levellized_cost_of_energy_kWh =
529             (n_years * total_annualized_cost) /
530             this->total_dispatch_kWh;
531     }
532 }
533
534 return;
535 } /* computeEconomics() */
```

4.19.3.8 computeRealDiscountAnnual()

```
double Production::computeRealDiscountAnnual (
    double nominal_inflation_annual,
    double nominal_discount_annual )
```

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

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

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

Parameters

<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.

```

467 {
468     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
469     real_discount_annual /= 1 + nominal_inflation_annual;
470
471     return real_discount_annual;
472 } /* __computeRealDiscountAnnual() */

```

4.19.3.9 getProductionkW()

```

double Production::getProductionkW (
    int timestep )

```

A method to simply fetch the normalized production at a particular point in the given normalized production time series, multiply by the rated capacity of the asset, and return.

Returns

The production [kW] for the asset at the given point in time, as defined by the given normalized production time series.

```

555 {
556     double production_kW =
557         this->normalized_production_vec[timestep] * this->capacity_kW;
558
559     return production_kW;
560 } /* getProductionkW() */

```

4.19.3.10 handleReplacement()

```

void Production::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented in [Wind](#), [Wave](#), [Tidal](#), [Solar](#), [Renewable](#), [Noncombustion](#), [Hydro](#), [Diesel](#), and [Combustion](#).

```

424 {
425     // 1. reset attributes
426     this->is_running = false;
427
428     // 2. log replacement
429     this->n_replacements++;
430
431     // 3. incur capital cost in timestep
432     this->capital_cost_vec[timestep] = this->capital_cost;
433
434     return;
435 } /* __handleReplacement() */

```

4.19.4 Member Data Documentation

4.19.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.19.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.19.4.3 capital_cost_vec

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

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

4.19.4.4 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.

4.19.4.5 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.

4.19.4.6 interpolator

```
Interpolator Production::interpolator
```

[Interpolator](#) component of [Production](#).

4.19.4.7 is_running

```
bool Production::is_running
```

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

4.19.4.8 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.

4.19.4.9 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).

4.19.4.10 levlized_cost_of_energy_kWh

```
double Production::levlized_cost_of_energy_kWh
```

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

4.19.4.11 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.19.4.12 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.19.4.13 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.19.4.14 n_years

```
double Production::n_years
```

The number of years being modelled.

4.19.4.15 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.19.4.16 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

4.19.4.17 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

4.19.4.18 normalized_production_series_given

```
bool Production::normalized_production_series_given
```

A boolean which indicates whether or not a normalized production time series is given.

4.19.4.19 normalized_production_vec

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

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

4.19.4.20 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.

4.19.4.21 operation_maintenance_cost_vec

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

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

4.19.4.22 path_2_normalized_production_time_series

```
std::string Production::path_2_normalized_production_time_series
```

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

4.19.4.23 print_flag

```
bool Production::print_flag
```

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

4.19.4.24 production_vec_kW

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

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

4.19.4.25 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.

4.19.4.26 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.19.4.27 running_hours

```
double Production::running_hours
```

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

4.19.4.28 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.

4.19.4.29 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

4.19.4.30 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](#)

4.20 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.
- std::string [path_2_normalized_production_time_series](#) = ""
A string defining the path (either relative or absolute) to the given normalized production time series.

4.20.1 Detailed Description

A structure which bundles the necessary inputs for the [Production](#) constructor. Provides default values for every necessary input.

4.20.2 Member Data Documentation

4.20.2.1 capacity_kW

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

The rated production capacity [kW] of the asset.

4.20.2.2 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).

4.20.2.3 nominal_discount_annual

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

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

4.20.2.4 nominal_inflation_annual

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

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

4.20.2.5 path_2_normalized_production_time_series

```
std::string ProductionInputs::path_2_normalized_production_time_series = ""
```

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

4.20.2.6 print_flag

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

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

4.20.2.7 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:

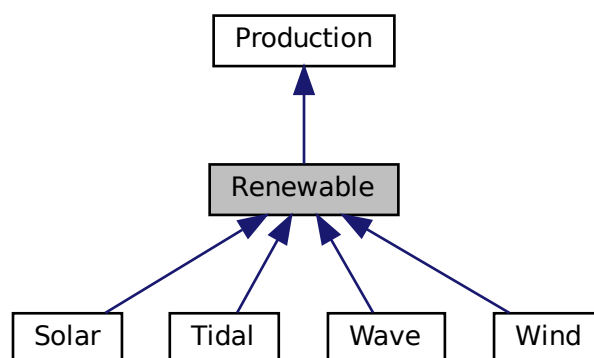
- header/Production/[Production.h](#)

4.21 Renewable Class Reference

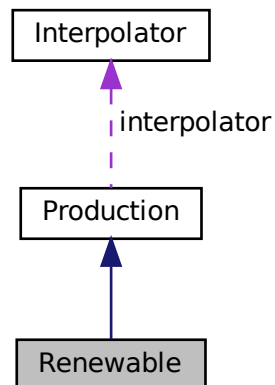
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

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

Public Attributes

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

Private Member Functions

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

4.21.1 Detailed Description

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

4.21.2 Constructor & Destructor Documentation

4.21.2.1 `Renewable()` [1/2]

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

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

```
125 {
126     //...
127
128     return;
129 } /* Renewable() */
```

4.21.2.2 `Renewable()` [2/2]

```
Renewable::Renewable (
    int n_points,
    double n_years,
    RenewableInputs renewable_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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

Parameters

<code>n_points</code>	The number of points in the modelling time series.
<code>n_years</code>	The number of years being modelled.
<code>renewable_inputs</code>	A structure of Renewable constructor inputs.
<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.

```

161 :
162 Production(
163     n_points,
164     n_years,
165     renewable_inputs.production_inputs,
166     time_vec_hrs_ptr
167 )
168 {
169     // 1. check inputs
170     this->__checkInputs(renewable_inputs);
171
172     // 2. set attributes
173     //...
174
175     // 3. construction print
176     if (this->print_flag) {
177         std::cout << "Renewable object constructed at " << this << std::endl;
178     }
179
180     return;
181 } /* Renewable() */

```

4.21.2.3 ~Renewable()

```

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

```

Destructor for the [Renewable](#) class.

```

384 {
385     // 1. destruction print
386     if (this->print_flag) {
387         std::cout << "Renewable object at " << this << " destroyed" << std::endl;
388     }
389
390     return;
391 } /* ~Renewable() */

```

4.21.3 Member Function Documentation

4.21.3.1 __checkInputs()

```

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

```

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

```

62 {
63     //...
64
65     return;
66 } /* __checkInputs() */

```

4.21.3.2 `__handleStartStop()`

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

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

```
89 {
90     if (this->is_running) {
91         // handle stopping
92         if (production_kW <= 0) {
93             this->is_running = false;
94         }
95     }
96
97     else {
98         // handle starting
99         if (production_kW > 0) {
100             this->is_running = true;
101             this->n_starts++;
102         }
103     }
104
105     return;
106 } /* __handleStartStop() */
```

4.21.3.3 `__writeSummary()`

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

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

```
97 {return;}
```

4.21.3.4 `__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](#).

```
104 {return;}
```

4.21.3.5 `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](#).

```

265 {
266     // 1. handle start/stop
267     this->__handleStartStop(timestep, dt_hrs, production_kW);
268
269     // 2. invoke base class method
270     load_kW = Production::commit(
271         timestep,
272         dt_hrs,
273         production_kW,
274         load_kW
275     );
276
277
278     //...
279
280     return load_kW;
281 } /* commit() */

```

4.21.3.6 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](#).

```

224 {
225     // 1. invoke base class method
226     Production::computeEconomics(time_vec_hrs_ptr);
227
228     return;
229 } /* computeEconomics() */

```

4.21.3.7 computeProductionkW() [1/2]

```

virtual double Renewable::computeProductionkW (
    int ,

```



```
double ,
double ) [inline], [virtual]
```

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

```
121 {return 0;}
```

4.21.3.8 computeProductionkW() [2/2]

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

Reimplemented in [Wave](#).

```
122 {return 0;}
```

4.21.3.9 handleReplacement()

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

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

Parameters

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

Reimplemented from [Production](#).

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

```
199 {
200     // 1. reset attributes
201     //...
202
203     // 2. invoke base class method
204     Production :: handleReplacement(timestep);
205
206     return;
207 } /* __handleReplacement() */
```

4.21.3.10 writeResults()

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

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

Parameters

<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>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.

```

325 {
326     // 1. handle sentinel
327     if (max_lines < 0) {
328         max_lines = this->n_points;
329     }
330
331     // 2. create subdirectories
332     write_path += "Production/";
333     if (not std::filesystem::is_directory(write_path)) {
334         std::filesystem::create_directory(write_path);
335     }
336
337     write_path += "Renewable/";
338     if (not std::filesystem::is_directory(write_path)) {
339         std::filesystem::create_directory(write_path);
340     }
341
342     write_path += this->type_str;
343     write_path += "_";
344     write_path += std::to_string(int(ceil(this->capacity_kw)));
345     write_path += "kW_idx";
346     write_path += std::to_string(renewable_index);
347     write_path += "/";
348     std::filesystem::create_directory(write_path);
349
350     // 3. write summary
351     this->__writeSummary(write_path);
352
353     // 4. write time series
354     if (max_lines > this->n_points) {
355         max_lines = this->n_points;
356     }
357
358     if (max_lines > 0) {
359         this->__writeTimeSeries(
360             write_path,
361             time_vec_hrs_ptr,
362             resource_map_1D_ptr,
363             resource_map_2D_ptr,
364             max_lines
365         );
366     }
367
368     return;
369 } /* writeResults() */

```

4.21.4 Member Data Documentation

4.21.4.1 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.

4.21.4.2 type

`RenewableType` `Renewable::type`

The type (`RenewableType`) of the asset.

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

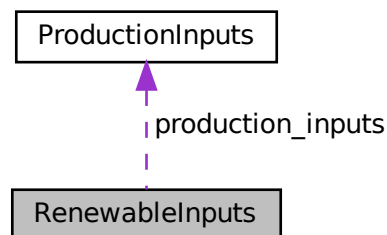
- `header/Production/Renewable/Renewable.h`
- `source/Production/Renewable/Renewable.cpp`

4.22 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.

4.22.1 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`.

4.22.2 Member Data Documentation

4.22.2.1 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](#)

4.23 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](#) ([NoncombustionType](#), 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 [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<double>> 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<vector<double>>> 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 [__checkResourceKey1D](#) (int, [NoncombustionType](#))
Helper method to check if given resource key (1D) is already in use.
- void [__checkTimePoint](#) (double, double, std::string, [ElectricalLoad](#) *)
Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.
- void [__throwLengthError](#) (std::string, [ElectricalLoad](#) *)
Helper method to throw data length error (if not the same as the given electrical load time series).
- void [__readHydroResource](#) (std::string, int, [ElectricalLoad](#) *)
Helper method to handle reading a hydro resource time series into [Resources](#).
- 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](#).

4.23.1 Detailed Description

A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

4.23.2 Constructor & Destructor Documentation

4.23.2.1 Resources()

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

Constructor for the [Resources](#) class.

```
755 {
756     return;
757 } /* Resources() */
```

4.23.2.2 ~Resources()

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

Destructor for the [Resources](#) class.

```
967 {
968     this->clear();
969     return;
970 } /* ~Resources() */
```

4.23.3 Member Function Documentation

4.23.3.1 `__checkResourceKey1D()` [1/2]

```
void Resources::__checkResourceKey1D (
    int resource_key,
    NoncombustionType noncombustion_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.
<i>noncombustion_type</i>	The type of renewable resource being added to Resources .

```
139 {
140     if (this->resource_map_1D.count(resource_key) > 0) {
141         std::string error_str = "ERROR: Resources::addResource(";
142
143         switch (noncombustion_type) {
144             case (NoncombustionType :: HYDRO): {
145                 error_str += "HYDRO): ";
146
147                 break;
148             }
149
150             default: {
151                 error_str += "UNDEFINED_TYPE): ";
152
153                 break;
154             }
155         }
156
157         error_str += "resource key (1D) ";
158         error_str += std::to_string(resource_key);
159         error_str += " is already in use";
160
161         #ifdef _WIN32
162             std::cout << error_str << std::endl;
163         #endif
164
165         throw std::invalid_argument(error_str);
166     }
167
168     return;
169 } /* __checkResourceKey1D() */
```

4.23.3.2 `__checkResourceKey1D()` [2/2]

```
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.
<i>renewable_type</i>	The type of renewable resource being added to Resources .

```

72 {
73     if (this->resource_map_1D.count(resource_key) > 0) {
74         std::string error_str = "ERROR:  Resources::addResource(";
75
76         switch (renewable_type) {
77             case (RenewableType :: SOLAR): {
78                 error_str += "SOLAR):  ";
79
80                 break;
81             }
82
83             case (RenewableType :: TIDAL): {
84                 error_str += "TIDAL):  ";
85
86                 break;
87             }
88
89             case (RenewableType :: WIND): {
90                 error_str += "WIND):  ";
91
92                 break;
93             }
94
95             default: {
96                 error_str += "UNDEFINED_TYPE):  ";
97
98                 break;
99             }
100         }
101
102         error_str += "resource key (1D) ";
103         error_str += std::to_string(resource_key);
104         error_str += " is already in use";
105
106         #ifdef _WIN32
107             std::cout << error_str << std::endl;
108         #endif
109
110         throw std::invalid_argument(error_str);
111     }
112
113     return;
114 } /* __checkResourceKey1D() */

```

4.23.3.3 __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.
---------------------	---

```

192 {
193     if (this->resource_map_2D.count(resource_key) > 0) {
194         std::string error_str = "ERROR:  Resources::addResource(";
195
196         switch (renewable_type) {
197             case (RenewableType :: WAVE): {
198                 error_str += "WAVE):  ";
199
200                 break;
201             }
202
203             default: {
204                 error_str += "UNDEFINED_TYPE):  ";
205
206                 break;
207             }
208         }
209

```

```

210     error_str += "resource key (2D) ";
211     error_str += std::to_string(resource_key);
212     error_str += " is already in use";
213
214     #ifdef _WIN32
215         std::cout << error_str << std::endl;
216     #endif
217
218     throw std::invalid_argument(error_str);
219 }
220
221 return;
222 } /* __checkResourceKey2D() */

```

4.23.3.4 __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. The given time series should align point-wise with the previously given electrical load time series.

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.

```

259 {
260     if (time_received_hrs != time_expected_hrs) {
261         std::string error_str = "ERROR: Resources::addResource(): ";
262         error_str += "the given resource time series at ";
263         error_str += path_2_resource_data;
264         error_str += " does not align with the ";
265         error_str += "previously given electrical load time series at ";
266         error_str += electrical_load_ptr->path_2_electrical_load_time_series;
267
268         #ifdef _WIN32
269             std::cout << error_str << std::endl;
270         #endif
271
272         throw std::runtime_error(error_str);
273     }
274
275     return;
276 } /* __checkTimePoint() */

```

4.23.3.5 __readHydroResource()

```

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

```

Helper method to handle reading a hydro 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.

```

348 {
349     // 1. init CSV reader, record path and type
350     io::CSVReader<2> CSV(path_2_resource_data);
351
352     CSV.read_header(
353         io::ignore_extra_column,
354         "Time (since start of data) [hrs]",
355         "Hydro Inflow [m3/hr]"
356     );
357
358     this->path_map_1D.insert(
359         std::pair<int, std::string>(resource_key, path_2_resource_data)
360     );
361
362     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "HYDRO"));
363
364     // 2. init map element
365     this->resource_map_1D.insert(
366         std::pair<int, std::vector<double>>(resource_key, {})
367     );
368     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
369
370
371     // 3. read in resource data, check against time series (point-wise and length)
372     int n_points = 0;
373     double time_hrs = 0;
374     double time_expected_hrs = 0;
375     double hydro_resource_m3hr = 0;
376
377     while (CSV.read_row(time_hrs, hydro_resource_m3hr)) {
378         if (n_points > electrical_load_ptr->n_points) {
379             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
380         }
381
382         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
383         this->__checkTimePoint(
384             time_hrs,
385             time_expected_hrs,
386             path_2_resource_data,
387             electrical_load_ptr
388         );
389
390         this->resource_map_1D[resource_key][n_points] = hydro_resource_m3hr;
391
392         n_points++;
393     }
394
395     // 4. check data length
396     if (n_points != electrical_load_ptr->n_points) {
397         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
398     }
399
400     return;
401 } /* __readHydroResource() */

```

4.23.3.6 __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.

```

431 {
432     // 1. init CSV reader, record path and type
433     io::CSVReader<2> CSV(path_2_resource_data);
434
435     CSV.read_header(
436         io::ignore_extra_column,
437         "Time (since start of data) [hrs]",
438         "Solar GHI [kW/m2]"
439     );
440
441     this->path_map_1D.insert(
442         std::pair<int, std::string>(resource_key, path_2_resource_data)
443     );
444
445     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));
446
447     // 2. init map element
448     this->resource_map_1D.insert(
449         std::pair<int, std::vector<double>>(resource_key, {})
450     );
451     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
452
453     // 3. read in resource data, check against time series (point-wise and length)
454     int n_points = 0;
455     double time_hrs = 0;
456     double time_expected_hrs = 0;
457     double solar_resource_kWm2 = 0;
458
459     while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
460         if (n_points > electrical_load_ptr->n_points) {
461             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
462         }
463
464         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
465         this->__checkTimePoint(
466             time_hrs,
467             time_expected_hrs,
468             path_2_resource_data,
469             electrical_load_ptr
470         );
471
472         this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
473         n_points++;
474     }
475
476     // 4. check data length
477     if (n_points != electrical_load_ptr->n_points) {
478         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
479     }
480
481     return;
482 }
483 /* __readSolarResource() */
484 }

```

4.23.3.7 __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.

```

514 {
515     // 1. init CSV reader, record path and type
516     io::CSVReader<2> CSV(path_2_resource_data);
517
518     CSV.read_header(
519         io::ignore_extra_column,
520         "Time (since start of data) [hrs]",
521         "Tidal Speed (hub depth) [m/s]"
522     );
523
524     this->path_map_1D.insert(
525         std::pair<int, std::string>(resource_key, path_2_resource_data)
526     );
527
528     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));
529
530     // 2. init map element
531     this->resource_map_1D.insert(
532         std::pair<int, std::vector<double>>(resource_key, {})
533     );
534     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
535
536     // 3. read in resource data, check against time series (point-wise and length)
537     int n_points = 0;
538     double time_hrs = 0;
539     double time_expected_hrs = 0;
540     double tidal_resource_ms = 0;
541
542     while (CSV.read_row(time_hrs, tidal_resource_ms)) {
543         if (n_points > electrical_load_ptr->n_points) {
544             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
545         }
546
547         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
548         this->__checkTimePoint(
549             time_hrs,
550             time_expected_hrs,
551             path_2_resource_data,
552             electrical_load_ptr
553         );
554
555         this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;
556         n_points++;
557     }
558
559     // 4. check data length
560     if (n_points != electrical_load_ptr->n_points) {
561         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
562     }
563
564     return;
565 }
566 /* __readTidalResource() */

```

4.23.3.8 __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

<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.

```

597 {
598     // 1. init CSV reader, record path and type
599     io::CSVReader<3> CSV(path_2_resource_data);
600
601     CSV.read_header(
602         io::ignore_extra_column,
603         "Time (since start of data) [hrs]",
604         "Significant Wave Height [m]",
605         "Energy Period [s]"
606     );
607
608     this->path_map_2D.insert(
609         std::pair<int, std::string>(resource_key, path_2_resource_data)
610     );
611
612     this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));
613
614     // 2. init map element
615     this->resource_map_2D.insert(
616         std::pair<int, std::vector<std::vector<double>>>(resource_key, {})
617     );
618     this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});
619
620
621     // 3. read in resource data, check against time series (point-wise and length)
622     int n_points = 0;
623     double time_hrs = 0;
624     double time_expected_hrs = 0;
625     double significant_wave_height_m = 0;
626     double energy_period_s = 0;
627
628     while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
629         if (n_points > electrical_load_ptr->n_points) {
630             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
631         }
632
633         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
634         this->__checkTimePoint(
635             time_hrs,
636             time_expected_hrs,
637             path_2_resource_data,
638             electrical_load_ptr
639         );
640
641         this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
642         this->resource_map_2D[resource_key][n_points][1] = energy_period_s;
643
644         n_points++;
645     }
646
647     // 4. check data length
648     if (n_points != electrical_load_ptr->n_points) {
649         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
650     }
651
652     return;
653 } /* __readWaveResource() */

```

4.23.3.9 __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.

```

683 {
684     // 1. init CSV reader, record path and type
685     io::CSVReader<2> CSV(path_2_resource_data);
686
687     CSV.read_header(
688         io::ignore_extra_column,
689         "Time (since start of data) [hrs]",
690         "Wind Speed (hub height) [m/s]"
691     );
692
693     this->path_map_1D.insert(
694         std::pair<int, std::string>(resource_key, path_2_resource_data)
695     );
696
697     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));
698
699     // 2. init map element
700     this->resource_map_1D.insert(
701         std::pair<int, std::vector<double>>(resource_key, {})
702     );
703     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
704
705
706     // 3. read in resource data, check against time series (point-wise and length)
707     int n_points = 0;
708     double time_hrs = 0;
709     double time_expected_hrs = 0;
710     double wind_resource_ms = 0;
711
712     while (CSV.read_row(time_hrs, wind_resource_ms)) {
713         if (n_points > electrical_load_ptr->n_points) {
714             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
715         }
716
717         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
718         this->__checkTimePoint(
719             time_hrs,
720             time_expected_hrs,
721             path_2_resource_data,
722             electrical_load_ptr
723         );
724
725         this->resource_map_1D[resource_key][n_points] = wind_resource_ms;
726
727         n_points++;
728     }
729
730     // 4. check data length
731     if (n_points != electrical_load_ptr->n_points) {
732         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
733     }
734
735     return;
736 } /* __readWindResource() */

```

4.23.3.10 __throwLengthError()

```

void Resources::__throwLengthError (
    std::string path_2_resource_data,
    ElectricalLoad * electrical_load_ptr ) [private]

```

Helper method to throw data length error (if not the same as the given electrical load time series).

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.

```

303 {
304     std::string error_str = "ERROR: Resources::addResource(): ";
305     error_str += "the given resource time series at ";
306     error_str += path_2_resource_data;
307     error_str += " is not the same length as the previously given electrical";
308     error_str += " load time series at ";
309     error_str += electrical_load_ptr->path_2_electrical_load_time_series;
310
311     #ifdef _WIN32
312         std::cout << error_str << std::endl;
313     #endif
314
315     throw std::runtime_error(error_str);
316
317     return;
318 } /* __throwLengthError() */

```

4.23.3.11 addResource() [1/2]

```

void Resources::addResource (
    NoncombustionType noncombustion_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>noncombustion_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.

```

794 {
795     switch (noncombustion_type) {
796         case (NoncombustionType :: HYDRO): {
797             this->__checkResourceKey1D(resource_key, noncombustion_type);
798
799             this->__readHydroResource (
800                 path_2_resource_data,
801                 resource_key,
802                 electrical_load_ptr
803             );
804
805             break;
806         }
807
808         default: {
809             std::string error_str = "ERROR: Resources :: addResource(): ";
810             error_str += "noncombustion type ";
811             error_str += std::to_string(noncombustion_type);
812             error_str += " has no associated resource";
813
814             #ifdef _WIN32
815                 std::cout << error_str << std::endl;
816             #endif
817
818             throw std::runtime_error(error_str);
819
820             break;
821         }
822     }
823
824     return;

```

```
825 } /* addResource() */
```

4.23.3.12 addResource() [2/2]

```
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.

```
862 {
863     switch (renewable_type) {
864         case (RenewableType :: SOLAR): {
865             this->__checkResourceKey1D(resource_key, renewable_type);
866
867             this->__readSolarResource(
868                 path_2_resource_data,
869                 resource_key,
870                 electrical_load_ptr
871             );
872
873             break;
874         }
875
876         case (RenewableType :: TIDAL): {
877             this->__checkResourceKey1D(resource_key, renewable_type);
878
879             this->__readTidalResource(
880                 path_2_resource_data,
881                 resource_key,
882                 electrical_load_ptr
883             );
884
885             break;
886         }
887
888         case (RenewableType :: WAVE): {
889             this->__checkResourceKey2D(resource_key, renewable_type);
890
891             this->__readWaveResource(
892                 path_2_resource_data,
893                 resource_key,
894                 electrical_load_ptr
895             );
896
897             break;
898         }
899
900         case (RenewableType :: WIND): {
901             this->__checkResourceKey1D(resource_key, renewable_type);
902
903             this->__readWindResource(
904                 path_2_resource_data,
905                 resource_key,
906                 electrical_load_ptr
907             );
```

```

908
909         break;
910     }
911
912     default: {
913         std::string error_str = "ERROR: Resources :: addResource(: ";
914         error_str += "renewable type ";
915         error_str += std::to_string(renewable_type);
916         error_str += " not recognized";
917
918         #ifdef _WIN32
919             std::cout << error_str << std::endl;
920         #endif
921
922         throw std::runtime_error(error_str);
923
924         break;
925     }
926 }
927
928 return;
929 } /* addResource() */

```

4.23.3.13 clear()

```

void Resources::clear (
    void )

```

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

```

943 {
944     this->resource_map_1D.clear();
945     this->string_map_1D.clear();
946     this->path_map_1D.clear();
947
948     this->resource_map_2D.clear();
949     this->string_map_2D.clear();
950     this->path_map_2D.clear();
951
952     return;
953 } /* clear() */

```

4.23.4 Member Data Documentation

4.23.4.1 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.

4.23.4.2 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.

4.23.4.3 resource_map_1D

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

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

4.23.4.4 resource_map_2D

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

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

4.23.4.5 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.

4.23.4.6 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:

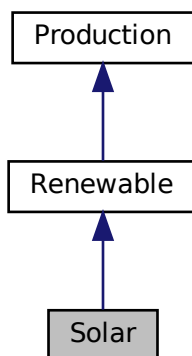
- header/[Resources.h](#)
- source/[Resources.cpp](#)

4.24 Solar Class Reference

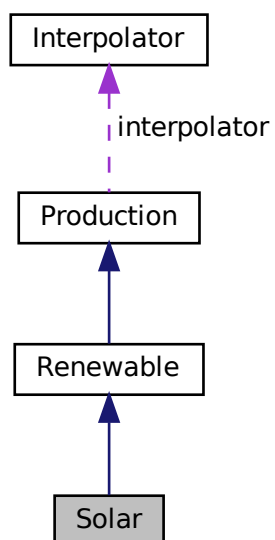
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

```
#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](#), std::vector< double > *)
Constructor (intended) for the [Solar](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Solar](#) (void)
Destructor for the [Solar](#) class.

Public Attributes

- double [derating](#)
The derating of the solar PV array (i.e., shadowing, soiling, etc.).

Private Member Functions

- void [__checkInputs](#) ([SolarInputs](#))
Helper method to check inputs to the [Solar](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic solar PV array capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Solar](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
Helper method to write time series results for [Solar](#).

4.24.1 Detailed Description

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

4.24.2 Constructor & Destructor Documentation

4.24.2.1 Solar() [1/2]

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

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

```
322 {
323     //...
324
325     return;
326 } /* Solar() */
```

4.24.2.2 Solar() [2/2]

```
Solar::Solar (
    int n_points,
    double n_years,
    SolarInputs solar_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```
358 :
359 Renewable(
360     n_points,
361     n_years,
362     solar_inputs.renewable_inputs,
363     time_vec_hrs_ptr
364 )
365 {
366     // 1. check inputs
367     this->__checkInputs(solar_inputs);
368
369     // 2. set attributes
370     this->type = RenewableType :: SOLAR;
371     this->type_str = "SOLAR";
372
373     this->resource_key = solar_inputs.resource_key;
374
375     this->derating = solar_inputs.derating;
376
377     if (solar_inputs.capital_cost < 0) {
378         this->capital_cost = this->__getGenericCapitalCost();
379     }
380     else {
381         this->capital_cost = solar_inputs.capital_cost;
382     }
383
384     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
385         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
386     }
387     else {
388         this->operation_maintenance_cost_kWh =
389             solar_inputs.operation_maintenance_cost_kWh;
390     }
391
392     if (not this->is_sunk) {
393         this->capital_cost_vec[0] = this->capital_cost;
394     }
395 }
```

```

396     // 3. construction print
397     if (this->print_flag) {
398         std::cout << "Solar object constructed at " << this << std::endl;
399     }
400
401     return;
402 } /* Renewable() */

```

4.24.2.3 ~Solar()

```

Solar::~~Solar (
    void )

```

Destructor for the [Solar](#) class.

```

548 {
549     // 1. destruction print
550     if (this->print_flag) {
551         std::cout << "Solar object at " << this << " destroyed" << std::endl;
552     }
553
554     return;
555 } /* ~Solar() */

```

4.24.3 Member Function Documentation

4.24.3.1 __checkInputs()

```

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

```

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

```

62 {
63     // 1. check derating
64     if (
65         solar_inputs.derating < 0 or
66         solar_inputs.derating > 1
67     ) {
68         std::string error_str = "ERROR: Solar(): ";
69         error_str += "SolarInputs::derating must be in the closed interval [0, 1]";
70
71         #ifdef _WIN32
72             std::cout << error_str << std::endl;
73         #endif
74
75         throw std::invalid_argument(error_str);
76     }
77
78     return;
79 } /* __checkInputs() */

```

4.24.3.2 `__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].

```
101 {
102     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;
103
104     return capital_cost_per_kW * this->capacity_kW;
105 } /* __getGenericCapitalCost() */
```

4.24.3.3 `__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].

```
128 {
129     return 0.01;
130 } /* __getGenericOpMaintCost() */
```

4.24.3.4 `__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](#).

```

148 {
149     // 1. create filestream
150     write_path += "summary_results.md";
151     std::ofstream ofs;
152     ofs.open(write_path, std::ofstream::out);
153
154     // 2. write summary results (markdown)
155     ofs << "# ";
156     ofs << std::to_string(int(ceil(this->capacity_kW)));
157     ofs << " kW SOLAR Summary Results\n";
158     ofs << "\n-----\n\n";
159
160     // 2.1. Production attributes
161     ofs << "## Production Attributes\n";
162     ofs << "\n";
163
164     ofs << "Capacity: " << this->capacity_kW << " kW \n";
165     ofs << "\n";
166
167     ofs << "Production Override: (N = 0 / Y = 1): "
168         << this->normalized_production_series_given << " \n";
169     if (this->normalized_production_series_given) {
170         ofs << "Path to Normalized Production Time Series: "
171             << this->path_2_normalized_production_time_series << " \n";
172     }
173     ofs << "\n";
174
175     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
176     ofs << "Capital Cost: " << this->capital_cost << " \n";
177     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
178         << " per kWh produced \n";
179     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
180         << " \n";
181     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
182         << " \n";
183     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
184     ofs << "\n";
185
186     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
187     ofs << "\n-----\n\n";
188
189     // 2.2. Renewable attributes
190     ofs << "## Renewable Attributes\n";
191     ofs << "\n";
192
193     ofs << "Resource Key (1D): " << this->resource_key << " \n";
194
195     ofs << "\n-----\n\n";
196
197     // 2.3. Solar attributes
198     ofs << "## Solar Attributes\n";
199     ofs << "\n";
200
201     ofs << "Derating Factor: " << this->derating << " \n";
202
203     ofs << "\n-----\n\n";
204
205     // 2.4. Solar Results
206     ofs << "## Results\n";
207     ofs << "\n";
208
209     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
210     ofs << "\n";
211
212     ofs << "Total Dispatch: " << this->total_dispatch_kWh
213         << " kWh \n";
214
215     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
216         << " per kWh dispatched \n";
217     ofs << "\n";
218
219     ofs << "Running Hours: " << this->running_hours << " \n";
220     ofs << "Replacements: " << this->n_replacements << " \n";
221
222     ofs << "\n-----\n\n";
223
224     ofs.close();
225     return;
226 } /* __writeSummary() */

```

4.24.3.5 `__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](#).

```
264 {
265     // 1. create filestream
266     write_path += "time_series_results.csv";
267     std::ofstream ofs;
268     ofs.open(write_path, std::ofstream::out);
269
270     // 2. write time series results (comma separated value)
271     ofs << "Time (since start of data) [hrs],";
272     ofs << "Solar Resource [kW/m2],";
273     ofs << "Production [kW],";
274     ofs << "Dispatch [kW],";
275     ofs << "Storage [kW],";
276     ofs << "Curtailement [kW],";
277     ofs << "Capital Cost (actual),";
278     ofs << "Operation and Maintenance Cost (actual),";
279     ofs << "\n";
280
281     for (int i = 0; i < max_lines; i++) {
282         ofs << time_vec_hrs_ptr->at(i) << ", ";
283
284         if (not this->normalized_production_series_given) {
285             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ", ";
286         }
287
288         else {
289             ofs << "OVERRIDE" << ", ";
290         }
291
292         ofs << this->production_vec_kW[i] << ", ";
293         ofs << this->dispatch_vec_kW[i] << ", ";
294         ofs << this->storage_vec_kW[i] << ", ";
295         ofs << this->curtailement_vec_kW[i] << ", ";
296         ofs << this->capital_cost_vec[i] << ", ";
297         ofs << this->operation_maintenance_cost_vec[i] << ", ";
298         ofs << "\n";
299     }
300
301     ofs.close();
302     return;
303 } /* __writeTimeSeries() */
```

4.24.3.6 `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](#).

```
520 {
521     // 1. invoke base class method
522     load_kW = Renewable::commit(
523         timestep,
524         dt_hrs,
525         production_kW,
526         load_kW
527     );
528
529
530     //...
531
532     return load_kW;
533 } /* commit() */
```

4.24.3.7 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](#).

```

462 {
463     // given production time series override
464     if (this->normalized_production_series_given) {
465         double production_kW = Production :: getProductionkW(timestep);
466
467         return production_kW;
468     }
469
470     // check if no resource
471     if (solar_resource_kWm2 <= 0) {
472         return 0;
473     }
474
475     // compute production
476     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
477
478     // cap production at capacity
479     if (production_kW > this->capacity_kW) {
480         production_kW = this->capacity_kW;
481     }
482
483     return production_kW;
484 } /* computeProductionkW() */

```

4.24.3.8 handleReplacement()

```

void Solar::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

420 {
421     // 1. reset attributes
422     //...
423
424     // 2. invoke base class method
425     Renewable :: handleReplacement(timestep);
426
427     return;
428 } /* __handleReplacement() */

```

4.24.4 Member Data Documentation

4.24.4.1 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:

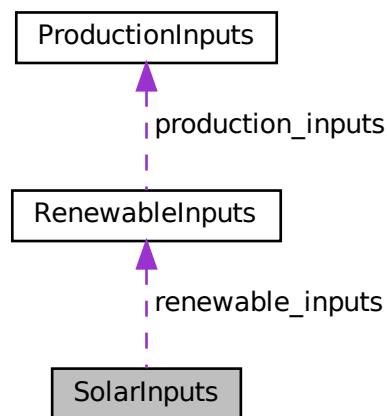
- header/Production/Renewable/[Solar.h](#)
- source/Production/Renewable/[Solar.cpp](#)

4.25 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.).

4.25.1 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](#).

4.25.2 Member Data Documentation

4.25.2.1 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].

4.25.2.2 derating

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

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

4.25.2.3 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].

4.25.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.25.2.5 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:

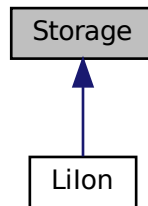
- [header/Production/Renewable/Solar.h](#)

4.26 Storage Class Reference

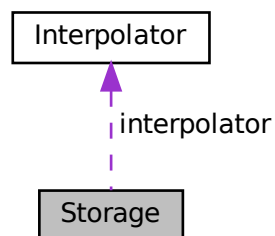
The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

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

Inheritance diagram for Storage:



Collaboration diagram for Storage:



Public Member Functions

- [Storage](#) (void)
Constructor (dummy) for the [Storage](#) class.
- [Storage](#) (int, double, [StorageInputs](#))
Constructor (intended) for the [Storage](#) class.
- virtual void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [getAvailablekW](#) (double)
- virtual double [getAcceptablekW](#) (double)
- virtual void [commitCharge](#) (int, double, double)

- virtual double `commitDischarge` (int, double, double, double)
- void `writeResults` (std::string, std::vector< double > *, int, int=-1)

Method which writes `Storage` results to an output directory.

- virtual `~Storage` (void)

Destructor for the `Storage` class.

Public Attributes

- `StorageType` type

The type (`StorageType`) of the asset.

- `Interpolator` interpolator

`Interpolator` component of `Storage`.

- bool `print_flag`

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

- bool `is_depleted`

A boolean which indicates whether or not the asset is currently considered depleted.

- bool `is_sunk`

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

- int `n_points`

The number of points in the modelling time series.

- int `n_replacements`

The number of times the asset has been replaced.

- double `n_years`

The number of years being modelled.

- double `power_capacity_kW`

The rated power capacity [kW] of the asset.

- double `energy_capacity_kWh`

The rated energy capacity [kWh] of the asset.

- double `charge_kWh`

The energy [kWh] stored in the asset.

- double `power_kW`

The power [kW] currently being charged/discharged by the asset.

- double `nominal_inflation_annual`

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

- double `nominal_discount_annual`

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

- double `real_discount_annual`

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

- double `capital_cost`

The capital cost of the asset (undefined currency).

- double `operation_maintenance_cost_kWh`

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

- double `net_present_cost`

The net present cost of this asset.

- double `total_discharge_kWh`

The total energy discharged [kWh] over the `Model` run.

- double `levellized_cost_of_energy_kWh`

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

- `std::string type_str`
A string describing the type of the asset.
- `std::vector< double > charge_vec_kWh`
A vector of the charge state [kWh] at each point in the modelling time series.
- `std::vector< double > charging_power_vec_kW`
A vector of the charging power [kW] at each point in the modelling time series.
- `std::vector< double > discharging_power_vec_kW`
A vector of the discharging power [kW] at each point in the modelling time series.
- `std::vector< double > capital_cost_vec`
A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- `std::vector< double > operation_maintenance_cost_vec`
A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- `void __checkInputs (int, double, StorageInputs)`
Helper method to check inputs to the [Storage](#) constructor.
- `double __computeRealDiscountAnnual (double, double)`
Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- `virtual void __writeSummary (std::string)`
- `virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)`

4.26.1 Detailed Description

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

4.26.2 Constructor & Destructor Documentation

4.26.2.1 [Storage\(\)](#) [1/2]

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

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

```
176 {
177     return;
178 } /* Storage() */
```

4.26.2.2 [Storage\(\)](#) [2/2]

```
Storage::Storage (
    int n_points,
    double n_years,
    StorageInputs storage_inputs )
```

Constructor (intended) for the [Storage](#) 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>storage_inputs</i>	A structure of Storage constructor inputs.

```

207 {
208     // 1. check inputs
209     this->__checkInputs(n_points, n_years, storage_inputs);
210
211     // 2. set attributes
212     this->print_flag = storage_inputs.print_flag;
213     this->is_depleted = false;
214     this->is_sunk = storage_inputs.is_sunk;
215
216     this->n_points = n_points;
217     this->n_replacements = 0;
218
219     this->n_years = n_years;
220
221     this->power_capacity_kW = storage_inputs.power_capacity_kW;
222     this->energy_capacity_kWh = storage_inputs.energy_capacity_kWh;
223
224     this->charge_kWh = 0;
225     this->power_kW = 0;
226
227     this->nominal_inflation_annual = storage_inputs.nominal_inflation_annual;
228     this->nominal_discount_annual = storage_inputs.nominal_discount_annual;
229
230     this->real_discount_annual = this->__computeRealDiscountAnnual(
231         storage_inputs.nominal_inflation_annual,
232         storage_inputs.nominal_discount_annual
233     );
234
235     this->capital_cost = 0;
236     this->operation_maintenance_cost_kWh = 0;
237     this->net_present_cost = 0;
238     this->total_discharge_kWh = 0;
239     this->levellized_cost_of_energy_kWh = 0;
240
241     this->charge_vec_kWh.resize(this->n_points, 0);
242     this->charging_power_vec_kW.resize(this->n_points, 0);
243     this->discharging_power_vec_kW.resize(this->n_points, 0);
244
245     this->capital_cost_vec.resize(this->n_points, 0);
246     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
247
248     // 3. construction print
249     if (this->print_flag) {
250         std::cout << "Storage object constructed at " << this << std::endl;
251     }
252
253     return;
254 } /* Storage() */

```

4.26.2.3 ~Storage()

```

Storage::~Storage (
    void ) [virtual]

```

Destructor for the [Storage](#) class.

```

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

```

4.26.3 Member Function Documentation

4.26.3.1 `__checkInputs()`

```
void Storage::__checkInputs (
    int n_points,
    double n_years,
    StorageInputs storage_inputs ) [private]
```

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

Parameters

<code>n_points</code>	The number of points in the modelling time series.
<code>storage_inputs</code>	A structure of Storage constructor inputs.

```
70 {
71     // 1. check n_points
72     if (n_points <= 0) {
73         std::string error_str = "ERROR: Storage(): n_points must be > 0";
74
75         #ifdef _WIN32
76             std::cout << error_str << std::endl;
77         #endif
78
79         throw std::invalid_argument(error_str);
80     }
81
82     // 2. check n_years
83     if (n_years <= 0) {
84         std::string error_str = "ERROR: Storage(): n_years must be > 0";
85
86         #ifdef _WIN32
87             std::cout << error_str << std::endl;
88         #endif
89
90         throw std::invalid_argument(error_str);
91     }
92
93     // 3. check power_capacity_kW
94     if (storage_inputs.power_capacity_kW <= 0) {
95         std::string error_str = "ERROR: Storage(): ";
96         error_str += "StorageInputs::power_capacity_kW must be > 0";
97
98         #ifdef _WIN32
99             std::cout << error_str << std::endl;
100         #endif
101
102         throw std::invalid_argument(error_str);
103     }
104
105     // 4. check energy_capacity_kWh
106     if (storage_inputs.energy_capacity_kWh <= 0) {
107         std::string error_str = "ERROR: Storage(): ";
108         error_str += "StorageInputs::energy_capacity_kWh must be > 0";
109
110         #ifdef _WIN32
111             std::cout << error_str << std::endl;
112         #endif
113
114         throw std::invalid_argument(error_str);
115     }
116
117     return;
118 } /* __checkInputs() */
```

4.26.3.2 `__computeRealDiscountAnnual()`

```
double Storage::__computeRealDiscountAnnual (
    double nominal_inflation_annual,
    double nominal_discount_annual ) [private]
```

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

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

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

Parameters

<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.

```

152 {
153     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
154     real_discount_annual /= 1 + nominal_inflation_annual;
155
156     return real_discount_annual;
157 } /* __computeRealDiscountAnnual() */

```

4.26.3.3 __writeSummary()

```

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

```

Reimplemented in [Lilon](#).

```

104 {return;}

```

4.26.3.4 __writeTimeSeries()

```

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

```

Reimplemented in [Lilon](#).

```

105 {return;}

```

4.26.3.5 commitCharge()

```

virtual void Storage::commitCharge (
    int ,
    double ,
    double ) [inline], [virtual]

```

Reimplemented in [Lilon](#).

```

159 {return;}

```

4.26.3.6 commitDischarge()

```
virtual double Storage::commitDischarge (
    int ,
    double ,
    double ,
    double ) [inline], [virtual]
```

Reimplemented in [Lilon](#).

```
160 {return 0;}
```

4.26.3.7 computeEconomics()

```
void Storage::computeEconomics (
    std::vector< double > * time_vec_hrs_ptr )
```

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

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

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

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

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

Parameters

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

1. compute levellized cost of energy (per unit discharged)

```
307 {
308     // 1. compute net present cost
309     double t_hrs = 0;
310     double real_discount_scalar = 0;
311
312     for (int i = 0; i < this->n_points; i++) {
313         t_hrs = time_vec_hrs_ptr->at(i);
314
315         real_discount_scalar = 1.0 / pow(
316             1 + this->real_discount_annual,
317             t_hrs / 8760
318         );
319
320         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
321
322         this->net_present_cost +=
323             real_discount_scalar * this->operation_maintenance_cost_vec[i];
324     }
325
326     // assuming 8,760 hours per year
327     if (this->total_discharge_kWh <= 0) {
328         this->levellized_cost_of_energy_kWh = this->net_present_cost;
329     }
330
331     else {
332         double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
333
334         double capital_recovery_factor =
335             (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
336             (pow(1 + this->real_discount_annual, n_years) - 1);
337
338         double total_annualized_cost = capital_recovery_factor *
339             this->net_present_cost;
340     }
```

```

341
342     this->levelled_cost_of_energy_kWh =
343         (n_years * total_annualized_cost) /
344         this->total_discharge_kWh;
345     }
346
347     return;
348 } /* computeEconomics() */

```

4.26.3.8 getAcceptablekW()

```

virtual double Storage::getAcceptablekW (
    double ) [inline], [virtual]

```

Reimplemented in [Lilon](#).

```

157 {return 0;}

```

4.26.3.9 getAvailablekW()

```

virtual double Storage::getAvailablekW (
    double ) [inline], [virtual]

```

Reimplemented in [Lilon](#).

```

156 {return 0;}

```

4.26.3.10 handleReplacement()

```

void Storage::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented in [Lilon](#).

```

272 {
273     // 1. reset attributes
274     this->charge_kWh = 0;
275     this->power_kW = 0;
276
277     // 2. log replacement
278     this->n_replacements++;
279
280     // 3. incur capital cost in timestep
281     this->capital_cost_vec[timestep] = this->capital_cost;
282
283     return;
284 } /* __handleReplacement() */

```

4.26.3.11 writeResults()

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

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

Parameters

<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>storage_index</i>	An integer which corresponds to the index of the Storage 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.

```
385 {
386     // 1. handle sentinel
387     if (max_lines < 0) {
388         max_lines = this->n_points;
389     }
390
391     // 2. create subdirectories
392     write_path += "Storage/";
393     if (not std::filesystem::is_directory(write_path)) {
394         std::filesystem::create_directory(write_path);
395     }
396
397     write_path += this->type_str;
398     write_path += "_";
399     write_path += std::to_string(int(ceil(this->power_capacity_kW)));
400     write_path += "kW_";
401     write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
402     write_path += "kWh_idx";
403     write_path += std::to_string(storage_index);
404     write_path += "/";
405     std::filesystem::create_directory(write_path);
406
407     // 3. write summary
408     this->__writeSummary(write_path);
409
410     // 4. write time series
411     if (max_lines > this->n_points) {
412         max_lines = this->n_points;
413     }
414
415     if (max_lines > 0) {
416         this->__writeTimeSeries(
417             write_path,
418             time_vec_hrs_ptr,
419             max_lines
420         );
421     }
422
423     return;
424 } /* writeResults() */
```

4.26.4 Member Data Documentation

4.26.4.1 capital_cost

```
double Storage::capital_cost
```

The capital cost of the asset (undefined currency).

4.26.4.2 capital_cost_vec

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

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

4.26.4.3 charge_kWh

```
double Storage::charge_kWh
```

The energy [kWh] stored in the asset.

4.26.4.4 charge_vec_kWh

```
std::vector<double> Storage::charge_vec_kWh
```

A vector of the charge state [kWh] at each point in the modelling time series.

4.26.4.5 charging_power_vec_kW

```
std::vector<double> Storage::charging_power_vec_kW
```

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

4.26.4.6 discharging_power_vec_kW

```
std::vector<double> Storage::discharging_power_vec_kW
```

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

4.26.4.7 energy_capacity_kWh

```
double Storage::energy_capacity_kWh
```

The rated energy capacity [kWh] of the asset.

4.26.4.8 interpolator

```
Interpolator Storage::interpolator
```

[Interpolator](#) component of [Storage](#).

4.26.4.9 is_depleted

```
bool Storage::is_depleted
```

A boolean which indicates whether or not the asset is currently considered depleted.

4.26.4.10 is_sunk

```
bool Storage::is_sunk
```

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

4.26.4.11 levellized_cost_of_energy_kWh

```
double Storage::levellized_cost_of_energy_kWh
```

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

4.26.4.12 n_points

```
int Storage::n_points
```

The number of points in the modelling time series.

4.26.4.13 n_replacements

```
int Storage::n_replacements
```

The number of times the asset has been replaced.

4.26.4.14 n_years

```
double Storage::n_years
```

The number of years being modelled.

4.26.4.15 net_present_cost

```
double Storage::net_present_cost
```

The net present cost of this asset.

4.26.4.16 nominal_discount_annual

```
double Storage::nominal_discount_annual
```

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

4.26.4.17 nominal_inflation_annual

```
double Storage::nominal_inflation_annual
```

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

4.26.4.18 operation_maintenance_cost_kWh

```
double Storage::operation_maintenance_cost_kWh
```

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

4.26.4.19 operation_maintenance_cost_vec

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

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

4.26.4.20 power_capacity_kW

```
double Storage::power_capacity_kW
```

The rated power capacity [kW] of the asset.

4.26.4.21 power_kW

```
double Storage::power_kW
```

The power [kW] currently being charged/discharged by the asset.

4.26.4.22 print_flag

```
bool Storage::print_flag
```

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

4.26.4.23 real_discount_annual

```
double Storage::real_discount_annual
```

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

4.26.4.24 total_discharge_kWh

```
double Storage::total_discharge_kWh
```

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

4.26.4.25 type

`StorageType` `Storage::type`

The type (`StorageType`) of the asset.

4.26.4.26 type_str

`std::string` `Storage::type_str`

A string describing the type of the asset.

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

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

4.27 StorageInputs Struct Reference

A structure which bundles the necessary inputs for the [Storage](#) constructor. Provides default values for every necessary input.

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

Public Attributes

- bool [print_flag](#) = false
A flag which indicates whether or not object construct/destruction should be verbose.
- bool [is_sunk](#) = false
A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double [power_capacity_kW](#) = 100
The rated power capacity [kW] of the asset.
- double [energy_capacity_kWh](#) = 1000
The rated energy capacity [kWh] of the asset.
- double [nominal_inflation_annual](#) = 0.02
The nominal, annual inflation rate to use in computing model economics.
- double [nominal_discount_annual](#) = 0.04
The nominal, annual discount rate to use in computing model economics.

4.27.1 Detailed Description

A structure which bundles the necessary inputs for the [Storage](#) constructor. Provides default values for every necessary input.

4.27.2 Member Data Documentation

4.27.2.1 energy_capacity_kWh

```
double StorageInputs::energy_capacity_kWh = 1000
```

The rated energy capacity [kWh] of the asset.

4.27.2.2 is_sunk

```
bool StorageInputs::is_sunk = false
```

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

4.27.2.3 nominal_discount_annual

```
double StorageInputs::nominal_discount_annual = 0.04
```

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

4.27.2.4 nominal_inflation_annual

```
double StorageInputs::nominal_inflation_annual = 0.02
```

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

4.27.2.5 power_capacity_kW

```
double StorageInputs::power_capacity_kW = 100
```

The rated power capacity [kW] of the asset.

4.27.2.6 print_flag

```
bool StorageInputs::print_flag = false
```

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

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

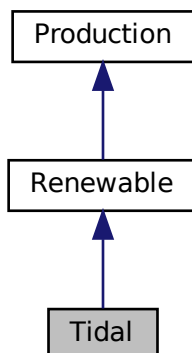
- header/Storage/[Storage.h](#)

4.28 Tidal Class Reference

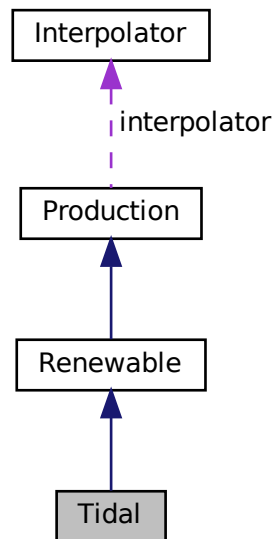
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

```
#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](#), std::vector< double > *)
Constructor (intended) for the [Tidal](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Tidal](#) (void)
Destructor for the [Tidal](#) class.

Public Attributes

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

Private Member Functions

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

4.28.1 Detailed Description

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

4.28.2 Constructor & Destructor Documentation

4.28.2.1 Tidal() [1/2]

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

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

```
481 {
482     return;
483 } /* Tidal() */
```

4.28.2.2 Tidal() [2/2]

```
Tidal::Tidal (
    int n_points,
    double n_years,
    TidalInputs tidal_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```

515 :
516 Renewable(
517     n_points,
518     n_years,
519     tidal_inputs.renewable_inputs,
520     time_vec_hrs_ptr
521 )
522 {
523     // 1. check inputs
524     this->__checkInputs(tidal_inputs);
525
526     // 2. set attributes
527     this->type = RenewableType :: TIDAL;
528     this->type_str = "TIDAL";
529
530     this->resource_key = tidal_inputs.resource_key;
531
532     this->design_speed_ms = tidal_inputs.design_speed_ms;
533
534     this->power_model = tidal_inputs.power_model;
535
536     switch (this->power_model) {
537         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
538             this->power_model_string = "CUBIC";
539
540             break;
541         }
542
543         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
544             this->power_model_string = "EXPONENTIAL";
545
546             break;
547         }
548
549         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
550             this->power_model_string = "LOOKUP";
551
552             break;
553         }
554
555         default: {
556             std::string error_str = "ERROR: Tidal(): ";
557             error_str += "power production model ";
558             error_str += std::to_string(this->power_model);
559             error_str += " not recognized";
560
561             #ifdef _WIN32
562                 std::cout << error_str << std::endl;
563             #endif
564
565             throw std::runtime_error(error_str);
566
567             break;
568         }
569     }
570
571     if (tidal_inputs.capital_cost < 0) {
572         this->capital_cost = this->__getGenericCapitalCost();
573     }
574     else {
575         this->capital_cost = tidal_inputs.capital_cost;
576     }
577
578     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
579         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
580     }
581     else {
582         this->operation_maintenance_cost_kWh =
583             tidal_inputs.operation_maintenance_cost_kWh;
584     }
585
586     if (not this->is_sunk) {
587         this->capital_cost_vec[0] = this->capital_cost;
588     }
589
590     // 3. construction print

```

```

591     if (this->print_flag) {
592         std::cout << "Tidal object constructed at " << this << std::endl;
593     }
594
595     return;
596 } /* Renewable() */

```

4.28.2.3 ~Tidal()

```

Tidal::~~Tidal (
    void )

```

Destructor for the [Tidal](#) class.

```

783 {
784     // 1. destruction print
785     if (this->print_flag) {
786         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
787     }
788
789     return;
790 } /* ~Tidal() */

```

4.28.3 Member Function Documentation

4.28.3.1 __checkInputs()

```

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

```

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

Ref: [Bir et al. \[2011\]](#)

Ref: [Lewis et al. \[2021\]](#)

```

65 {
66     // 1. check design_speed_ms
67     if (tidal_inputs.design_speed_ms <= 0) {
68         std::string error_str = "ERROR: Tidal(): ";
69         error_str += "TidalInputs::design_speed_ms must be > 0";
70
71         #ifdef _WIN32
72             std::cout << error_str << std::endl;
73         #endif
74
75         throw std::invalid_argument(error_str);
76     }
77
78     else if (tidal_inputs.design_speed_ms < 2) {
79         std::string warning_str = "WARNING: Tidal(): ";
80         warning_str += "Setting TidalInputs::design_speed_ms to less than 2 m/s may be ";
81         warning_str += "technically unrealistic";
82
83         std::cout << warning_str << std::endl;
84     }
85
86     return;
87 } /* __checkInputs() */

```


4.28.3.2 `__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. \[2023\]](#)

Ref: [Bir et al. \[2011\]](#)

Ref: [Lewis et al. \[2021\]](#)

Ref: [Whitby and Ugalde-Loo \[2013\]](#)

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.

```
177 {
178     double production = 0;
179
180     if (
181         tidal_resource_ms < 0.15 * this->design_speed_ms or
182         tidal_resource_ms > 1.25 * this->design_speed_ms
183     ){
184         production = 0;
185     }
186
187     else if (
188         0.15 * this->design_speed_ms <= tidal_resource_ms and
189         tidal_resource_ms <= this->design_speed_ms
190     ) {
191         production = (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
192     }
193
194     else {
195         production = 1;
196     }
197
198     return production * this->capacity_kW;
199 } /* __computeCubicProductionkW() */
```

4.28.3.3 `__computeExponentialProductionkW()`

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

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

Ref: [Truelove et al. \[2019\]](#)

Parameters

<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.

```

233 {
234     double production = 0;
235
236     double turbine_speed =
237         (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
238
239     if (turbine_speed < -0.71 or turbine_speed > 0.65) {
240         production = 0;
241     }
242
243     else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
244         production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
245     }
246
247     else {
248         production = 1;
249     }
250
251     return production * this->capacity_kW;
252 } /* __computeExponentialProductionkW() */

```

4.28.3.4 __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.

```

284 {
285     // *** WORK IN PROGRESS *** //
286
287     return 0;
288 } /* __computeLookupProductionkW() */

```

4.28.3.5 `__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].

```
109 {
110     double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;
111
112     return capital_cost_per_kW * this->capacity_kW;
113 } /* __getGenericCapitalCost() */
```

4.28.3.6 `__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].

```
136 {
137     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
138
139     return operation_maintenance_cost_kWh;
140 } /* __getGenericOpMaintCost() */
```

4.28.3.7 `__writeSummary()`

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

Helper method to write summary 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.
-------------------	--

Reimplemented from [Renewable](#).

```

306 {
307     // 1. create filestream
308     write_path += "summary_results.md";
309     std::ofstream ofs;
310     ofs.open(write_path, std::ofstream::out);
311
312     // 2. write summary results (markdown)
313     ofs << "# ";
314     ofs << std::to_string(int(ceil(this->capacity_kW)));
315     ofs << " kW TIDAL Summary Results\n";
316     ofs << "\n-----\n\n";
317
318     // 2.1. Production attributes
319     ofs << "## Production Attributes\n";
320     ofs << "\n";
321
322     ofs << "Capacity: " << this->capacity_kW << " kW \n";
323     ofs << "\n";
324
325     ofs << "Production Override: (N = 0 / Y = 1): "
326     << this->normalized_production_series_given << " \n";
327     if (this->normalized_production_series_given) {
328         ofs << "Path to Normalized Production Time Series: "
329         << this->path_2_normalized_production_time_series << " \n";
330     }
331     ofs << "\n";
332
333     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
334     ofs << "Capital Cost: " << this->capital_cost << " \n";
335     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
336     << " per kWh produced \n";
337     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
338     << " \n";
339     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
340     << " \n";
341     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
342     ofs << "\n";
343
344     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
345     ofs << "\n-----\n\n";
346
347     // 2.2. Renewable attributes
348     ofs << "## Renewable Attributes\n";
349     ofs << "\n";
350
351     ofs << "Resource Key (1D): " << this->resource_key << " \n";
352
353     ofs << "\n-----\n\n";
354
355     // 2.3. Tidal attributes
356     ofs << "## Tidal Attributes\n";
357     ofs << "\n";
358
359     ofs << "Power Production Model: " << this->power_model_string << " \n";
360     ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
361
362     ofs << "\n-----\n\n";
363
364     // 2.4. Tidal Results
365     ofs << "## Results\n";
366     ofs << "\n";
367
368     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
369     ofs << "\n";
370
371     ofs << "Total Dispatch: " << this->total_dispatch_kWh
372     << " kWh \n";
373
374     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
375     << " per kWh dispatched \n";
376     ofs << "\n";
377
378     ofs << "Running Hours: " << this->running_hours << " \n";
379     ofs << "Replacements: " << this->n_replacements << " \n";
380
381     ofs << "\n-----\n\n";
382

```

```

383     ofs.close();
384
385     return;
386 } /* __writeSummary() */

```

4.28.3.8 __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](#).

```

424 {
425     // 1. create filestream
426     write_path += "time_series_results.csv";
427     std::ofstream ofs;
428     ofs.open(write_path, std::ofstream::out);
429
430     // 2. write time series results (comma separated value)
431     ofs << "Time (since start of data) [hrs],";
432     ofs << "Tidal Resource [m/s],";
433     ofs << "Production [kW],";
434     ofs << "Dispatch [kW],";
435     ofs << "Storage [kW],";
436     ofs << "Curtailment [kW],";
437     ofs << "Capital Cost (actual),";
438     ofs << "Operation and Maintenance Cost (actual),";
439     ofs << "\n";
440
441     for (int i = 0; i < max_lines; i++) {
442         ofs << time_vec_hrs_ptr->at(i) << ",";
443
444         if (not this->normalized_production_series_given) {
445             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ",";
446         }
447
448         else {
449             ofs << "OVERRIDE" << ",";
450         }
451
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() */

```

4.28.3.9 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](#).

```
755 {
756     // 1. invoke base class method
757     load_kW = Renewable::commit(
758         timestep,
759         dt_hrs,
760         production_kW,
761         load_kW
762     );
763
764
765     //...
766
767     return load_kW;
768 } /* commit() */
```

4.28.3.10 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](#).

```

654 {
655     // given production time series override
656     if (this->normalized_production_series_given) {
657         double production_kW = Production :: getProductionkW(timestep);
658
659         return production_kW;
660     }
661
662     // check if no resource
663     if (tidal_resource_ms <= 0) {
664         return 0;
665     }
666
667     // compute production
668     double production_kW = 0;
669
670     switch (this->power_model) {
671         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
672             production_kW = this->__computeCubicProductionkW(
673                 timestep,
674                 dt_hrs,
675                 tidal_resource_ms
676             );
677
678             break;
679         }
680
681         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
682             production_kW = this->__computeExponentialProductionkW(
683                 timestep,
684                 dt_hrs,
685                 tidal_resource_ms
686             );
687
688             break;
689         }
690
691         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
692             production_kW = this->__computeLookupProductionkW(
693                 timestep,
694                 dt_hrs,
695                 tidal_resource_ms
696             );
697
698             break;
699         }
700     }
701
702     default: {
703         std::string error_str = "ERROR: Tidal::computeProductionkW(): ";
704         error_str += "power model ";
705         error_str += std::to_string(this->power_model);
706         error_str += " not recognized";
707
708         #ifdef _WIN32
709             std::cout << error_str << std::endl;
710         #endif
711
712         throw std::runtime_error(error_str);
713
714         break;
715     }
716 }
717
718 return production_kW;
719 } /* computeProductionkW() */

```

4.28.3.11 handleReplacement()

```

void Tidal::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

614 {
615     // 1. reset attributes
616     //...
617
618     // 2. invoke base class method
619     Renewable :: handleReplacement(timestep);
620
621     return;
622 } /* __handleReplacement() */

```

4.28.4 Member Data Documentation

4.28.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

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

4.28.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

4.28.4.3 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:

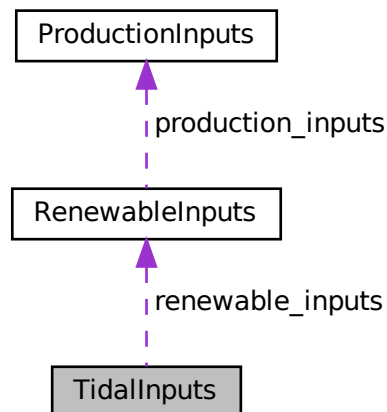
- header/Production/Renewable/[Tidal.h](#)
- source/Production/Renewable/[Tidal.cpp](#)

4.29 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.

4.29.1 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](#).

4.29.2 Member Data Documentation

4.29.2.1 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].

4.29.2.2 design_speed_ms

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

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

4.29.2.3 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].

4.29.2.4 power_model

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

The tidal power production model to be applied.

4.29.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.29.2.6 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:

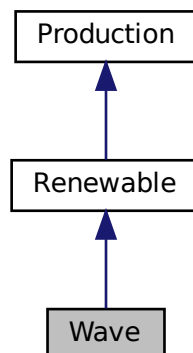
- [header/Production/Renewable/Tidal.h](#)

4.30 Wave Class Reference

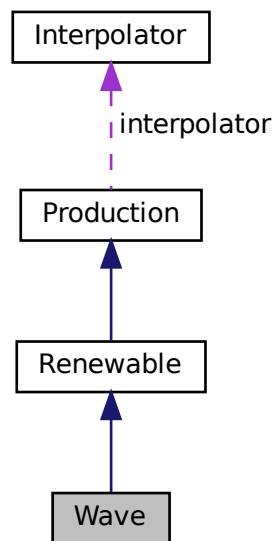
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

```
#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](#), std::vector< double > *)
Constructor (intended) for the [Wave](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double, double)
Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Wave](#) (void)
Destructor for the [Wave](#) class.

Public Attributes

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

Private Member Functions

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

4.30.1 Detailed Description

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

4.30.2 Constructor & Destructor Documentation

4.30.2.1 `Wave()` [1/2]

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

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

```
543 {
544     return;
545 } /* Wave() */
```

4.30.2.2 `Wave()` [2/2]

```
Wave::Wave (
    int n_points,
    double n_years,
    WaveInputs wave_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```

577 :
578 Renewable(
579     n_points,
580     n_years,
581     wave_inputs.renewable_inputs,
582     time_vec_hrs_ptr
583 )
584 {
585     // 1. check inputs
586     this->__checkInputs(wave_inputs);
587
588     // 2. set attributes
589     this->type = RenewableType :: WAVE;
590     this->type_str = "WAVE";
591
592     this->resource_key = wave_inputs.resource_key;
593
594     this->design_significant_wave_height_m =
595         wave_inputs.design_significant_wave_height_m;
596     this->design_energy_period_s = wave_inputs.design_energy_period_s;
597
598     this->power_model = wave_inputs.power_model;
599
600     switch (this->power_model) {
601         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
602             this->power_model_string = "GAUSSIAN";
603
604             break;
605         }
606
607         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
608             this->power_model_string = "PARABOLOID";
609
610             break;
611         }
612
613         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
614             this->power_model_string = "LOOKUP";
615
616             this->interpolator.addData2D(
617                 0,
618                 wave_inputs.path_2_normalized_performance_matrix
619             );
620
621             break;
622         }
623
624         default: {
625             std::string error_str = "ERROR: Wave(): ";
626             error_str += "power production model ";
627             error_str += std::to_string(this->power_model);
628             error_str += " not recognized";
629
630             #ifdef _WIN32
631                 std::cout << error_str << std::endl;
632             #endif
633
634             throw std::runtime_error(error_str);
635
636             break;
637         }
638     }
639
640     if (wave_inputs.capital_cost < 0) {
641         this->capital_cost = this->__getGenericCapitalCost();
642     }
643     else {
644         this->capital_cost = wave_inputs.capital_cost;
645     }
646
647     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
648         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
649     }
650     else {
651         this->operation_maintenance_cost_kWh =
652             wave_inputs.operation_maintenance_cost_kWh;

```

```

653     }
654
655     if (not this->is_sunk) {
656         this->capital_cost_vec[0] = this->capital_cost;
657     }
658
659     // 3. construction print
660     if (this->print_flag) {
661         std::cout << "Wave object constructed at " << this << std::endl;
662     }
663
664     return;
665 } /* Renewable() */

```

4.30.2.3 ~Wave()

```

Wave::~~Wave (
    void )

```

Destructor for the [Wave](#) class.

```

858 {
859     // 1. destruction print
860     if (this->print_flag) {
861         std::cout << "Wave object at " << this << " destroyed" << std::endl;
862     }
863
864     return;
865 } /* ~Wave() */

```

4.30.3 Member Function Documentation

4.30.3.1 __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.
--------------------	---

```

64 {
65     // 1. check design_significant_wave_height_m
66     if (wave_inputs.design_significant_wave_height_m <= 0) {
67         std::string error_str = "ERROR: Wave(): ";
68         error_str += "WaveInputs::design_significant_wave_height_m 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     // 2. check design_energy_period_s
78     if (wave_inputs.design_energy_period_s <= 0) {
79         std::string error_str = "ERROR: Wave(): ";
80         error_str += "WaveInputs::design_energy_period_s must be > 0";
81
82         #ifdef _WIN32

```

```

83         std::cout << error_str << std::endl;
84     #endif
85
86     throw std::invalid_argument(error_str);
87 }
88
89 // 3. if WAVE_POWER_LOOKUP, check that path is given
90 if (
91     wave_inputs.power_model == WavePowerProductionModel::WAVE_POWER_LOOKUP and
92     wave_inputs.path_2_normalized_performance_matrix.empty()
93 ) {
94     std::string error_str = "ERROR: Wave() power model was set to ";
95     error_str += "WavePowerProductionModel::WAVE_POWER_LOOKUP, but no path to a ";
96     error_str += "normalized performance matrix was given";
97
98     #ifdef _WIN32
99         std::cout << error_str << std::endl;
100    #endif
101
102    throw std::invalid_argument(error_str);
103 }
104
105 return;
106 } /* __checkInputs() */

```

4.30.3.2 __computeGaussianProductionkW()

```

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

```

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

Ref: [Truelove et al. \[2019\]](#)

Parameters

<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.

```

201 {
202     double H_s_nondim =
203         (significant_wave_height_m - this->design_significant_wave_height_m) /
204         this->design_significant_wave_height_m;
205
206     double T_e_nondim =
207         (energy_period_s - this->design_energy_period_s) /
208         this->design_energy_period_s;
209
210     double production = exp(
211         -2.25119 * pow(T_e_nondim, 2) +
212         3.44570 * T_e_nondim * H_s_nondim -
213         4.01508 * pow(H_s_nondim, 2)
214     );
215
216     return production * this->capacity_kW;

```



```
217 } /* __computeGaussianProductionkW() */
```

4.30.3.3 __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.

```
318 {
319     double prod = this->interpolator.interp2D(
320         0,
321         significant_wave_height_m,
322         energy_period_s
323     );
324
325     return prod * this->capacity_kW;
326 } /* __computeLookupProductionkW() */
```

4.30.3.4 __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.

```

258 {
259     // first, check for idealized wave breaking (deep water)
260     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
261         return 0;
262     }
263
264     // otherwise, apply generic quadratic performance model
265     // (with outputs bounded to [0, 1])
266     double production =
267         0.289 * significant_wave_height_m -
268         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
269         0.0169 * energy_period_s;
270
271     if (production < 0) {
272         production = 0;
273     }
274
275     else if (production > 1) {
276         production = 1;
277     }
278
279     return production * this->capacity_kW;
280 } /* __computeParaboloidProductionkW() */

```

4.30.3.5 __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].

```

128 {
129     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
130
131     return capital_cost_per_kW * this->capacity_kW;
132 } /* __getGenericCapitalCost() */

```

4.30.3.6 __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].

```

156 {
157     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
158
159     return operation_maintenance_cost_kWh;
160 } /* __getGenericOpMaintCost() */

```

4.30.3.7 __writeSummary()

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

Helper method to write summary 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.
-------------------	--

Reimplemented from [Renewable](#).

```
344 {
345     // 1. create filestream
346     write_path += "summary_results.md";
347     std::ofstream ofs;
348     ofs.open(write_path, std::ofstream::out);
349
350     // 2. write summary results (markdown)
351     ofs << "# ";
352     ofs << std::to_string(int(ceil(this->capacity_kW)));
353     ofs << " kW WAVE Summary Results\n";
354     ofs << "\n-----\n\n";
355
356     // 2.1. Production attributes
357     ofs << "## Production Attributes\n";
358     ofs << "\n";
359
360     ofs << "Capacity: " << this->capacity_kW << " kW \n";
361     ofs << "\n";
362
363     ofs << "Production Override: (N = 0 / Y = 1): "
364         << this->normalized_production_series_given << " \n";
365     if (this->normalized_production_series_given) {
366         ofs << "Path to Normalized Production Time Series: "
367             << this->path_2_normalized_production_time_series << " \n";
368     }
369     ofs << "\n";
370
371     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
372     ofs << "Capital Cost: " << this->capital_cost << " \n";
373     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
374         << " per kWh produced \n";
375     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
376         << " \n";
377     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
378         << " \n";
379     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
380     ofs << "\n";
381
382     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
383     ofs << "\n-----\n\n";
384
385     // 2.2. Renewable attributes
386     ofs << "## Renewable Attributes\n";
387     ofs << "\n";
388
389     ofs << "Resource Key (2D): " << this->resource_key << " \n";
390
391     ofs << "\n-----\n\n";
392
393     // 2.3. Wave attributes
394     ofs << "## Wave Attributes\n";
395     ofs << "\n";
396
397     ofs << "Power Production Model: " << this->power_model_string << " \n";
398     switch (this->power_model) {
399     case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
400         ofs << "Design Significant Wave Height: "
401             << this->design_significant_wave_height_m << " m \n";
402
403         ofs << "Design Energy Period: " << this->design_energy_period_s << " s \n";
404
405         break;
406     }
407
408     case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
```

```

409         ofs << "Normalized Performance Matrix: "
410             << this->interpolator.path_map_2D[0] << " \n";
411
412         break;
413     }
414
415     default: {
416         // write nothing!
417
418         break;
419     }
420 }
421
422 ofs << "\n-----\n\n";
423
424 // 2.4. Wave Results
425 ofs << "## Results\n";
426 ofs << "\n";
427
428 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
429 ofs << "\n";
430
431 ofs << "Total Dispatch: " << this->total_dispatch_kWh
432     << " kWh \n";
433
434 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
435     << " per kWh dispatched \n";
436 ofs << "\n";
437
438 ofs << "Running Hours: " << this->running_hours << " \n";
439 ofs << "Replacements: " << this->n_replacements << " \n";
440
441 ofs << "\n-----\n\n";
442
443 ofs.close();
444
445 return;
446 } /* __writeSummary() */

```

4.30.3.8 __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 <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](#).

```

484 {
485     // 1. create filestream
486     write_path += "time_series_results.csv";
487     std::ofstream ofs;
488     ofs.open(write_path, std::ofstream::out);
489
490     // 2. write time series results (comma separated value)

```

```

491     ofs << "Time (since start of data) [hrs],";
492     ofs << "Significant Wave Height [m],";
493     ofs << "Energy Period [s],";
494     ofs << "Production [kW],";
495     ofs << "Dispatch [kW],";
496     ofs << "Storage [kW],";
497     ofs << "Curtailment [kW],";
498     ofs << "Capital Cost (actual),";
499     ofs << "Operation and Maintenance Cost (actual),";
500     ofs << "\n";
501
502     for (int i = 0; i < max_lines; i++) {
503         ofs << time_vec_hrs_ptr->at(i) << ", ";
504
505         if (not this->normalized_production_series_given) {
506             ofs << resource_map_2D_ptr->at(this->resource_key)[i][0] << ", ";
507             ofs << resource_map_2D_ptr->at(this->resource_key)[i][1] << ", ";
508         }
509
510         else {
511             ofs << "OVERRIDE" << ", ";
512             ofs << "OVERRIDE" << ", ";
513         }
514
515         ofs << this->production_vec_kW[i] << ", ";
516         ofs << this->dispatch_vec_kW[i] << ", ";
517         ofs << this->storage_vec_kW[i] << ", ";
518         ofs << this->curtailment_vec_kW[i] << ", ";
519         ofs << this->capital_cost_vec[i] << ", ";
520         ofs << this->operation_maintenance_cost_vec[i] << ", ";
521         ofs << "\n";
522     }
523
524     return;
525 } /* __writeTimeSeries() */

```

4.30.3.9 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](#).

```

830 {
831     // 1. invoke base class method
832     load_kW = Renewable::commit(
833         timestep,
834         dt_hrs,
835         production_kW,

```

```

836         load_kW
837     );
838
839
840     //...
841
842     return load_kW;
843 } /* commit() */

```

4.30.3.10 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>signficiant_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](#).

```

727 {
728     // given production time series override
729     if (this->normalized_production_series_given) {
730         double production_kW = Production::getProductionkW(timestep);
731         return production_kW;
732     }
733
734     // check if no resource
735     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
736         return 0;
737     }
738
739     // compute production
740     double production_kW = 0;
741
742     switch (this->power_model) {
743     case (WavePowerProductionModel::WAVE_POWER_PARABOLOID): {
744         production_kW = this->__computeParaboloidProductionkW(
745             timestep,
746             dt_hrs,
747             significant_wave_height_m,
748             energy_period_s
749         );
750         break;
751     }
752     case (WavePowerProductionModel::WAVE_POWER_GAUSSIAN): {
753         production_kW = this->__computeGaussianProductionkW(
754             timestep,
755             dt_hrs,

```

```

759         significant_wave_height_m,
760         energy_period_s
761     );
762
763     break;
764 }
765
766 case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
767     production_kW = this->__computeLookupProductionkW(
768         timestep,
769         dt_hrs,
770         significant_wave_height_m,
771         energy_period_s
772     );
773
774     break;
775 }
776
777 default: {
778     std::string error_str = "ERROR: Wave::computeProductionkW(): ";
779     error_str += "power model ";
780     error_str += std::to_string(this->power_model);
781     error_str += " not recognized";
782
783     #ifdef _WIN32
784         std::cout << error_str << std::endl;
785     #endif
786
787     throw std::runtime_error(error_str);
788
789     break;
790 }
791 }
792
793 return production_kW;
794 } /* computeProductionkW() */

```

4.30.3.11 handleReplacement()

```

void Wave::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

683 {
684     // 1. reset attributes
685     //...
686
687     // 2. invoke base class method
688     Renewable :: handleReplacement(timestep);
689
690     return;
691 } /* __handleReplacement() */

```

4.30.4 Member Data Documentation

4.30.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

4.30.4.2 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.

4.30.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

4.30.4.4 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:

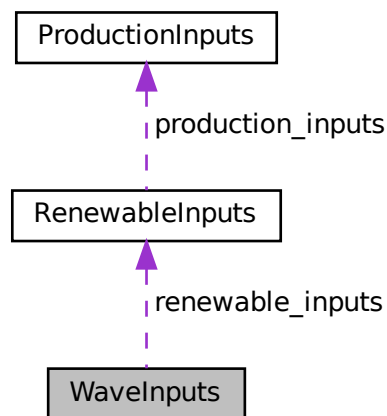
- [header/Production/Renewable/Wave.h](#)
- [source/Production/Renewable/Wave.cpp](#)

4.31 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.
- std::string `path_2_normalized_performance_matrix` = ""
A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.1 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](#).

4.31.2 Member Data Documentation

4.31.2.1 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].

4.31.2.2 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.

4.31.2.3 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.

4.31.2.4 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].

4.31.2.5 path_2_normalized_performance_matrix

```
std::string WaveInputs::path_2_normalized_performance_matrix = ""
```

A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.2.6 power_model

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

The wave power production model to be applied.

4.31.2.7 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.31.2.8 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:

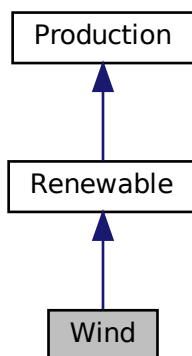
- [header/Production/Renewable/Wave.h](#)

4.32 Wind Class Reference

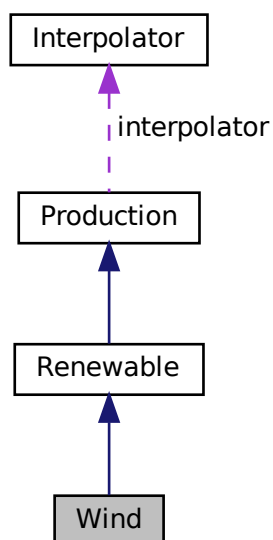
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

```
#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](#), std::vector< double > *)
Constructor (intended) for the [Wind](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Wind](#) (void)
Destructor for the [Wind](#) class.

Public Attributes

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

Private Member Functions

- void [__checkInputs](#) ([WindInputs](#))
Helper method to check inputs to the [Wind](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic wind turbine capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__computeCubicProductionkW](#) (int, double, double)
Helper method to compute wind turbine production under a cubic production model.
- 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](#).

4.32.1 Detailed Description

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

4.32.2 Constructor & Destructor Documentation

4.32.2.1 Wind() [1/2]

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

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

```
501 {
502     return;
503 } /* Wind() */
```

4.32.2.2 Wind() [2/2]

```
Wind::Wind (
    int n_points,
    double n_years,
    WindInputs wind_inputs,
    std::vector< double > * time_vec_hrs_ptr )
```

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.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```
535 :
536 Renewable(
537     n_points,
538     n_years,
539     wind_inputs.renewable_inputs,
540     time_vec_hrs_ptr
541 )
542 {
543     // 1. check inputs
544     this->__checkInputs(wind_inputs);
545
546     // 2. set attributes
547     this->type = RenewableType :: WIND;
548     this->type_str = "WIND";
549
550     this->resource_key = wind_inputs.resource_key;
551
552     this->design_speed_ms = wind_inputs.design_speed_ms;
553
554     this->power_model = wind_inputs.power_model;
555
556     switch (this->power_model) {
557         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
558             this->power_model_string = "CUBIC";
559
560             break;
561         }
562
563         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
564             this->power_model_string = "EXPONENTIAL";
565
566         }
```

```

566         break;
567     }
568
569     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
570         this->power_model_string = "LOOKUP";
571
572         break;
573     }
574
575     default: {
576         std::string error_str = "ERROR: Wind(): ";
577         error_str += "power production model ";
578         error_str += std::to_string(this->power_model);
579         error_str += " not recognized";
580
581         #ifdef _WIN32
582             std::cout << error_str << std::endl;
583         #endif
584
585         throw std::runtime_error(error_str);
586
587         break;
588     }
589 }
590
591 if (wind_inputs.capital_cost < 0) {
592     this->capital_cost = this->__getGenericCapitalCost();
593 }
594 else {
595     this->capital_cost = wind_inputs.capital_cost;
596 }
597
598 if (wind_inputs.operation_maintenance_cost_kWh < 0) {
599     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
600 }
601 else {
602     this->operation_maintenance_cost_kWh =
603         wind_inputs.operation_maintenance_cost_kWh;
604 }
605
606 if (not this->is_sunk) {
607     this->capital_cost_vec[0] = this->capital_cost;
608 }
609
610 // 3. construction print
611 if (this->print_flag) {
612     std::cout << "Wind object constructed at " << this << std::endl;
613 }
614
615 return;
616 } /* Renewable() */

```

4.32.2.3 ~Wind()

```

Wind::~~Wind (
    void )

```

Destructor for the [Wind](#) class.

```

802 {
803     // 1. destruction print
804     if (this->print_flag) {
805         std::cout << "Wind object at " << this << " destroyed" << std::endl;
806     }
807
808     return;
809 } /* ~Wind() */

```

4.32.3 Member Function Documentation

4.32.3.1 `__checkInputs()`

```
void Wind::__checkInputs (
    WindInputs wind_inputs ) [private]
```

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

Ref: [Zafar \[2018\]](#)

Parameters

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

```
66 {
67     // 1. check design_speed_ms
68     if (wind_inputs.design_speed_ms <= 0) {
69         std::string error_str = "ERROR: Wind(): ";
70         error_str += "WindInputs::design_speed_ms must be > 0";
71
72         #ifdef WIN32
73             std::cout << error_str << std::endl;
74         #endif
75
76         throw std::invalid_argument(error_str);
77     }
78
79     else if (wind_inputs.design_speed_ms < 12) {
80         std::string warning_str = "WARNING: Wind(): ";
81         warning_str += "Setting WindInputs::design_speed_ms to less than 12 m/s may be ";
82         warning_str += "technically unrealistic";
83
84         std::cout << warning_str << std::endl;
85     }
86
87     return;
88 } /* __checkInputs() */
```

4.32.3.2 `__computeCubicProductionkW()`

```
double Wind::__computeCubicProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]
```

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

Ref: [Milan et al. \[2010\]](#)

Ref: [Zafar \[2018\]](#)

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.

```

176 {
177     double production = 0;
178
179     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
180         this->design_speed_ms;
181
182     if (turbine_speed < -0.7857 or turbine_speed > 0.7857) {
183         production = 0;
184     }
185
186     else if (turbine_speed >= -0.7857 and turbine_speed <= 0) {
187         production = (1 / pow(this->design_speed_ms, 3)) * pow(wind_resource_ms, 3);
188     }
189
190     else {
191         production = 1;
192     }
193
194     return production * this->capacity_kW;
195 } /* __computeCubicProductionkW() */

```

4.32.3.3 __computeExponentialProductionkW()

```

double Wind::__computeExponentialProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]

```

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

Ref: [Truelove et al. \[2019\]](#)

Parameters

<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.

```

229 {
230     double production = 0;
231
232     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
233         this->design_speed_ms;
234
235     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
236         production = 0;
237     }
238
239     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
240         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
241     }
242
243     else {
244         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
245     }
246
247     return production * this->capacity_kW;
248 } /* __computeExponentialProductionkW() */

```

4.32.3.4 `__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.

```
280 {
281     // *** WORK IN PROGRESS *** //
282
283     return 0;
284 } /* __computeLookupProductionkW() */
```

4.32.3.5 `__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].

```
110 {
111     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
112
113     return capital_cost_per_kW * this->capacity_kW;
114 } /* __getGenericCapitalCost() */
```

4.32.3.6 `__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].

```
137 {
138     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
139
140     return operation_maintenance_cost_kWh;
141 } /* __getGenericOpMaintCost() */
```

4.32.3.7 `__writeSummary()`

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

Helper method to write summary 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.
-------------------	--

Reimplemented from [Renewable](#).

```
302 {
303     // 1. create filestream
304     write_path += "summary_results.md";
305     std::ofstream ofs;
306     ofs.open(write_path, std::ofstream::out);
307
308     // 2. write summary results (markdown)
309     ofs << "# ";
310     ofs << std::to_string(int(ceil(this->capacity_kW)));
311     ofs << " kW WIND Summary Results\n";
312     ofs << "\n-----\n\n";
313
314
315     // 2.1. Production attributes
316     ofs << "## Production Attributes\n";
317     ofs << "\n";
318
319     ofs << "Capacity: " << this->capacity_kW << " kW \n";
320     ofs << "\n";
321
322     ofs << "Production Override: (N = 0 / Y = 1): "
323         << this->normalized_production_series_given << " \n";
324     if (this->normalized_production_series_given) {
325         ofs << "Path to Normalized Production Time Series: "
326             << this->path_2_normalized_production_time_series << " \n";
327     }
328     ofs << "\n";
329
330     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
331     ofs << "Capital Cost: " << this->capital_cost << " \n";
```

```

332 ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
333     << " per kWh produced \n";
334 ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
335     << " \n";
336 ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
337     << " \n";
338 ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
339 ofs << "\n";
340
341 ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
342 ofs << "\n-----\n\n";
343
344 // 2.2. Renewable attributes
345 ofs << "## Renewable Attributes\n";
346 ofs << "\n";
347
348 ofs << "Resource Key (ID): " << this->resource_key << " \n";
349
350 ofs << "\n-----\n\n";
351
352 // 2.3. Wind attributes
353 ofs << "## Wind Attributes\n";
354 ofs << "\n";
355
356 ofs << "Power Production Model: " << this->power_model_string << " \n";
357 switch (this->power_model) {
358     case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
359         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
360
361         break;
362     }
363
364     case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
365         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
366
367         break;
368     }
369
370     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
371         //...
372
373         break;
374     }
375
376     default: {
377         // write nothing!
378
379         break;
380     }
381 }
382
383 ofs << "\n-----\n\n";
384
385 // 2.4. Wind Results
386 ofs << "## Results\n";
387 ofs << "\n";
388
389 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
390 ofs << "\n";
391
392 ofs << "Total Dispatch: " << this->total_dispatch_kWh
393     << " kWh \n";
394
395 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
396     << " per kWh dispatched \n";
397 ofs << "\n";
398
399 ofs << "Running Hours: " << this->running_hours << " \n";
400 ofs << "Replacements: " << this->n_replacements << " \n";
401
402 ofs << "\n-----\n\n";
403
404 ofs.close();
405
406 return;
407 } /* __writeSummary() */

```

4.32.3.8 __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](#).

```
445 {
446     // 1. create filestream
447     write_path += "time_series_results.csv";
448     std::ofstream ofs;
449     ofs.open(write_path, std::ofstream::out);
450
451     // 2. write time series results (comma separated value)
452     ofs << "Time (since start of data) [hrs],";
453     ofs << "Wind Resource [m/s],";
454     ofs << "Production [kW],";
455     ofs << "Dispatch [kW],";
456     ofs << "Storage [kW],";
457     ofs << "Curtailement [kW],";
458     ofs << "Capital Cost (actual),";
459     ofs << "Operation and Maintenance Cost (actual),";
460     ofs << "\n";
461
462     for (int i = 0; i < max_lines; i++) {
463         ofs << time_vec_hrs_ptr->at(i) << ", ";
464
465         if (not this->normalized_production_series_given) {
466             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ", ";
467         }
468
469         else {
470             ofs << "OVERRIDE" << ", ";
471         }
472
473         ofs << this->production_vec_kW[i] << ", ";
474         ofs << this->dispatch_vec_kW[i] << ", ";
475         ofs << this->storage_vec_kW[i] << ", ";
476         ofs << this->curtailement_vec_kW[i] << ", ";
477         ofs << this->capital_cost_vec[i] << ", ";
478         ofs << this->operation_maintenance_cost_vec[i] << ", ";
479         ofs << "\n";
480     }
481
482     return;
483 } /* __writeTimeSeries() */
```

4.32.3.9 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](#).

```

774 {
775     // 1. invoke base class method
776     load_kW = Renewable::commit(
777         timestep,
778         dt_hrs,
779         production_kW,
780         load_kW
781     );
782
783
784     //...
785
786     return load_kW;
787 } /* commit() */

```

4.32.3.10 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](#).

```

674 {
675     // given production time series override
676     if (this->normalized_production_series_given) {
677         double production_kW = Production::getProductionkW(timestep);
678
679         return production_kW;
680     }
681

```

```

682     // check if no resource
683     if (wind_resource_ms <= 0) {
684         return 0;
685     }
686
687     // compute production
688     double production_kW = 0;
689
690     switch (this->power_model) {
691         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
692             production_kW = this->__computeCubicProductionkW(
693                 timestep,
694                 dt_hrs,
695                 wind_resource_ms
696             );
697             break;
698         }
699
700         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
701             production_kW = this->__computeExponentialProductionkW(
702                 timestep,
703                 dt_hrs,
704                 wind_resource_ms
705             );
706             break;
707         }
708
709         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
710             production_kW = this->__computeLookupProductionkW(
711                 timestep,
712                 dt_hrs,
713                 wind_resource_ms
714             );
715             break;
716         }
717
718         default: {
719             std::string error_str = "ERROR: Wind::computeProductionkW(): ";
720             error_str += "power model ";
721             error_str += std::to_string(this->power_model);
722             error_str += " not recognized";
723
724             #ifdef _WIN32
725                 std::cout << error_str << std::endl;
726             #endif
727
728             throw std::runtime_error(error_str);
729
730             break;
731         }
732     }
733
734     return production_kW;
735 }
736
737 /* computeProductionkW() */
738 }

```

4.32.3.11 handleReplacement()

```

void Wind::handleReplacement (
    int timestep ) [virtual]

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

634 {
635     // 1. reset attributes
636     //...

```

```
637
638     // 2. invoke base class method
639     Renewable :: handleReplacement(timestep);
640
641     return;
642 } /* __handleReplacement() */
```

4.32.4 Member Data Documentation

4.32.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

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

4.32.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

4.32.4.3 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:

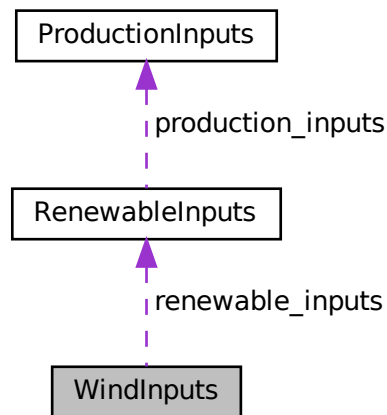
- header/Production/Renewable/[Wind.h](#)
- source/Production/Renewable/[Wind.cpp](#)

4.33 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` = 14
The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- [WindPowerProductionModel](#) `power_model` = [WindPowerProductionModel](#) :: `WIND_POWER_CUBIC`
The wind power production model to be applied.

4.33.1 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](#).

4.33.2 Member Data Documentation

4.33.2.1 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].

4.33.2.2 design_speed_ms

```
double WindInputs::design_speed_ms = 14
```

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

4.33.2.3 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].

4.33.2.4 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel :: WIND_POWER_CUBIC
```

The wind power production model to be applied.

4.33.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.33.2.6 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

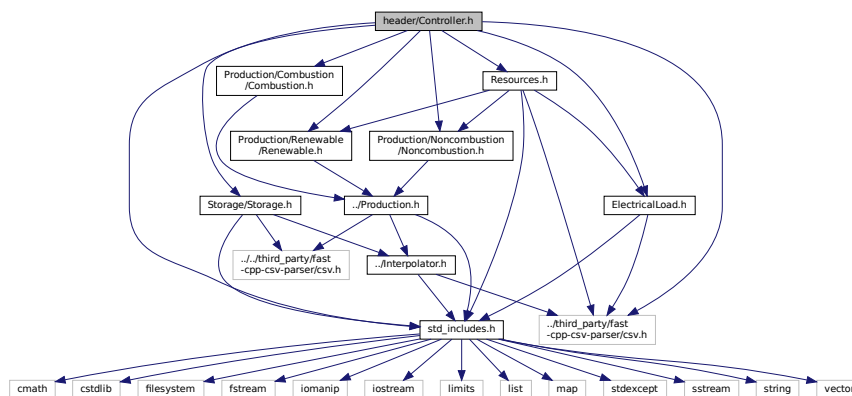
File Documentation

5.1 header/Controller.h File Reference

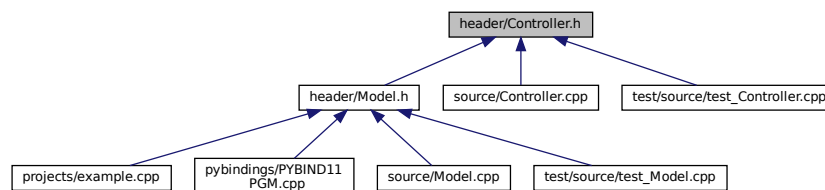
Header file for the [Controller](#) class.

```
#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/Noncombustion/Noncombustion.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 for 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.

```

69     {
70         LOAD\_FOLLOWING,
71         CYCLE\_CHARGING,
72         N\_CONTROL\_MODES
73     };

```

5.2 header/doxygen_cite.h File Reference

Header file which simply cites the doxygen tool.

5.2.1 Detailed Description

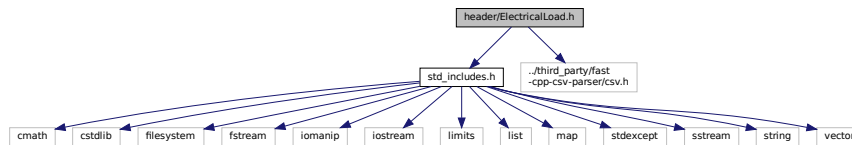
Header file which simply cites the doxygen tool.

Ref: [van Heesch](#). [2023]

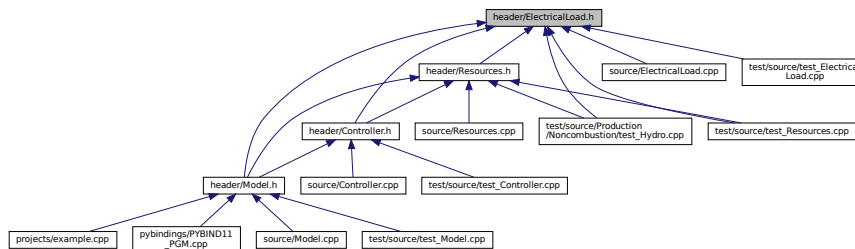
5.3 header/ElectricalLoad.h File Reference

Header file for 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](#).

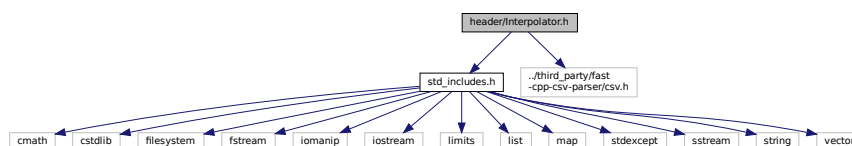
5.3.1 Detailed Description

Header file for the [ElectricalLoad](#) class.

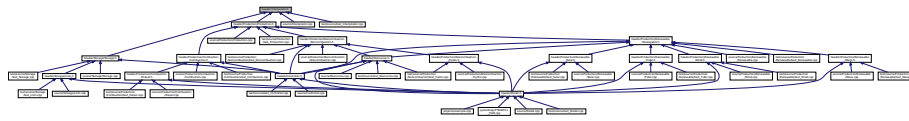
5.4 header/Interpolator.h File Reference

Header file for the [Interpolator](#) class.

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



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



Classes

- struct [InterpolatorStruct1D](#)
A struct which holds two parallel vectors for use in 1D interpolation.
- struct [InterpolatorStruct2D](#)
A struct which holds two parallel vectors and a matrix for use in 2D interpolation.
- class [Interpolator](#)
A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

5.4.1 Detailed Description

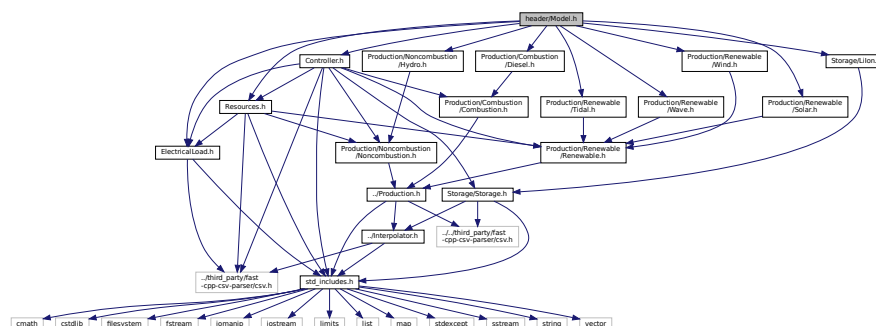
Header file for the [Interpolator](#) class.

5.5 header/Model.h File Reference

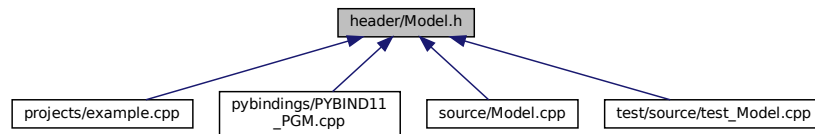
Header file for the [Model](#) class.

```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Noncombustion/Hydro.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
#include "Production/Renewable/Wave.h"
#include "Production/Renewable/Wind.h"
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:



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



Classes

- struct [ModelInputs](#)

A structure which bundles the necessary inputs for the [Model](#) constructor. Provides default values for every necessary input (except `path_2_electrical_load_time_series`, for which a valid input must be provided).

- class [Model](#)

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

5.5.1 Detailed Description

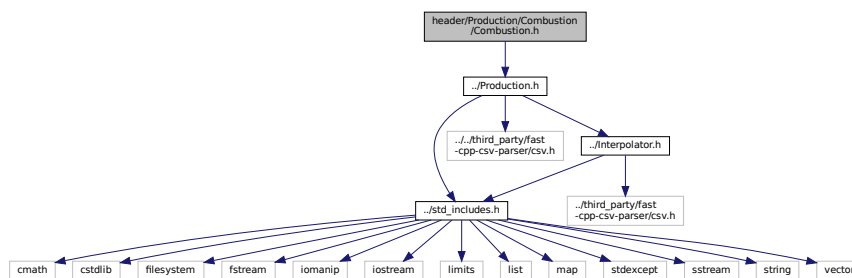
Header file for the [Model](#) class.

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

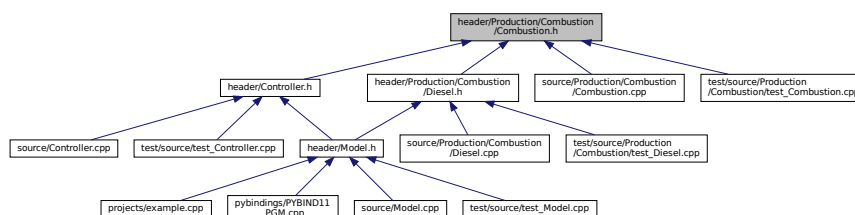
Header file for the [Combustion](#) class.

```
#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.
- enum [FuelMode](#) { [FUEL_MODE_LINEAR](#) , [FUEL_MODE_LOOKUP](#) , [N_FUEL_MODES](#) }
An enumeration of the fuel modes for the [Combustion](#) asset which are supported by PGMcpp.

5.6.1 Detailed Description

Header file for the [Combustion](#) class.

Header file for the [Noncombustion](#) class.

5.6.2 Enumeration Type Documentation

5.6.2.1 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.

```
58         {
59     DIESEL,
60     N\_COMBUSTION\_TYPES
61 };
```

5.6.2.2 FuelMode

```
enum FuelMode
```


An enumeration of the fuel modes for the [Combustion](#) asset which are supported by PGMcpp.

Enumerator

FUEL_MODE_LINEAR	A linearized fuel curve model (i.e., HOMER-like model)
FUEL_MODE_LOOKUP	Interpolating over a given fuel lookup table.
N_FUEL_MODES	A simple hack to get the number of elements in FuelMode.

```

71     {
72         FUEL_MODE_LINEAR,
73         FUEL_MODE_LOOKUP,
74         N_FUEL_MODES
75     };

```

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

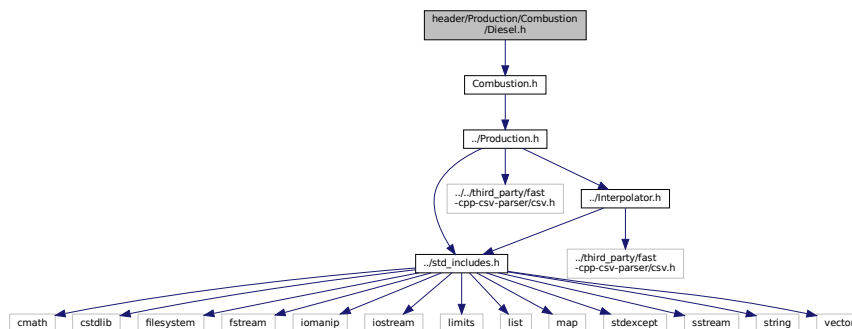
Header file for the [Diesel](#) class.

```

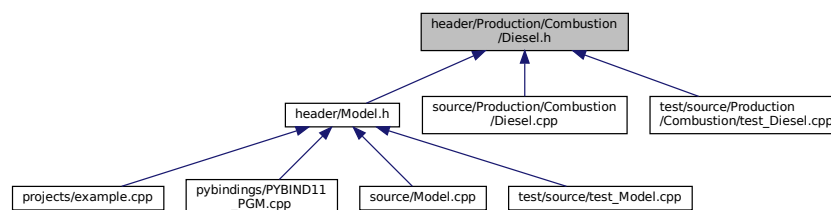
#include "Combustion.h"

```

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.

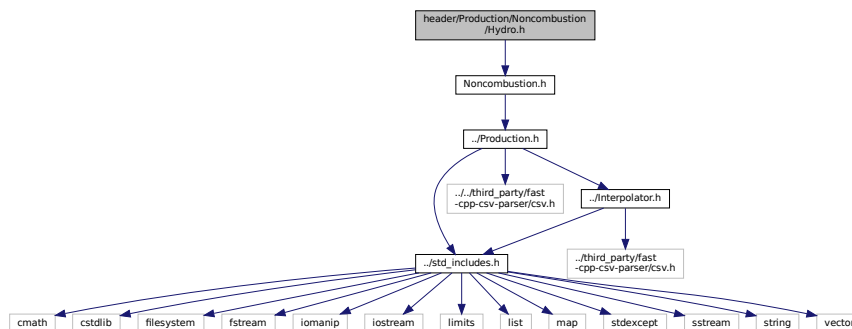
5.7.1 Detailed Description

Header file for the [Diesel](#) class.

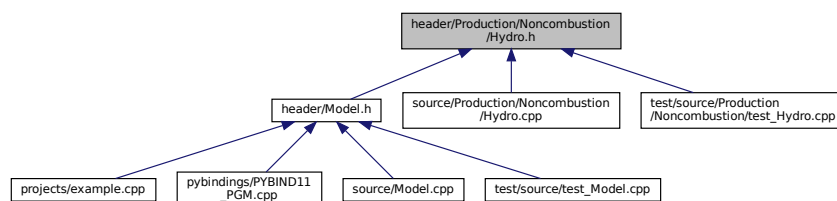
5.8 header/Production/Noncombustion/Hydro.h File Reference

Header file for the [Hydro](#) class.

```
#include "Noncombustion.h"
Include dependency graph for Hydro.h:
```



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



Classes

- struct [HydroInputs](#)

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

- class [Hydro](#)

A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

Enumerations

- enum [HydroTurbineType](#) { [HYDRO_TURBINE_PELTON](#) , [HYDRO_TURBINE_FRANCIS](#) , [HYDRO_TURBINE_KAPLAN](#) , [N_HYDRO_TURBINES](#) }

An enumeration of the types of hydroelectric turbine supported by PGMcpp.

- enum [HydroInterpKeys](#) { [GENERATOR_EFFICIENCY_INTERP_KEY](#) , [TURBINE_EFFICIENCY_INTERP_KEY](#) , [FLOW_TO_POWER_INTERP_KEY](#) , [N_HYDRO_INTERP_KEYS](#) }

An enumeration of the [Interpolator](#) keys used by the [Hydro](#) asset.

5.8.1 Detailed Description

Header file for the [Hydro](#) class.

5.8.2 Enumeration Type Documentation

5.8.2.1 HydroInterpKeys

enum [HydroInterpKeys](#)

An enumeration of the [Interpolator](#) keys used by the [Hydro](#) asset.

Enumerator

GENERATOR_EFFICIENCY_INTERP_KEY	The key for generator efficiency interpolation.
TURBINE_EFFICIENCY_INTERP_KEY	The key for turbine efficiency interpolation.
FLOW_TO_POWER_INTERP_KEY	The key for flow to power interpolation.
N_HYDRO_INTERP_KEYS	A simple hack to get the number of elements in HydroInterpKeys.

```

72         {
73     GENERATOR\_EFFICIENCY\_INTERP\_KEY,
74     TURBINE\_EFFICIENCY\_INTERP\_KEY,
75     FLOW\_TO\_POWER\_INTERP\_KEY,
76     N\_HYDRO\_INTERP\_KEYS
77 };

```

5.8.2.2 HydroTurbineType

enum [HydroTurbineType](#)

An enumeration of the types of hydroelectric turbine supported by PGMcpp.

Enumerator

HYDRO_TURBINE_PELTON	A Pelton turbine (impluse)
HYDRO_TURBINE_FRANCIS	A Francis turbine (reaction)
HYDRO_TURBINE_KAPLAN	A Kaplan turbine (reaction)
N_HYDRO_TURBINES	A simple hack to get the number of elements in HydroTurbineType.

```

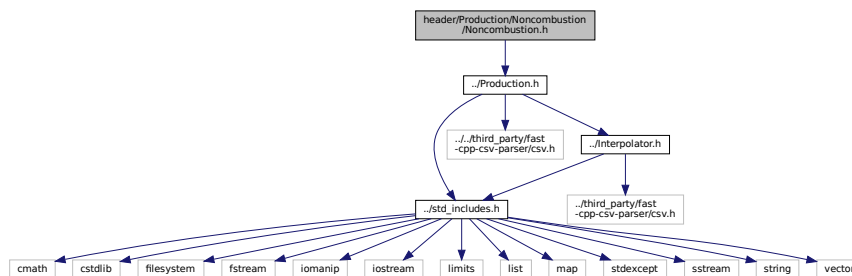
58     {
59         HYDRO_TURBINE_PELTON,
60         HYDRO_TURBINE_FRANCIS,
61         HYDRO_TURBINE_KAPLAN,
62         N_HYDRO_TURBINES
63     };

```

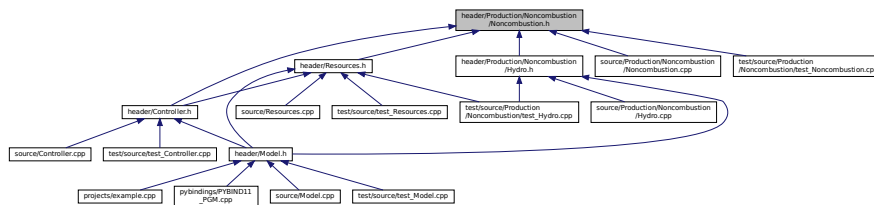
5.9 header/Production/Noncombustion/Noncombustion.h File Reference

```
#include "../Production.h"
```

Include dependency graph for Noncombustion.h:



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



Classes

- struct [NoncombustionInputs](#)

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

- class [Noncombustion](#)

The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

Enumerations

- enum [NoncombustionType](#) { [HYDRO](#) , [N_NONCOMBUSTION_TYPES](#) }

An enumeration of the types of [Noncombustion](#) asset supported by PGMcpp.

5.9.1 Enumeration Type Documentation

5.9.1.1 NoncombustionType

enum `NoncombustionType`

An enumeration of the types of `Noncombustion` asset supported by PGMcpp.

Enumerator

HYDRO	A hydroelectric generator (either with reservoir or not)
N_NONCOMBUSTION_TYPES	A simple hack to get the number of elements in <code>NoncombustionType</code> .

```

58         {
59     HYDRO,
60     N_NONCOMBUSTION_TYPES
61 };

```

5.10 header/Production/Production.h File Reference

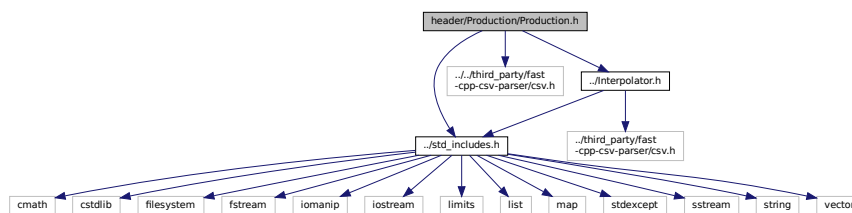
Header file for the `Production` class.

```

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

5.10.1 Detailed Description

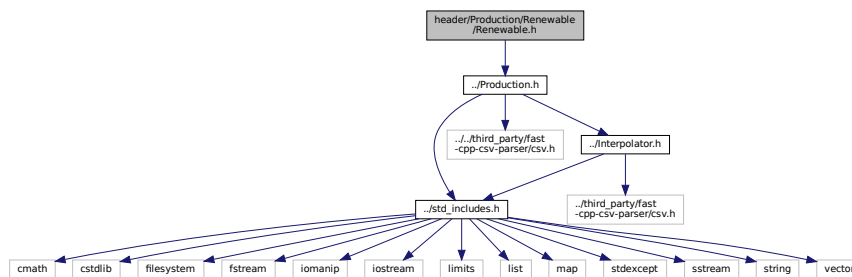
Header file for the [Production](#) class.

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

Header file for 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.

5.11.1 Detailed Description

Header file for the `Renewable` class.

5.11.2 Enumeration Type Documentation

5.11.2.1 RenewableType

enum `RenewableType`

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

Enumerator

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

```

58         {
59     SOLAR,
60     TIDAL,
61     WAVE,
62     WIND,
63     N_RENEWABLE_TYPES
64 };

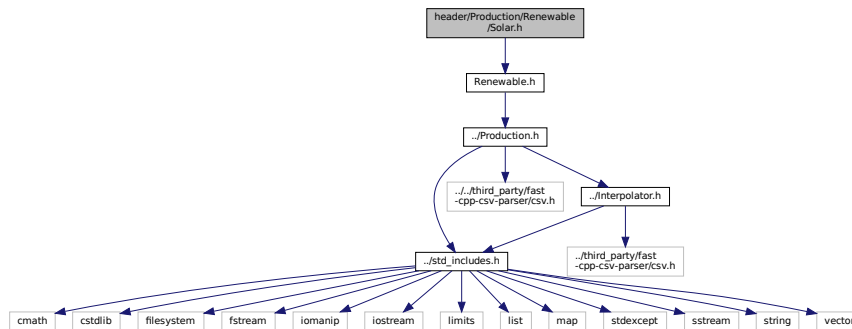
```

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

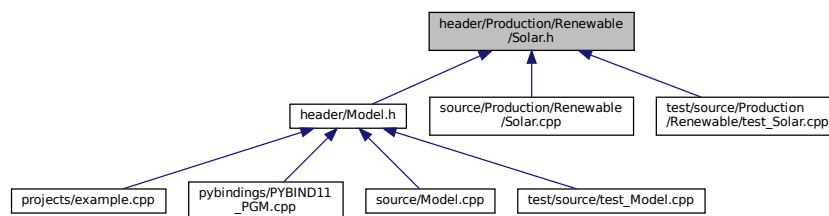
Header file for the `Solar` class.


```
#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.

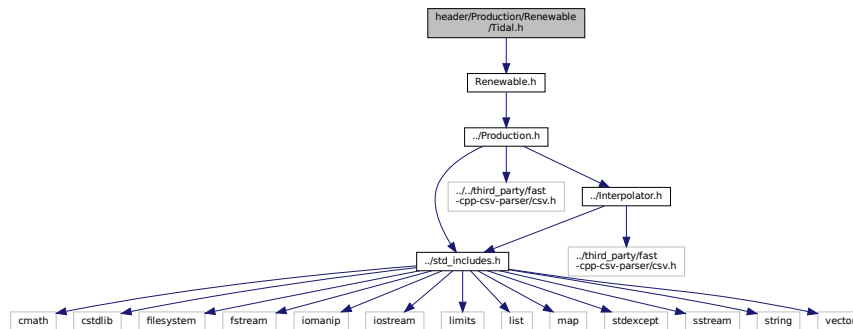
5.12.1 Detailed Description

Header file for the [Solar](#) class.

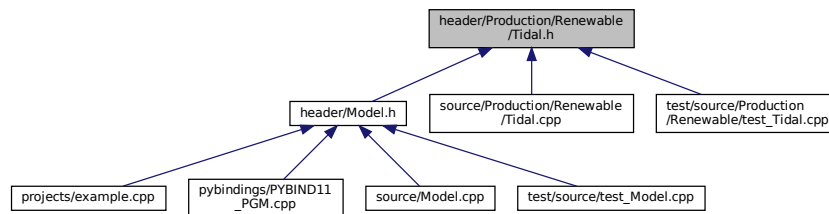
5.13 header/Production/Renewable/Tidal.h File Reference

Header file for 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](#) }

5.13.1 Detailed Description

Header file for the [Tidal](#) class.

5.13.2 Enumeration Type Documentation

5.13.2.1 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.

```

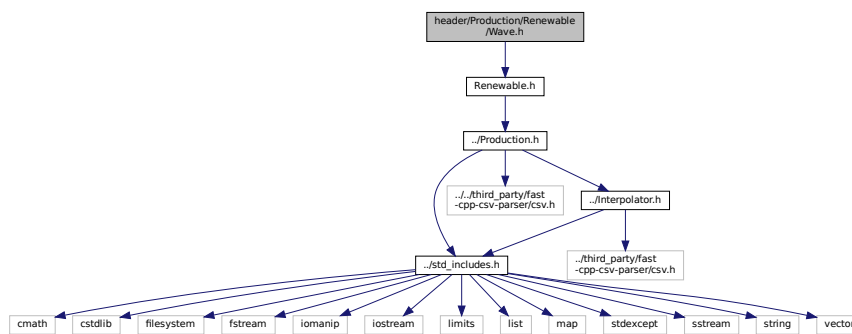
59
60     TIDAL_POWER_CUBIC,
61     TIDAL_POWER_EXPONENTIAL,
62     TIDAL_POWER_LOOKUP,
63     N_TIDAL_POWER_PRODUCTION_MODELS
64 };

```

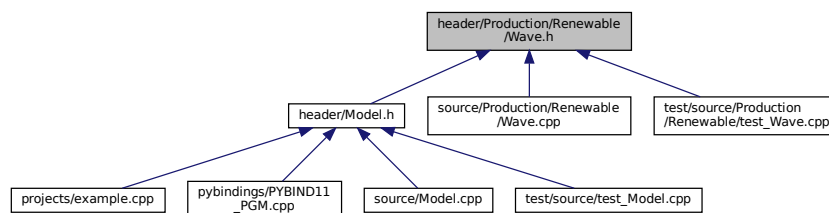
5.14 header/Production/Renewable/Wave.h File Reference

Header file for 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](#) }

5.14.1 Detailed Description

Header file for the [Wave](#) class.

5.14.2 Enumeration Type Documentation

5.14.2.1 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.

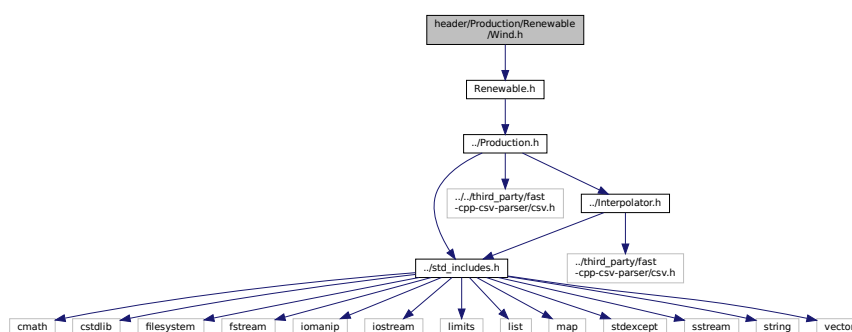
```
59
60     WAVE_POWER_GAUSSIAN,
61     WAVE_POWER_PARABOLOID,
62     WAVE_POWER_LOOKUP,
63     N_WAVE_POWER_PRODUCTION_MODELS
64 };
```

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

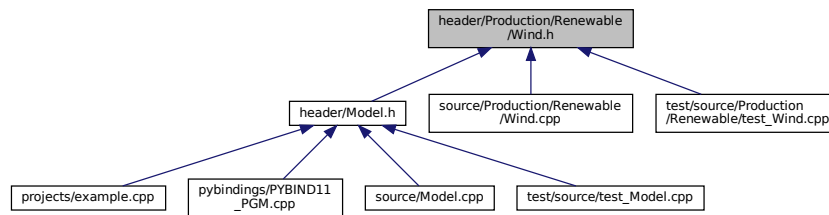
Header file for the [Wind](#) class.

```
#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_CUBIC](#) , [WIND_POWER_EXPONENTIAL](#) , [WIND_POWER_LOOKUP](#) , [N_WIND_POWER_PRODUCTION_MODELS](#) }

5.15.1 Detailed Description

Header file for the [Wind](#) class.

5.15.2 Enumeration Type Documentation

5.15.2.1 WindPowerProductionModel

```
enum WindPowerProductionModel
```

Enumerator

WIND_POWER_CUBIC	A cubic power production model.
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 .

```
59     {
60         WIND\_POWER\_CUBIC,
```

```

61     WIND_POWER_EXPONENTIAL,
62     WIND_POWER_LOOKUP,
63     N_WIND_POWER_PRODUCTION_MODELS
64 };

```

5.16 header/Resources.h File Reference

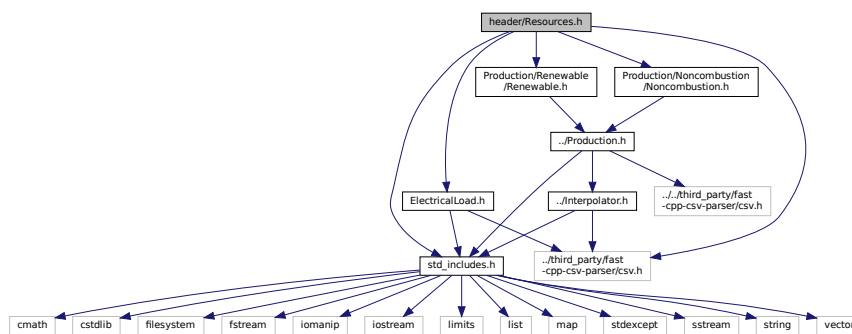
Header file for the [Resources](#) class.

```

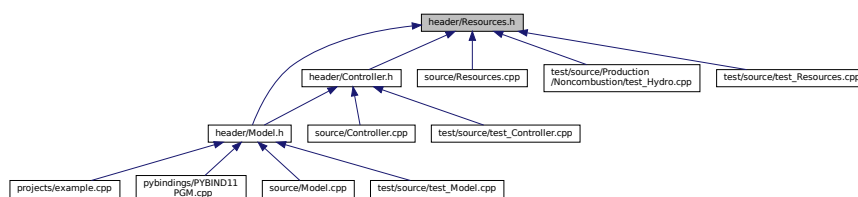
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"

```

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](#).

5.16.1 Detailed Description

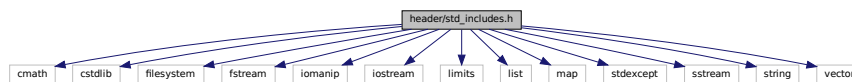
Header file for the [Resources](#) class.

5.17 header/std_includes.h File Reference

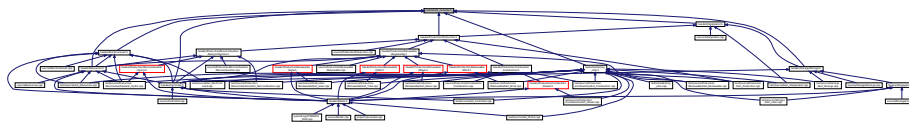
Header file which simply batches together some standard includes.

```
#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:



5.17.1 Detailed Description

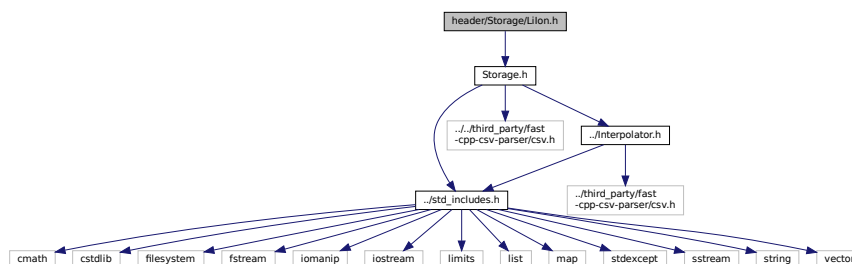
Header file which simply batches together some standard includes.

5.18 header/Storage/Lilon.h File Reference

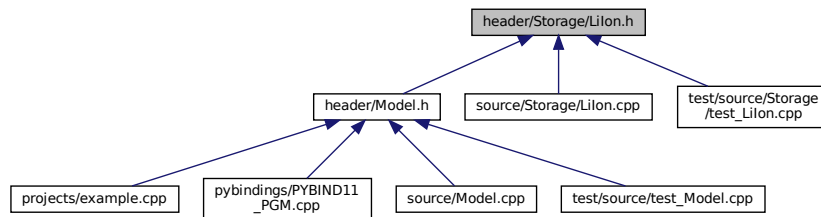
Header file for the [Lilon](#) class.

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

Include dependency graph for Lilon.h:



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



Classes

- struct [LilonInputs](#)

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

- class [Lilon](#)

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

5.18.1 Detailed Description

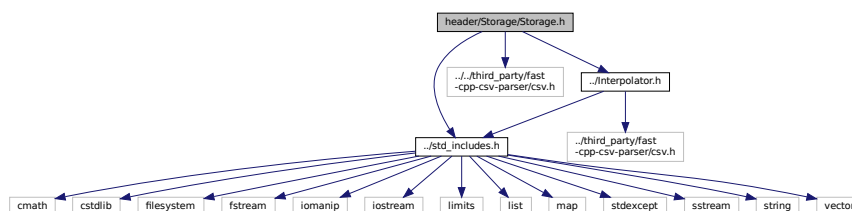
Header file for the [Lilon](#) class.

5.19 header/Storage/Storage.h File Reference

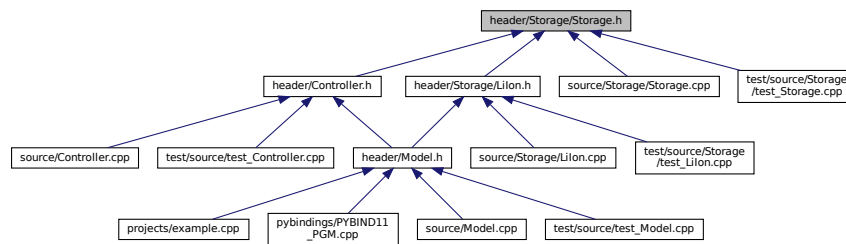
Header file for the [Storage](#) class.

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

Include dependency graph for Storage.h:



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



Classes

- struct [StorageInputs](#)
A structure which bundles the necessary inputs for the [Storage](#) constructor. Provides default values for every necessary input.
- class [Storage](#)
The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

Enumerations

- enum [StorageType](#) { [LIION](#) , [N_STORAGE_TYPES](#) }
An enumeration of the types of [Storage](#) asset supported by PGMcpp.

5.19.1 Detailed Description

Header file for the [Storage](#) class.

5.19.2 Enumeration Type Documentation

5.19.2.1 StorageType

```
enum StorageType
```

An enumeration of the types of [Storage](#) asset supported by PGMcpp.

Enumerator

LIION	A system of lithium ion batteries.
N_STORAGE_TYPES	A simple hack to get the number of elements in StorageType .

```
61     {
62     LIION,
```

```

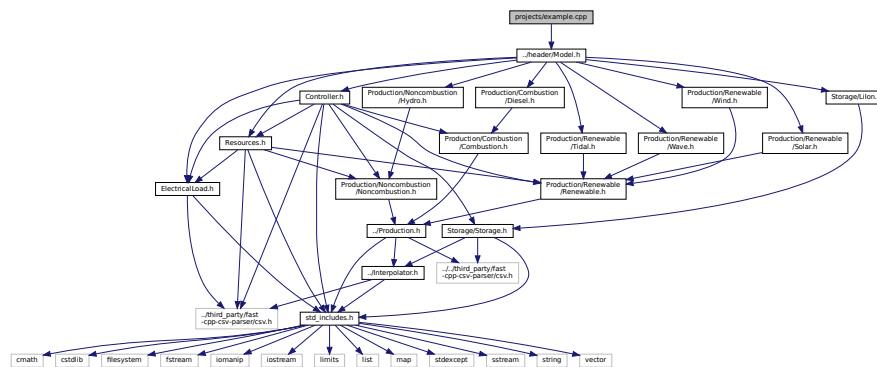
63     N_STORAGE_TYPES
64 };

```

5.20 projects/example.cpp File Reference

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

Include dependency graph for example.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.20.1 Function Documentation

5.20.1.1 main()

```

int main (
    int argc,
    char ** argv )
{
    /*
    * 1. construct Model object
    *
    * This block constructs a Model object, which is the central container for the
    * entire microgrid model.
    *
    * The first argument that must be provided to the Model constructor is a valid
    * path (either relative or absolute) to a time series of electrical load data.
    * For an example of the expected format, see
    *
    * data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv
    *
    * Note that the length of the given electrical load time series defines the
    * modelled project life (so if you want to model n years of microgrid operation,
    * then you must pass a path to n years worth of electrical load data). In addition,
    * the given electrical load time series defines which points in time are modelled.
    * As such, all subsequent time series data which is passed in must (1) be of the
    * same length as the electrical load time series, and (2) provide data for the
    * same set of points in time. Of course, the electrical load time series can be
    * of arbitrary length, and it need not be a uniform time series.
    *
    * The second argument that one can provide is the desired dispatch control mode.
    */
}

```

```

74      * If nothing is given here, then the model will default to simple load following
75      * control. However, one can stipulate which control mode to use by altering the
76      * control_mode attribute of the ModelInputs structure. In this case, the
77      * cycle charging control mode is being set.
78      */
79
80      std::string path_2_electrical_load_time_series =
81          "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
82
83      ModelInputs model_inputs;
84
85      model_inputs.path_2_electrical_load_time_series =
86          path_2_electrical_load_time_series;
87
88      model_inputs.control_mode = ControlMode :: CYCLE_CHARGING;
89
90      Model model(model_inputs);
91
92
93
94      /*
95      * 2. add Diesel objects to Model
96      *
97      * This block defines and adds a set of diesel generators to the Model object.
98      *
99      * In this example, a single DieselInputs structure is used to define and add
100     * three diesel generators to the model.
101     *
102     * The first diesel generator is defined as a 300 kW generator (which shows an
103     * example of how to access and alter an encapsulated attribute of DieselInputs).
104     * In addition, the diesel generator is taken to be a sunk cost (and so no capital
105     * cost is incurred in the first time step; the opposite is true for non-sunk
106     * assets).
107     *
108     * The last two diesel generators are defined as 150 kW each. Likewise, they are
109     * also sunk assets (since the same DieselInputs structure is being re-used without
110     * overwriting the is_sunk attribute).
111     *
112     * For more details on the various attributes of DieselInputs, refer to the
113     * PGMcpp manual. For instance, note that no economic inputs are given; in this
114     * example, the default values apply.
115     */
116
117     DieselInputs diesel_inputs;
118
119     // 2.1. add 1 x 300 kW diesel generator (since mean load is ~250 kW)
120     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 300;
121     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
122
123     model.addDiesel(diesel_inputs);
124
125     // 2.2. add 2 x 150 kW diesel generators (since max load is 500 kW)
126     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
127
128     model.addDiesel(diesel_inputs);
129     model.addDiesel(diesel_inputs);
130
131
132
133     /*
134     * 3. add renewable resources to Model
135     *
136     * This block adds a set of renewable resource time series to the Model object.
137     *
138     * The first resource added is a solar resource time series, which gives
139     * horizontal irradiance [kW/m2] at each point in time. Again, remember that all
140     * given time series must align with the electrical load time series (i.e., same
141     * length, same points). For an example of the expected format, see
142     *
143     * data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv
144     *
145     * Finally, note the declaration of a solar resource key. This variable will be
146     * re-used later to associate a solar PV array object with this particular solar
147     * resource. This method of key association between resource and asset allows for
148     * greater flexibility in modelling production assets that are exposed to different
149     * renewable resources (due to being geographically separated, etc.).
150     *
151     * The second resource added is a tidal resource time series, which gives tidal
152     * stream speed [m/s] at each point in time. For an example of the expected format,
153     * see
154     *
155     * data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv
156     *
157     * Again, note the tidal resource key.
158     *
159     * The third resource added is a wave resource time series, which gives significant
160     * wave height [m] and energy period [s] at each point in time. For an example of

```

```

161      * the expected format, see
162      *
163      * data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv
164      *
165      * Again, note the wave resource key.
166      *
167      * The fourth resource added is a wind resource time series, which gives wind speed
168      * [m/s] at each point in time. For an example of the expected format, see
169      *
170      * data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv
171      *
172      * Again, note the wind resource key.
173      *
174      * The fifth resource added is a hydro resource time series, which gives inflow
175      * rate [m3/hr] at each point in time. For an example of the expected format, see
176      *
177      * data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv
178      *
179      * Again, note the hydro resource key.
180      */
181
182      // 3.1. add solar resource time series
183      int solar_resource_key = 0;
184      std::string path_2_solar_resource_data =
185          "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
186
187      model.addResource(
188          RenewableType :: SOLAR,
189          path_2_solar_resource_data,
190          solar_resource_key
191      );
192
193      // 3.2. add tidal resource time series
194      int tidal_resource_key = 1;
195      std::string path_2_tidal_resource_data =
196          "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
197
198      model.addResource(
199          RenewableType :: TIDAL,
200          path_2_tidal_resource_data,
201          tidal_resource_key
202      );
203
204      // 3.3. add wave resource time series
205      int wave_resource_key = 2;
206      std::string path_2_wave_resource_data =
207          "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
208
209      model.addResource(
210          RenewableType :: WAVE,
211          path_2_wave_resource_data,
212          wave_resource_key
213      );
214
215      // 3.4. add wind resource time series
216      int wind_resource_key = 3;
217      std::string path_2_wind_resource_data =
218          "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
219
220      model.addResource(
221          RenewableType :: WIND,
222          path_2_wind_resource_data,
223          wind_resource_key
224      );
225
226      // 3.5. add hydro resource time series
227      int hydro_resource_key = 4;
228      std::string path_2_hydro_resource_data =
229          "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
230
231      model.addResource(
232          NoncombustionType :: HYDRO,
233          path_2_hydro_resource_data,
234          hydro_resource_key
235      );
236
237
238      /*
239      * 4. add Hydro object to Model
240      *
241      * This block defines and adds a hydroelectric asset to the Model object.
242      *
243      * In this example, a 300 kW hydroelectric station with a 10,000 m3 reservoir
244      * is defined. The initial reservoir state is set to 50% (so half full), and the
245      * hydroelectric asset is taken to be a sunk asset (so no capital cost incurred
246      * in the first time step). Note the association with the previously given hydro

```

```

248     * resource series by way of the hydro resource key.
249     *
250     * For more details on the various attributes of HydroInputs, refer to the
251     * PGMcpp manual. For instance, note that no economic inputs are given; in this
252     * example, the default values apply.
253     */
254
255     HydroInputs hydro_inputs;
256     hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
257     hydro_inputs.reservoir_capacity_m3 = 10000;
258     hydro_inputs.init_reservoir_state = 0.5;
259     hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
260     hydro_inputs.resource_key = hydro_resource_key;
261
262     model.addHydro(hydro_inputs);
263
264
265
266     /*
267     * 5. add Renewable objects to Model
268     *
269     * This block defines and adds a set of renewable production assets to the Model
270     * object.
271     *
272     * The first block defines and adds a solar PV array to the Model object. In this
273     * example, the installed solar capacity is set to 250 kW. Note the association
274     * with the previously given solar resource series by way of the solar resource
275     * key. Also, note that this asset is not taken as sunk (as the is_sunk attribute
276     * of the SolarInputs structure is unchanged and thus defaults to true). As such,
277     * this asset will incur a capital cost in the first time step.
278     *
279     * For more details on the various attributes of SolarInputs, refer to the PGMcpp
280     * manual. For instance, note that no economic inputs are given; in this
281     * example, the default values apply.
282     *
283     * The second block defines and adds a tidal turbine to the Model object. In this
284     * example, the installed tidal capacity is set to 120 kW. In addition, the design
285     * speed of the asset (i.e., the speed at which the rated capacity is achieved) is
286     * set to 2.5 m/s. Note the association with the previously given tidal resource
287     * series by way of the tidal resource key.
288     *
289     * For more details on the various attributes of TidalInputs, refer to the PGMcpp
290     * manual. For instance, note that no economic inputs are given; in this
291     * example, the default values apply.
292     *
293     * The third block defines and adds a wind turbine to the Model object. In this
294     * example, the installed wind capacity is set to 150 kW. In addition, the design
295     * speed of the asset is not given, and so will default to 8 m/s. Note the
296     * association with the previously given tidal resource series by way of the wind
297     * resource key.
298     *
299     * For more details on the various attributes of WindInputs, refer to the PGMcpp
300     * manual. For instance, note that no economic inputs are given; in this
301     * example, the default values apply.
302     *
303     * The fourth block defines and adds a wave energy converter to the Model object.
304     * In this example, the installed wave capacity is set to 100 kW. Note the
305     * association with the previously given wave resource series by way of the wave
306     * resource key.
307     *
308     * For more details on the various attributes of WaveInputs, refer to the PGMcpp
309     * manual. For instance, note that no economic inputs are given; in this
310     * example, the default values apply.
311     */
312
313     // 5.1. add 1 x 250 kW solar PV array
314     SolarInputs solar_inputs;
315
316     solar_inputs.renewable_inputs.production_inputs.capacity_kW = 250;
317     solar_inputs.resource_key = solar_resource_key;
318
319     model.addSolar(solar_inputs);
320
321     // 5.2. add 1 x 120 kW tidal turbine
322     TidalInputs tidal_inputs;
323
324     tidal_inputs.renewable_inputs.production_inputs.capacity_kW = 120;
325     tidal_inputs.design_speed_ms = 2.5;
326     tidal_inputs.resource_key = tidal_resource_key;
327
328     model.addTidal(tidal_inputs);
329
330     // 5.3. add 1 x 150 kW wind turbine
331     WindInputs wind_inputs;
332
333     wind_inputs.renewable_inputs.production_inputs.capacity_kW = 150;
334     wind_inputs.resource_key = wind_resource_key;

```

```

335
336     model.addWind(wind_inputs);
337
338     // 5.4. add 1 x 100 kW wave energy converter
339     WaveInputs wave_inputs;
340
341     wave_inputs.renewable_inputs.production_inputs.capacity_kW = 100;
342     wave_inputs.resource_key = wave_resource_key;
343
344     model.addWave(wave_inputs);
345
346
347
348     /*
349     * 6. add LiIon object to Model
350     *
351     * This block defines and adds a lithium ion battery energy storage system to the
352     * Model object.
353     *
354     * In this example, a battery energy storage system with a 500 kW power capacity
355     * and a 1050 kWh energy capacity (which represents about four hours of mean load
356     * autonomy) is defined.
357     *
358     * For more details on the various attributes of LiIonInputs, refer to the PGMcpp
359     * manual. For instance, note that no economic inputs are given; in this
360     * example, the default values apply.
361     */
362
363     // 6.1. add 1 x (500 kW, ) lithium ion battery energy storage system
364     LiIonInputs liion_inputs;
365
366     liion_inputs.storage_inputs.power_capacity_kW = 500;
367     liion_inputs.storage_inputs.energy_capacity_kWh = 1050;
368
369     model.addLiIon(liion_inputs);
370
371
372
373     /*
374     * 7. run and write results
375     *
376     * This block runs the model and then writes results to the given output path
377     * (either relative or absolute). Note that the writeResults() will create the
378     * last directory on the given path, but not any in-between directories, so be
379     * sure those exist before calling out to this method.
380     */
381
382     model.run();
383
384     model.writeResults("projects/example_cpp");
385
386     return 0;
387 } /* main() */

```

5.21 pybindings/PYBIND11_PGM.cpp File Reference

Bindings file for PGMcpp.

```

#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include "../header/Model.h"
#include "snippets/PYBIND11_Controller.cpp"
#include "snippets/PYBIND11_ElectricalLoad.cpp"
#include "snippets/PYBIND11_Interpolator.cpp"
#include "snippets/PYBIND11_Model.cpp"
#include "snippets/PYBIND11_Resources.cpp"
#include "snippets/Production/PYBIND11_Production.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"
#include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
#include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
#include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
#include "snippets/Production/Renewable/PYBIND11_Solar.cpp"

```

```
#include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
#include "snippets/Storage/PYBIND11_Storage.cpp"
#include "snippets/Storage/PYBIND11_LiIon.cpp"
Include dependency graph for PYBIND11_PGM.cpp:
```



Functions

- [PYBIND11_MODULE](#) (PGMcpp, m)

5.21.1 Detailed Description

Bindings file for PGMcpp.

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for PGMcpp. Only public attributes/methods are bound!

5.21.2 Function Documentation

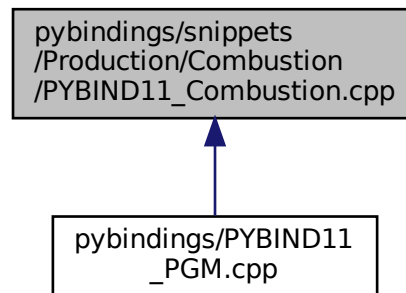
5.21.2.1 PYBIND11_MODULE()

```
PYBIND11_MODULE (
    PGMcpp ,
    m )
{
56
57
58     #include "snippets/PYBIND11_Controller.cpp"
59     #include "snippets/PYBIND11_ElectricalLoad.cpp"
60     #include "snippets/PYBIND11_Interpolator.cpp"
61     #include "snippets/PYBIND11_Model.cpp"
62     #include "snippets/PYBIND11_Resources.cpp"
63
64     #include "snippets/Production/PYBIND11_Production.cpp"
65
66     #include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
67     #include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"
68
69     #include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
70     #include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
71
72     #include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
73     #include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
74     #include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
75     #include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
76     #include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
77
78     #include "snippets/Storage/PYBIND11_Storage.cpp"
79     #include "snippets/Storage/PYBIND11_LiIon.cpp"
80
81 } /* PYBIND11_MODULE() */
```

5.22 pybindings/snippets/Production/Combustion/PYBIND11_↵ Combustion.cpp File Reference

Bindings file for the [Combustion](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [CombustionType::DIESEL](#) [value](#) ("N_COMBUSTION_TYPES", [CombustionType::N_COMBUSTION_TYPES](#))
- [FuelMode::FUEL_MODE_LINEAR](#) [value](#) ("FUEL_MODE_LOOKUP", [FuelMode::FUEL_MODE_LOOKUP](#))
.value("N_FUEL_MODES")
- [&CombustionInputs::production_inputs](#) [def_readwrite](#) ("fuel_mode", [&CombustionInputs::fuel_mode](#)) [.def_readwrite](#) ("nominal_fuel_escalation_annual"
- [&CombustionInputs::production_inputs](#) [&CombustionInputs::nominal_fuel_escalation_annual](#) [def_readwrite](#) ("cycle_charging_setpoint", [&CombustionInputs::cycle_charging_setpoint](#)) [.def_readwrite](#) ("path_2_fuel_interp_data"
- [&CombustionInputs::production_inputs](#) [&CombustionInputs::nominal_fuel_escalation_annual](#) [&CombustionInputs::path_2_fuel](#) [def](#) (pybind11::init())
- [&Emissions::CO2_kg](#) [def_readwrite](#) ("CO_kg", [&Emissions::CO_kg](#)) [.def_readwrite](#) ("NOx_kg"
- [&Emissions::CO2_kg](#) [&Emissions::NOx_kg](#) [def_readwrite](#) ("SOx_kg", [&Emissions::SOx_kg](#)) [.def_readwrite](#) ("CH4_kg"

Variables

- [&Emissions::CO2_kg](#) [&Emissions::NOx_kg](#) [&Emissions::CH4_kg](#) [def_readwrite](#) ("PM_kg", [&Emissions::PM_kg](#)) [.def](#) (pybind11 [&Combustion::type](#) [def_readwrite](#) ("fuel_mode", [&Combustion::fuel_mode](#)) [.def_readwrite](#) ("total_emissions"

5.22.1 Detailed Description

Bindings file for the [Combustion](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Combustion](#) class. Only public attributes/methods are bound!

5.22.2 Function Documentation

5.22.2.1 def()

```
&InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D::
&InterpolatorStruct2D::z_matrix def (
    pybind11::init() )
```

5.22.2.2 def_readwrite() [1/4]

```
& Emissions::CO2_kg def_readwrite (
    "CO_kg" ,
    &Emissions::CO_kg )
```

5.22.2.3 def_readwrite() [2/4]

```
& CombustionInputs::production_inputs & CombustionInputs::nominal_fuel_escalation_annual def↵
_readwrite (
    "cycle_charging_setpoint" ,
    &CombustionInputs::cycle_charging_setpoint )
```

5.22.2.4 def_readwrite() [3/4]

```
& CombustionInputs::production_inputs def_readwrite (
    "fuel_mode" ,
    &CombustionInputs::fuel_mode )
```

5.22.2.5 def_readwrite() [4/4]

```
& Emissions::CO2_kg & Emissions::NOx_kg def_readwrite (
    "SOx_kg" ,
    &Emissions::SOx_kg )
```

5.22.2.6 value() [1/2]

```
FuelMode::FUEL_MODE_LINEAR value (
    "FUEL_MODE_LOOKUP" ,
    FuelMode::FUEL_MODE_LOOKUP )
```

5.22.2.7 value() [2/2]

```
CombustionType::DIESEL value (
    "N_COMBUSTION_TYPES" ,
    CombustionType::N_COMBUSTION_TYPES )
```

5.22.3 Variable Documentation

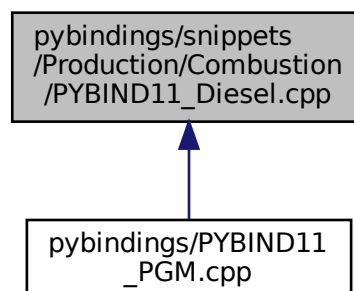
5.22.3.1 def_readwrite

```
&StorageInputs::print_flag &StorageInputs::power_capacity_kW &StorageInputs::nominal_inflation_annual
def_readwrite (
    "fuel_mode" ,
    &Combustion::fuel_mode )
```

5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference

Bindings file for the [Diesel](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- `&DieselInputs::combustion_inputs def_readwrite` ("replace_running_hrs", &DieselInputs::replace_running_hrs) .def_readwrite("capital_cost"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost def_readwrite` ("operation_maintenance_cost_kWh", &DieselInputs::operation_maintenance_cost_kWh) .def_readwrite("fuel_cost_L"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L def_readwrite` ("minimum_load_ratio", &DieselInputs::minimum_load_ratio) .def_readwrite("minimum_runtime_hrs"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs def_readwrite` ("linear_fuel_slope_LkWh", &DieselInputs::linear_fuel_slope_LkWh) .def_readwrite("linear_fuel_intercept_LkWh"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs &DieselInputs::linear_fuel_intercept_LkWh def_readwrite` ("CO2_emissions_intensity_kgL", &DieselInputs::CO2_emissions_intensity_kgL) .def_readwrite("CO2_emissions_intensity_kgL"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO2_emissions_intensity_kgL def_readwrite` ("NOx_emissions_intensity_kgL", &DieselInputs::NOx_emissions_intensity_kgL) .def_readwrite("SOx_emissions_intensity_kgL"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO2_emissions_intensity_kgL &DieselInputs::SOx_emissions_intensity_kgL def_readwrite` ("CH4_emissions_intensity_kgL", &DieselInputs::CH4_emissions_intensity_kgL) .def_readwrite("PM_emissions_intensity_kgL"
- `&DieselInputs::combustion_inputs &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO2_emissions_intensity_kgL &DieselInputs::SOx_emissions_intensity_kgL &DieselInputs::PM_emissions_intensity_kgL def` (pybind11::init())
- `&Diesel::minimum_load_ratio def_readwrite` ("minimum_runtime_hrs", &Diesel::minimum_runtime_hrs) .def_readwrite("time_since_last_start_hrs"

5.23.1 Detailed Description

Bindings file for the [Diesel](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Diesel](#) class. Only public attributes/methods are bound!

5.23.2 Function Documentation

5.23.2.1 def()

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO2_emissions_inte
& DieselInputs::SOx_emissions_intensity_kgL & DieselInputs::PM_emissions_intensity_kgL def (
    pybind11::init() )
```

5.23.2.2 def_readwrite() [1/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
& DieselInputs::SOx_emissions_intensity_kgL def_readwrite (
    "CH4_emissions_intensity_kgL" ,
    &DieselInputs::CH4_emissions_intensity_kgL )
```

5.23.2.3 def_readwrite() [2/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh def_readwrite (
    "CO2_emissions_intensity_kgL" ,
    &DieselInputs::CO2_emissions_intensity_kgL )
```

5.23.2.4 def_readwrite() [3/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs def_readwrite (
    "linear_fuel_slope_LkWh" ,
    &DieselInputs::linear_fuel_slope_LkWh )
```

5.23.2.5 def_readwrite() [4/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L
def_readwrite (
    "minimum_load_ratio" ,
    &DieselInputs::minimum_load_ratio )
```

5.23.2.6 def_readwrite() [5/8]

```
& Diesel::minimum_load_ratio def_readwrite (
    "minimum_runtime_hrs" ,
    &Diesel::minimum_runtime_hrs )
```

5.23.2.7 def_readwrite() [6/8]

```

& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
def_readwrite (
    "NOx_emissions_intensity_kgL" ,
    & DieselInputs::NOx_emissions_intensity_kgL )

```

5.23.2.8 def_readwrite() [7/8]

```

& DieselInputs::combustion_inputs & DieselInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    & DieselInputs::operation_maintenance_cost_kWh )

```

5.23.2.9 def_readwrite() [8/8]

```

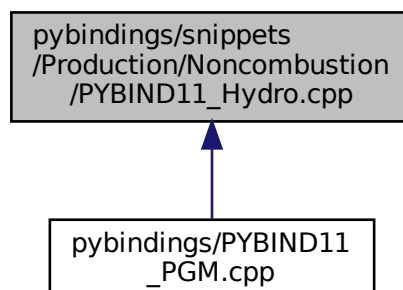
& DieselInputs::combustion_inputs def_readwrite (
    "replace_running_hrs" ,
    & DieselInputs::replace_running_hrs )

```

5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp File Reference

Bindings file for the [Hydro](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- `HydroTurbineType::HYDRO_TURBINE_PELTON` `value` ("HYDRO_TURBINE_FRANCIS", `HydroTurbineType::HYDRO_TURBINE_FRANCIS`) `.value`("HYDRO_TURBINE_KAPLAN")
- `HydroTurbineType::HYDRO_TURBINE_PELTON` `HydroTurbineType::HYDRO_TURBINE_KAPLAN` `value` ("N_HYDRO_TURBINES", `HydroTurbineType::N_HYDRO_TURBINES`)
- `&HydroInputs::noncombustion_inputs` `def_readwrite` ("resource_key", `&HydroInputs::resource_key`) `.def_readwrite`("capital_cost")
- `&HydroInputs::noncombustion_inputs` `&HydroInputs::capital_cost` `def_readwrite` ("operation_maintenance_cost_kWh", `&HydroInputs::operation_maintenance_cost_kWh`) `.def_readwrite`("fluid_density_kgm3")
- `&HydroInputs::noncombustion_inputs` `&HydroInputs::capital_cost` `&HydroInputs::fluid_density_kgm3` `def_readwrite` ("net_head_m", `&HydroInputs::net_head_m`) `.def_readwrite`("reservoir_capacity_m3")
- `&HydroInputs::noncombustion_inputs` `&HydroInputs::capital_cost` `&HydroInputs::fluid_density_kgm3` `&HydroInputs::reservoir_capacity_m3` `def_readwrite` ("init_reservoir_state", `&HydroInputs::init_reservoir_state`) `.def_readwrite`("turbine_type")
- `&HydroInputs::noncombustion_inputs` `&HydroInputs::capital_cost` `&HydroInputs::fluid_density_kgm3` `&HydroInputs::reservoir_capacity_m3` `&HydroInputs::turbine_type` `def` (`pybind11::init()`)
- `&Hydro::turbine_type` `def_readwrite` ("fluid_density_kgm3", `&Hydro::fluid_density_kgm3`) `.def_readwrite`("net_head_m")
- `&Hydro::turbine_type` `&Hydro::net_head_m` `def_readwrite` ("reservoir_capacity_m3", `&Hydro::reservoir_capacity_m3`) `.def_readwrite`("init_reservoir_state")
- `&Hydro::turbine_type` `&Hydro::net_head_m` `&Hydro::init_reservoir_state` `def_readwrite` ("stored_volume_m3", `&Hydro::stored_volume_m3`) `.def_readwrite`("minimum_power_kW")
- `&Hydro::turbine_type` `&Hydro::net_head_m` `&Hydro::init_reservoir_state` `&Hydro::minimum_power_kW` `def_readwrite` ("minimum_flow_m3hr", `&Hydro::minimum_flow_m3hr`) `.def_readwrite`("maximum_flow_m3hr")
- `&Hydro::turbine_type` `&Hydro::net_head_m` `&Hydro::init_reservoir_state` `&Hydro::minimum_power_kW` `&Hydro::maximum_flow_m3hr` `def_readwrite` ("turbine_flow_vec_m3hr", `&Hydro::turbine_flow_vec_m3hr`) `.def_readwrite`("spill_rate_vec_m3hr")

5.24.1 Detailed Description

Bindings file for the `Hydro` class. Intended to be #include'd in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs `pybind11` how to build Python bindings for the `Hydro` class. Only public attributes/methods are bound!

5.24.2 Function Documentation

5.24.2.1 `def()`

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 & HydroInputs::turbine_type def (
    pybind11::init() )
```

5.24.2.2 def_readwrite() [1/9]

```
& Hydro::turbine_type def_readwrite (
    "fluid_density_kgm3" ,
    &Hydro::fluid_density_kgm3 )
```

5.24.2.3 def_readwrite() [2/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 def_readwrite (
    "init_reservoir_state" ,
    &HydroInputs::init_reservoir_state )
```

5.24.2.4 def_readwrite() [3/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
def_readwrite (
    "minimum_flow_m3hr" ,
    &Hydro::minimum_flow_m3hr )
```

5.24.2.5 def_readwrite() [4/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
def_readwrite (
    "net_head_m" ,
    &HydroInputs::net_head_m )
```

5.24.2.6 def_readwrite() [5/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    &HydroInputs::operation_maintenance_cost_kWh )
```

5.24.2.7 def_readwrite() [6/9]

```
& Hydro::turbine_type & Hydro::net_head_m def_readwrite (
    "reservoir_capacity_m3" ,
    &Hydro::reservoir_capacity_m3 )
```

5.24.2.8 def_readwrite() [7/9]

```
& HydroInputs::noncombustion_inputs def_readwrite (
    "resource_key" ,
    &HydroInputs::resource_key )
```

5.24.2.9 def_readwrite() [8/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state def_readwrite (
    "stored_volume_m3" ,
    &Hydro::stored_volume_m3 )
```

5.24.2.10 def_readwrite() [9/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
& Hydro::maximum_flow_m3hr def_readwrite (
    "turbine_flow_vec_m3hr" ,
    &Hydro::turbine_flow_vec_m3hr )
```

5.24.2.11 value() [1/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON value (
    "HYDRO_TURBINE_FRANCIS" ,
    HydroTurbineType::HYDRO_TURBINE_FRANCIS )
```

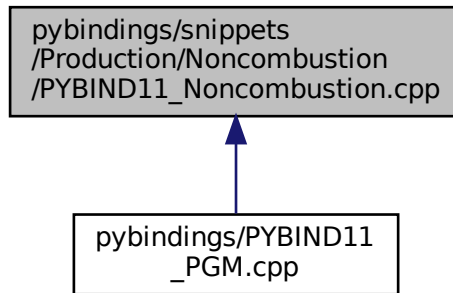
5.24.2.12 value() [2/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON HydroTurbineType::HYDRO_TURBINE_KAPLAN value (
    "N_HYDRO_TURBINES" ,
    HydroTurbineType::N_HYDRO_TURBINES )
```


5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp File Reference

Bindings file for the [Noncombustion](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [NoncombustionType::HYDRO](#) value ("N_NONCOMBUSTION_TYPES", [NoncombustionType::N_NONCOMBUSTION_TYPES](#))
- &[NoncombustionInputs::production_inputs](#) def (pybind11::init())

5.25.1 Detailed Description

Bindings file for the [Noncombustion](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Noncombustion](#) class. Only public attributes/methods are bound!

5.25.2 Function Documentation

5.25.2.1 def()

```
& NoncombustionInputs::production\_inputs def (
    pybind11::init() )
```

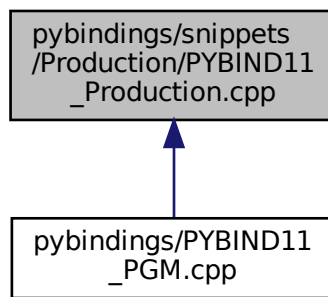
5.25.2.2 value()

```
NoncombustionType::HYDRO value (
    "N_NONCOMBUSTION_TYPES" ,
    NoncombustionType::N_NONCOMBUSTION_TYPES )
```

5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference

Bindings file for the [Production](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [&ProductionInputs::print_flag def_readwrite](#) ("is_sunk", &ProductionInputs::is_sunk) .def_readwrite("capacity_kW"
- [&ProductionInputs::print_flag &ProductionInputs::capacity_kW def_readwrite](#) ("nominal_inflation_annual", &ProductionInputs::nominal_inflation_annual) .def_readwrite("nominal_discount_annual"
- [&ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual def_readwrite](#) ("replace_running_hrs", &ProductionInputs::replace_running_hrs) .def_readwrite("path_2_normalized_production_time_series"
- [&ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual &ProductionInputs::path_2_normalized_production_time_series def](#) (pybind11::init())
- [&Production::interpolator def_readwrite](#) ("print_flag", &Production::print_flag) .def_readwrite("is_running"
- [&Production::interpolator &Production::is_running def_readwrite](#) ("is_sunk", &Production::is_sunk) .def_readwrite("normalized_production_series_given"
- [&Production::interpolator &Production::is_running &Production::normalized_production_series_given def_readwrite](#) ("n_points", &Production::n_points) .def_readwrite("n_starts"
- [&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts def_readwrite](#) ("n_replacements", &Production::n_replacements) .def_readwrite("n_years"
- [&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years def_readwrite](#) ("running_hours", &Production::running_hours) .def_readwrite("replace_running_hrs"

- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs def_readwrite ("capacity_kW", &Production::capacity_kW) .def_readwrite("nominal_inflation_annual"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual def_readwrite ("nominal_discount_annual", &Production::nominal_discount_annual) .def_readwrite("real_discount_annual"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual def_readwrite ("capital_cost", &Production::capital_cost) .def_readwrite("operation_maintenance_cost_kWh"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh def_readwrite ("net_present_cost", &Production::net_present_cost) .def_readwrite("total_dispatch_kWh"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh def_readwrite ("levellized_cost_of_energy_kWh", &Production::levellized_cost_of_energy_kWh) .def_readwrite("type_str"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str def_readwrite ("path_2_normalized_production_time_series", &Production::path_2_normalized_production_time_series) .def_readwrite("is_running_vec"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec def_readwrite ("normalized_production_vec", &Production::normalized_production_vec) .def_readwrite("production_vec_kW"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW def_readwrite ("dispatch_vec_kW", &Production::dispatch_vec_kW) .def_readwrite("storage_vec_kW"`
- `&Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW &Production::storage_vec_kW def_readwrite ("curtailment_vec_kW", &Production::curtailment_vec_kW) .def_readwrite("capital_cost_vec"`

5.26.1 Detailed Description

Bindings file for the [Production](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Production](#) class. Only public attributes/methods are bound!

5.26.2 Function Documentation

5.26.2.1 def()

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
& ProductionInputs::path_2_normalized_production_time_series def (
    pybind11::init() )
```

5.26.2.2 def_readwrite() [1/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs def_readwrite (
    "capacity_kW" ,
    &Production::capacity_kW )
```

5.26.2.3 def_readwrite() [2/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual def_readwrite (
    "capital_cost" ,
    &Production::capital_cost )
```

5.26.2.4 def_readwrite() [3/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW & Production::storage_vec_k
def_readwrite (
    "curtailment_vec_kW" ,
    &Production::curtailment_vec_kW )
```

5.26.2.5 def_readwrite() [4/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW def_readwrite (
    "dispatch_vec_kW" ,
    &Production::dispatch_vec_kW )
```

5.26.2.6 def_readwrite() [5/17]

```
& Production::interpolator & Production::is_running def_readwrite (
    "is_sunk" ,
    &Production::is_sunk )
```

5.26.2.7 def_readwrite() [6/17]

```
& ProductionInputs::print_flag def_readwrite (
    "is_sunk" ,
    &ProductionInputs::is_sunk )
```

5.26.2.8 def_readwrite() [7/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
def_readwrite (
    "levellized_cost_of_energy_kWh" ,
    &Production::levellized_cost_of_energy_kWh )
```

5.26.2.9 def_readwrite() [8/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
def_readwrite (
    "n_points" ,
    &Production::n_points )
```

5.26.2.10 def_readwrite() [9/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts def_readwrite (
    "n_replacements" ,
    &Production::n_replacements )
```

5.26.2.11 def_readwrite() [10/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh def_readwrite
(
    "net_present_cost" ,
    &Production::net_present_cost )
```

5.26.2.12 def_readwrite() [11/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
def_readwrite (
    "nominal_discount_annual" ,
    &Production::nominal_discount_annual )
```

5.26.2.13 def_readwrite() [12/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW def_readwrite (
    "nominal_inflation_annual" ,
    &ProductionInputs::nominal_inflation_annual )
```

5.26.2.14 def_readwrite() [13/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec def_readwrite (
    "normalized_production_vec" ,
    &Production::normalized_production_vec )
```

5.26.2.15 def_readwrite() [14/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflation
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str def_readwrite (
    "path_2_normalized_production_time_series" ,
    &Production::path_2_normalized_production_time_series )
```

5.26.2.16 def_readwrite() [15/17]

```
& Production::interpolator def_readwrite (
    "print_flag" ,
    &Production::print_flag )
```

5.26.2.17 def_readwrite() [16/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
def_readwrite (
    "replace_running_hrs" ,
    &ProductionInputs::replace_running_hrs )
```

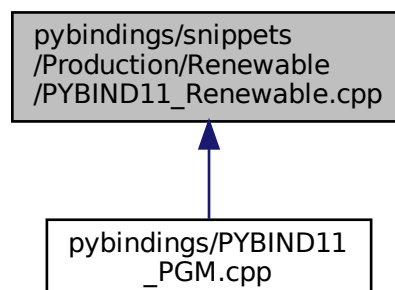
5.26.2.18 def_readwrite() [17/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years def_readwrite (
    "running_hours" ,
    &Production::running_hours )
```

5.27 pybindings/snippets/Production/Renewable/PYBIND11_↔ Renewable.cpp File Reference

Bindings file for the [Renewable](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- `RenewableType::SOLAR value ("TIDAL", RenewableType::TIDAL) .value("WAVE"`
- `RenewableType::SOLAR RenewableType::WAVE value ("WIND", RenewableType::WIND) .value("N_↔
RENEWABLE_TYPES"`
- `&RenewableInputs::production_inputs def (pybind11::init())`

5.27.1 Detailed Description

Bindings file for the `Renewable` class. Intended to be #include'd in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the `Renewable` class. Only public attributes/methods are bound!

5.27.2 Function Documentation

5.27.2.1 `def()`

```
& RenewableInputs::production_inputs def (
    pybind11::init() )
```

5.27.2.2 `value()` [1/2]

```
RenewableType::SOLAR value (
    "TIDAL" ,
    RenewableType::TIDAL )
```

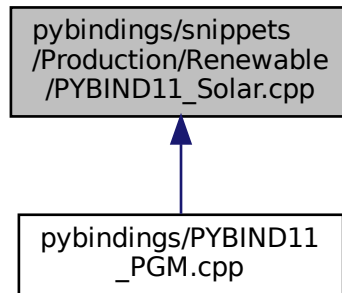
5.27.2.3 `value()` [2/2]

```
RenewableType::SOLAR RenewableType::WAVE value (
    "WIND" ,
    RenewableType::WIND )
```


5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference

Bindings file for the [Solar](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [&SolarInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", &SolarInputs::resource_key) .def_↔
readwrite("capital_cost"
- [&SolarInputs::renewable_inputs](#) [&SolarInputs::capital_cost](#) [def_readwrite](#) ("operation_maintenance_cost_↔
kWh", &SolarInputs::operation_maintenance_cost_kWh) .def_readwrite("derating"
- [&SolarInputs::renewable_inputs](#) [&SolarInputs::capital_cost](#) [&SolarInputs::derating](#) [def](#) (pybind11::init())

5.28.1 Detailed Description

Bindings file for the [Solar](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Solar](#) class. Only public attributes/methods are bound!

5.28.2 Function Documentation

5.28.2.1 def()

```

& SolarInputs::renewable\_inputs & SolarInputs::capital\_cost & SolarInputs::derating def (
    pybind11::init() )
  
```

5.28.2.2 `def_readwrite()` [1/2]

```
& SolarInputs::renewable_inputs & SolarInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    &SolarInputs::operation_maintenance_cost_kWh )
```

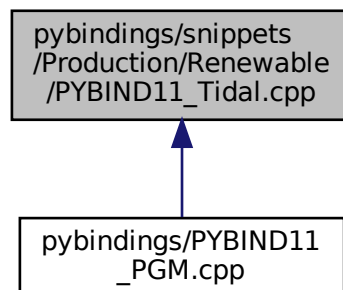
5.28.2.3 `def_readwrite()` [2/2]

```
& SolarInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &SolarInputs::resource_key )
```

5.29 `pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp` File Reference

Bindings file for the [Tidal](#) class. Intended to be `#include'd` in [PYBIND11_PGM.cpp](#).

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



Functions

- `TidalPowerProductionModel::TIDAL_POWER_CUBIC` `value` ("TIDAL_POWER_EXPONENTIAL", `TidalPowerProductionModel::TIDAL_POWER_EXPONENTIAL`) `.value`("TIDAL_POWER_LOOKUP"
- `TidalPowerProductionModel::TIDAL_POWER_CUBIC` `TidalPowerProductionModel::TIDAL_POWER_LOOKUP` `value` ("N_TIDAL_POWER_PRODUCTION_MODELS", `TidalPowerProductionModel::N_TIDAL_POWER_PRODUCTION_MODELS`)
- `&TidalInputs::renewable_inputs` `def_readwrite` ("resource_key", `&TidalInputs::resource_key`) `.def_readwrite`("capital_cost"
- `&TidalInputs::renewable_inputs` `&TidalInputs::capital_cost` `def_readwrite` ("operation_maintenance_cost_kWh", `&TidalInputs::operation_maintenance_cost_kWh`) `.def_readwrite`("design_speed_ms"

Variables

- `&TidalInputs::renewable_inputs &TidalInputs::capital_cost &TidalInputs::design_speed_ms` `def_readwrite("power↵_model", &TidalInputs::power_model) .def(pybind11 &Tidal::design_speed_ms def_readwrite ("power_↵model", &Tidal::power_model) .def_readwrite("power_model_string"`

5.29.1 Detailed Description

Bindings file for the [Tidal](#) class. Intended to be #include'd in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Tidal](#) class. Only public attributes/methods are bound!

5.29.2 Function Documentation

5.29.2.1 `def_readwrite()` [1/2]

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    &TidalInputs::operation_maintenance_cost_kWh )
```

5.29.2.2 `def_readwrite()` [2/2]

```
& TidalInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &TidalInputs::resource_key )
```

5.29.2.3 `value()` [1/2]

```
TidalPowerProductionModel::TIDAL_POWER_CUBIC TidalPowerProductionModel::TIDAL_POWER_LOOKUP
value (
    "N_TIDAL_POWER_PRODUCTION_MODELS" ,
    TidalPowerProductionModel::N_TIDAL_POWER_PRODUCTION_MODELS )
```

5.29.2.4 `value()` [2/2]

```
TidalPowerProductionModel::TIDAL_POWER_CUBIC value (
    "TIDAL_POWER_EXPONENTIAL" ,
    TidalPowerProductionModel::TIDAL_POWER_EXPONENTIAL )
```

5.29.3 Variable Documentation

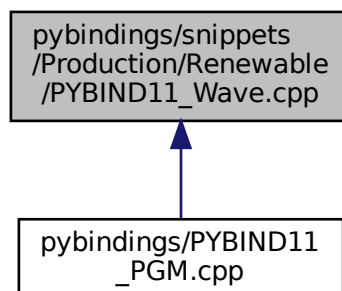
5.29.3.1 def_readwrite

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost & TidalInputs::design_speed_ms
def_readwrite ("power_model", &TidalInputs::power_model) .def(pybind11 & Tidal::design_speed_ms
def_readwrite("power_model", &Tidal::power_model) .def_readwrite("power_model_string" (
    "power_model" ,
    &Tidal::power_model )
```

5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference

Bindings file for the [Wave](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [WavePowerProductionModel::WAVE_POWER_GAUSSIAN](#) [value](#) ("WAVE_POWER_PARABOLOID", [WavePowerProductionModel::WAVE_POWER_PARABOLOID](#)) [.value](#)("WAVE_POWER_LOOKUP"
- [WavePowerProductionModel::WAVE_POWER_GAUSSIAN](#) [WavePowerProductionModel::WAVE_POWER_LOOKUP](#) [value](#) ("N_WAVE_POWER_PRODUCTION_MODELS", [WavePowerProductionModel::N_WAVE_POWER_PRODUCTION_MODELS](#))
- [&WaveInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", [&WaveInputs::resource_key](#)) [.def_readwrite](#)("capital_cost"
- [&WaveInputs::renewable_inputs](#) [&WaveInputs::capital_cost](#) [def_readwrite](#) ("operation_maintenance_cost_kWh", [&WaveInputs::operation_maintenance_cost_kWh](#)) [.def_readwrite](#)("design_significant_wave_height_m"
- [&WaveInputs::renewable_inputs](#) [&WaveInputs::capital_cost](#) [&WaveInputs::design_significant_wave_height_m](#) [def_readwrite](#) ("design_energy_period_s", [&WaveInputs::design_energy_period_s](#)) [.def_readwrite](#)("power_model"

Variables

- `&WaveInputs::renewable_inputs` `&WaveInputs::capital_cost` `&WaveInputs::design_significant_wave_height_m` `&WaveInputs::power_model` `def_readwrite("path_2_normalized_performance_matrix", &WaveInputs::path_2_normalized_performance_matrix)` `.def(pybind11 &Wave::design_significant_wave_height_m` `def_readwrite ("design_energy_period_s", &Wave::design_energy_period_s)` `.def_readwrite("power_model"`

5.30.1 Detailed Description

Bindings file for the `Wave` class. Intended to be `#include'd` in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the `Wave` class. Only public attributes/methods are bound!

5.30.2 Function Documentation

5.30.2.1 `def_readwrite()` [1/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
def_readwrite (
    "design_energy_period_s" ,
    &WaveInputs::design_energy_period_s )
```

5.30.2.2 `def_readwrite()` [2/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    &WaveInputs::operation_maintenance_cost_kWh )
```

5.30.2.3 `def_readwrite()` [3/3]

```
& WaveInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &WaveInputs::resource_key )
```

5.30.2.4 value() [1/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN WavePowerProductionModel::WAVE_POWER_LOOKUP
value (
    "N_WAVE_POWER_PRODUCTION_MODELS" ,
    WavePowerProductionModel::N_WAVE_POWER_PRODUCTION_MODELS )
```

5.30.2.5 value() [2/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN value (
    "WAVE_POWER_PARABOLOID" ,
    WavePowerProductionModel::WAVE_POWER_PARABOLOID )
```

5.30.3 Variable Documentation

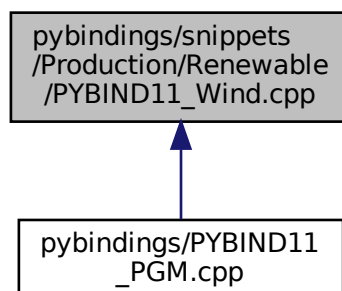
5.30.3.1 def_readwrite

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
& WaveInputs::power_model def_readwrite ( "path_2_normalized_performance_matrix", &Wave←
Inputs::path_2_normalized_performance_matrix ) .def(pybind11 & Wave::design_significant_wave_height_m
def_readwrite("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power←
_model" (
    "design_energy_period_s" ,
    &Wave::design_energy_period_s )
```

5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference

Bindings file for the [Wind](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- `WindPowerProductionModel::WIND_POWER_EXPONENTIAL` `value` ("WIND_POWER_LOOKUP", `WindPowerProductionModel::WIND_POWER_LOOKUP`) `.value`("N_WIND_POWER_PRODUCTION_MODELS"
- `&WindInputs::renewable_inputs` `def_readwrite` ("resource_key", `&WindInputs::resource_key`) `.def_readwrite`("capital_cost"
- `&WindInputs::renewable_inputs` `&WindInputs::capital_cost` `def_readwrite` ("operation_maintenance_cost_kWh", `&WindInputs::operation_maintenance_cost_kWh`) `.def_readwrite`("design_speed_ms"

Variables

- `&WindInputs::renewable_inputs` `&WindInputs::capital_cost` `&WindInputs::design_speed_ms` `def_readwrite`("power_model", `&WindInputs::power_model`) `.def`(`pybind11 &Wind::design_speed_ms` `def_readwrite`("power_model", `&Wind::power_model`) `.def_readwrite`("power_model_string"

5.31.1 Detailed Description

Bindings file for the `Wind` class. Intended to be `#include'd` in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs `pybind11` how to build Python bindings for the `Wind` class. Only public attributes/methods are bound!

5.31.2 Function Documentation

5.31.2.1 `def_readwrite()` [1/2]

```
& WindInputs::renewable_inputs & WindInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    &WindInputs::operation_maintenance_cost_kWh )
```

5.31.2.2 `def_readwrite()` [2/2]

```
& WindInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &WindInputs::resource_key )
```

5.31.2.3 `value()`

```
WindPowerProductionModel::WIND_POWER_EXPONENTIAL value (
    "WIND_POWER_LOOKUP" ,
    WindPowerProductionModel::WIND_POWER_LOOKUP )
```

5.31.3 Variable Documentation

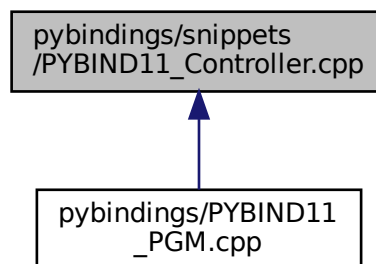
5.31.3.1 def_readwrite

```
& WindInputs::renewable_inputs & WindInputs::capital_cost & WindInputs::design_speed_ms def↔
_readwrite ("power_model", &WindInputs::power_model) .def(pybind11 & Wind::design_speed_ms
def_readwrite("power_model", &Wind::power_model) .def_readwrite("power_model_string" (
    "power_model" ,
    &Wind::power_model )
```

5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference

Bindings file for the [Controller](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [ControlMode::LOAD_FOLLOWING](#) value ("CYCLE_CHARGING", ControlMode::CYCLE_CHARGING) .value("N_CONTROL_MODES"
- [&Controller::control_mode](#) def_readwrite ("control_string", &Controller::control_string) .def_readwrite("net↔_load_vec_kW"
- [&Controller::control_mode](#) &Controller::net_load_vec_kW def_readwrite ("missed_load_vec_kW", &Controller↔::missed_load_vec_kW) .def_readwrite("combustion_map"
- [&Controller::control_mode](#) &Controller::net_load_vec_kW &Controller::combustion_map def (pybind11↔::init<>()) .def("setControlMode"
- [&Controller::control_mode](#) &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode↔ def ("init", &Controller::init) .def("applyDispatchControl"
- [&Controller::control_mode](#) &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode↔ &Controller::applyDispatchControl def ("clear", &Controller::clear)

5.32.1 Detailed Description

Bindings file for the [Controller](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Controller](#) class. Only public attributes/methods are bound!

5.32.2 Function Documentation

5.32.2.1 `def()` [1/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
& Controller::applyDispatchControl def (
    "clear" ,
    &Controller::clear )
```

5.32.2.2 `def()` [2/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
def (
    "init" ,
    &Controller::init )
```

5.32.2.3 `def()` [3/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map def (
    pybind11::init<> () )
```

5.32.2.4 `def_readwrite()` [1/2]

```
& Controller::control_mode def_readwrite (
    "control_string" ,
    &Controller::control_string )
```

5.32.2.5 def_readwrite() [2/2]

```
& Controller::control_mode & Controller::net_load_vec_kW def_readwrite (
    "missed_load_vec_kW" ,
    &Controller::missed_load_vec_kW )
```

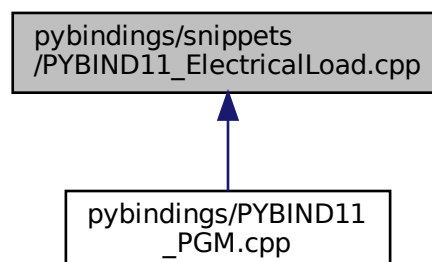
5.32.2.6 value()

```
ControlMode::LOAD_FOLLOWING value (
    "CYCLE_CHARGING" ,
    ControlMode::CYCLE_CHARGING )
```

5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference

Bindings file for the [ElectricalLoad](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [&ElectricalLoad::n_points](#) [def_readwrite](#) ("n_years", &ElectricalLoad::n_years) .def_readwrite("min_load_kW"
- [&ElectricalLoad::n_points](#) [&ElectricalLoad::min_load_kW](#) [def_readwrite](#) ("mean_load_kW", &ElectricalLoad::mean_load_kW) .def_readwrite("max_load_kW"
- [&ElectricalLoad::n_points](#) [&ElectricalLoad::min_load_kW](#) [&ElectricalLoad::max_load_kW](#) [def_readwrite](#) ("path_2_electrical_load_time_series", &ElectricalLoad::path_2_electrical_load_time_series) .def_readwrite("time_vec_hrs"
- [&ElectricalLoad::n_points](#) [&ElectricalLoad::min_load_kW](#) [&ElectricalLoad::max_load_kW](#) [&ElectricalLoad::time_vec_hrs](#) [def_readwrite](#) ("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs) .def_readwrite("load_vec_kW"

5.33.1 Detailed Description

Bindings file for the [ElectricalLoad](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [ElectricalLoad](#) class. Only public attributes/methods are bound!

5.33.2 Function Documentation

5.33.2.1 `def_readwrite()` [1/4]

```
& ElectricalLoad::n\_points & ElectricalLoad::min\_load\_kW & ElectricalLoad::max\_load\_kW & ElectricalLoad::time\_
def_readwrite (
    "dt_vec_hrs" ,
    &ElectricalLoad::dt\_vec\_hrs )
```

5.33.2.2 `def_readwrite()` [2/4]

```
& ElectricalLoad::n\_points & ElectricalLoad::min\_load\_kW def_readwrite (
    "mean_load_kW" ,
    &ElectricalLoad::mean\_load\_kW )
```

5.33.2.3 `def_readwrite()` [3/4]

```
& ElectricalLoad::n\_points def_readwrite (
    "n_years" ,
    &ElectricalLoad::n\_years )
```

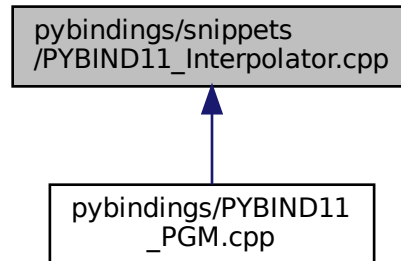
5.33.2.4 `def_readwrite()` [4/4]

```
& ElectricalLoad::n\_points & ElectricalLoad::min\_load\_kW & ElectricalLoad::max\_load\_kW def_↵
readwrite (
    "path_2_electrical_load_time_series" ,
    &ElectricalLoad::path\_2\_electrical\_load\_time\_series )
```

5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference

Bindings file for the [Interpolator](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [&InterpolatorStruct1D::n_points](#) [def_readwrite](#) ("x_vec", &InterpolatorStruct1D::x_vec) [.def_readwrite](#) ("min_x", &InterpolatorStruct1D::min_x) [.def_readwrite](#) ("max_x", &InterpolatorStruct1D::max_x) [.def_readwrite](#) ("y_vec", &InterpolatorStruct1D::y_vec) [.def_readwrite](#) ("z_matrix", &InterpolatorStruct1D::z_matrix)
- [&InterpolatorStruct1D::n_points](#) [&InterpolatorStruct1D::min_x](#) [def_readwrite](#) ("max_x", &InterpolatorStruct1D::max_x) [.def_readwrite](#) ("y_vec", &InterpolatorStruct1D::y_vec) [.def_readwrite](#) ("z_matrix", &InterpolatorStruct1D::z_matrix)
- [&InterpolatorStruct1D::n_points](#) [&InterpolatorStruct1D::min_x](#) [&InterpolatorStruct1D::y_vec](#) [def](#) (pybind11::init())
- [&InterpolatorStruct2D::n_rows](#) [def_readwrite](#) ("n_cols", &InterpolatorStruct2D::n_cols) [.def_readwrite](#) ("x_vec", &InterpolatorStruct2D::x_vec) [.def_readwrite](#) ("min_x", &InterpolatorStruct2D::min_x) [.def_readwrite](#) ("max_x", &InterpolatorStruct2D::max_x) [.def_readwrite](#) ("y_vec", &InterpolatorStruct2D::y_vec) [.def_readwrite](#) ("min_y", &InterpolatorStruct2D::min_y) [.def_readwrite](#) ("max_y", &InterpolatorStruct2D::max_y) [.def_readwrite](#) ("z_matrix", &InterpolatorStruct2D::z_matrix)
- [&InterpolatorStruct2D::n_rows](#) [&InterpolatorStruct2D::x_vec](#) [&InterpolatorStruct2D::max_x](#) [def_readwrite](#) ("y_vec", &InterpolatorStruct2D::y_vec) [.def_readwrite](#) ("min_y", &InterpolatorStruct2D::min_y) [.def_readwrite](#) ("max_y", &InterpolatorStruct2D::max_y) [.def_readwrite](#) ("z_matrix", &InterpolatorStruct2D::z_matrix)
- [&InterpolatorStruct2D::n_rows](#) [&InterpolatorStruct2D::x_vec](#) [&InterpolatorStruct2D::max_x](#) [&InterpolatorStruct2D::min_y](#) [def_readwrite](#) ("max_y", &InterpolatorStruct2D::max_y) [.def_readwrite](#) ("z_matrix", &InterpolatorStruct2D::z_matrix)
- [&Interpolator::interp_map_1D](#) [def_readwrite](#) ("path_map_1D", &Interpolator::path_map_1D) [.def_readwrite](#) ("interp_map_2D", &Interpolator::path_map_2D)

5.34.1 Detailed Description

Bindings file for the [Interpolator](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Interpolator](#) class. Only public attributes/methods are bound!

5.34.2 Function Documentation

5.34.2.1 def()

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x & InterpolatorStruct1D::y_vec  
def (   
    pybind11::init() )
```

5.34.2.2 def_readwrite() [1/7]

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x def_readwrite (   
    "max_x" ,   
    &InterpolatorStruct1D::max_x )
```

5.34.2.3 def_readwrite() [2/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x &   
InterpolatorStruct2D::min_y def_readwrite (   
    "max_y" ,   
    &InterpolatorStruct2D::max_y )
```

5.34.2.4 def_readwrite() [3/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec def_readwrite (   
    "min_x" ,   
    &InterpolatorStruct2D::min_x )
```

5.34.2.5 def_readwrite() [4/7]

```
& InterpolatorStruct2D::n_rows def_readwrite (   
    "n_cols" ,   
    &InterpolatorStruct2D::n_cols )
```

5.34.2.6 def_readwrite() [5/7]

```
& Interpolator::interp_map_1D def_readwrite (   
    "path_map_1D" ,   
    &Interpolator::path_map_1D )
```

5.34.2.7 `def_readwrite()` [6/7]

```
& InterpolatorStruct1D::n_points def_readwrite (
    "x_vec" ,
    &InterpolatorStruct1D::x_vec )
```

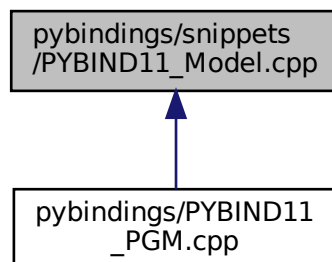
5.34.2.8 `def_readwrite()` [7/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x
def_readwrite (
    "y_vec" ,
    &InterpolatorStruct2D::y_vec )
```

5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference

Bindings file for the [Model](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Variables

- `&ModelInputs::path_2_electrical_load_time_series` `def_readwrite("control_mode", &ModelInputs::control_mode)` `.def(pybind11 &Model::total_fuel_consumed_L` `def_readwrite ("total_emissions", &Model::total_emissions)` `.def_readwrite("net_present_cost"`

5.35.1 Detailed Description

Bindings file for the [Model](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Model](#) class. Only public attributes/methods are bound!

5.35.2 Variable Documentation

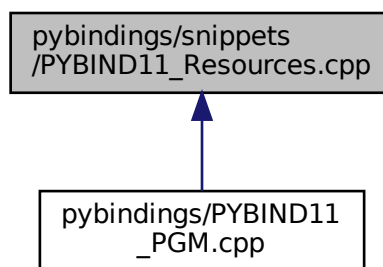
5.35.2.1 def_readwrite

```
& ModelInputs::path_2_electrical_load_time_series def_readwrite ("control_mode", &Model↵
Inputs::control_mode) .def(pybind11 & Model::total_fuel_consumed_L & Model::net_present_cost &
Model::total_dispatch_discharge_kWh & Model::controller & Model::resources & Model::noncombustion_ptr_vec
def_readwrite("renewable_ptr_vec", &Model::renewable_ptr_vec) .def_readwrite("storage_ptr_vec"
(
    "total_emissions" ,
    &Model::total_emissions )
```

5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference

Bindings file for the [Resources](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [&Resources::resource_map_1D](#) `def_readwrite` ("string_map_1D", &Resources::string_map_1D) `.def↵`
`readwrite("path_map_1D"`
- [&Resources::resource_map_1D](#) & [Resources::path_map_1D](#) `def_readwrite` ("resource_map_2D", &Resources↵
`::resource_map_2D) .def_readwrite("string_map_2D"`

5.36.1 Detailed Description

Bindings file for the [Resources](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Resources](#) class. Only public attributes/methods are bound!

5.36.2 Function Documentation

5.36.2.1 `def_readwrite()` [1/2]

```
& Resources::resource_map_1D & Resources::path_map_1D def_readwrite (
    "resource_map_2D" ,
    &Resources::resource_map_2D )
```

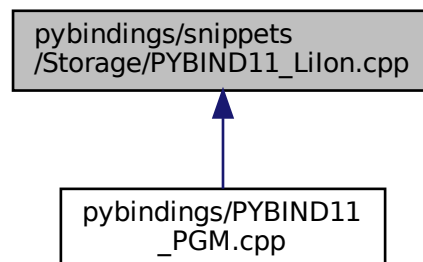
5.36.2.2 `def_readwrite()` [2/2]

```
& Resources::resource_map_1D def_readwrite (
    "string_map_1D" ,
    &Resources::string_map_1D )
```

5.37 `pybindings/snippets/Storage/PYBIND11_Lilon.cpp` File Reference

Bindings file for the [Lilon](#) class. Intended to be `#include'd` in [PYBIND11_PGM.cpp](#).

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



Functions

- `&LilonInputs::storage_inputs def_readwrite ("capital_cost", &LilonInputs::capital_cost) .def_readwrite("operation↔_maintenance_cost_kWh"`
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh def_readwrite ("init_SOC", &LilonInputs::init_SOC) .def_readwrite("min_SOC"`
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC def_readwrite ("hysteresis_SOC", &LilonInputs::hysteresis_SOC) .def_readwrite("max_SOC"`

- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `def_readwrite` ("charging_efficiency", `&LilonInputs::charging_efficiency`) `.def_readwrite`("discharging_efficiency"
- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `def_readwrite` ("replace_SOH", `&LilonInputs::replace_SOH`) `.def_readwrite`("power_degradation_flag"
- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `&LilonInputs::power_degradation_flag` `def_readwrite` ("degradation_alpha", `&LilonInputs::degradation_alpha`) `.def_readwrite`("degradation_beta"
- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `&LilonInputs::power_degradation_flag` `&LilonInputs::degradation_beta` `def_readwrite` ("degradation_B_hat_cal_0", `&LilonInputs::degradation_B_hat_cal_0`) `.def_readwrite`("degradation_r_cal"
- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `&LilonInputs::power_degradation_flag` `&LilonInputs::degradation_beta` `&LilonInputs::degradation_r_cal` `def_readwrite` ("degradation_Ea_cal_0", `&LilonInputs::degradation_Ea_cal_0`) `.def_readwrite`("degradation_a_cal"
- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `&LilonInputs::power_degradation_flag` `&LilonInputs::degradation_beta` `&LilonInputs::degradation_r_cal` `&LilonInputs::degradation_a_cal` `def_readwrite` ("degradation_s_cal", `&LilonInputs::degradation_s_cal`) `.def_readwrite`("gas_constant_JmolK"

Variables

- `&LilonInputs::storage_inputs` `&LilonInputs::operation_maintenance_cost_kWh` `&LilonInputs::min_SOC` `&LilonInputs::max_SOC` `&LilonInputs::discharging_efficiency` `&LilonInputs::power_degradation_flag` `&LilonInputs::degradation_beta` `&LilonInputs::degradation_r_cal` `&LilonInputs::degradation_a_cal` `&LilonInputs::gas_constant_JmolK` `def_readwrite`("gas_constant_JmolK", `&LilonInputs::gas_constant_JmolK`) `.def`(pybind11 `&Lilon::power_degradation_flag` `def_readwrite` ("dynamic_energy_capacity_kWh", `&Lilon::dynamic_energy_capacity_kWh`) `.def_readwrite`("dynamic_energy_capacity_kWh"

5.37.1 Detailed Description

Bindings file for the `Lilon` class. Intended to be `#include'd` in `PYBIND11_PGM.cpp`.

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the `Lilon` class. Only public attributes/methods are bound!

5.37.2 Function Documentation

5.37.2.1 `def_readwrite()` [1/9]

```
& LiIonInputs::storage_inputs def_readwrite (
    "capital_cost" ,
    &LiIonInputs::capital_cost )
```

5.37.2.2 def_readwrite() [2/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC def_readwrite (
    "charging_efficiency" ,
    &LiIonInputs::charging_efficiency )
```

5.37.2.3 def_readwrite() [3/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
def_readwrite (
    "degradation_alpha" ,
    &LiIonInputs::degradation_alpha )
```

5.37.2.4 def_readwrite() [4/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta def_readwrite (
    "degradation_B_hat_cal_0" ,
    &LiIonInputs::degradation_B_hat_cal_0 )
```

5.37.2.5 def_readwrite() [5/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal def_readwrite (
    "degradation_Ea_cal_0" ,
    &LiIonInputs::degradation_Ea_cal_0 )
```

5.37.2.6 def_readwrite() [6/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal & LiIonInputs::degradation_a_cal
def_readwrite (
    "degradation_s_cal" ,
    &LiIonInputs::degradation_s_cal )
```

5.37.2.7 def_readwrite() [7/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
def_readwrite (
    "hysteresis_SOC" ,
    &LiIonInputs::hysteresis_SOC )
```

5.37.2.8 def_readwrite() [8/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh def_readwrite (
    "init_SOC" ,
    &LiIonInputs::init_SOC )
```

5.37.2.9 def_readwrite() [9/9]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency def_readwrite (
    "replace_SOH" ,
    &LiIonInputs::replace_SOH )
```

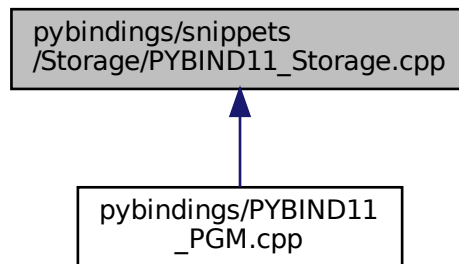
5.37.3 Variable Documentation**5.37.3.1 def_readwrite**

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::power_degradation_flag
& LiIonInputs::degradation_beta & LiIonInputs::degradation_r_cal & LiIonInputs::degradation_a_cal
& LiIonInputs::gas_constant_JmolK def_readwrite ("gas_constant_JmolK", &LiIonInputs::gas_↵
constant_JmolK) .def(pybind11 & LiIon::power_degradation_flag & LiIon::dynamic_power_capacity_kW
& LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal & LiIon::degradation_a_cal
& LiIon::gas_constant_JmolK & LiIon::init_SOC & LiIon::hysteresis_SOC & LiIon::charging_efficiency
def_readwrite("discharging_efficiency", &LiIon::discharging_efficiency) .def_readwrite("SOH_↵
vec" (
    "dynamic_energy_capacity_kWh" ,
    &LiIon::dynamic_energy_capacity_kWh )
```

5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference

Bindings file for the [Storage](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [StorageType::LIION value](#) ("N_STORAGE_TYPES", StorageType::N_STORAGE_TYPES)
- [&StorageInputs::print_flag](#) [def_readwrite](#) ("is_sunk", &StorageInputs::is_sunk) [.def_readwrite](#)("power_capacity_kW"
- [&StorageInputs::print_flag](#) [&StorageInputs::power_capacity_kW](#) [def_readwrite](#) ("energy_capacity_kWh", &StorageInputs::energy_capacity_kWh) [.def_readwrite](#)("nominal_inflation_annual"

Variables

- [&StorageInputs::print_flag](#) [&StorageInputs::power_capacity_kW](#) [&StorageInputs::nominal_inflation_annual](#) [def_readwrite](#)("nominal_discount_annual", &StorageInputs::nominal_discount_annual) [.def](#)(pybind11 [&Storage::type](#) [def_readwrite](#) ("interpolator", &Storage::interpolator) [.def_readwrite](#)("print_flag"

5.38.1 Detailed Description

Bindings file for the [Storage](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

Ref: [Jakob \[2023\]](#)

A file which instructs pybind11 how to build Python bindings for the [Storage](#) class. Only public attributes/methods are bound!

5.38.2 Function Documentation

5.38.2.1 def_readwrite() [1/2]

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW def_readwrite (
    "energy_capacity_kWh" ,
    &StorageInputs::energy_capacity_kWh )
```

5.38.2.2 def_readwrite() [2/2]

```
& StorageInputs::print_flag def_readwrite (
    "is_sunk" ,
    &StorageInputs::is_sunk )
```

5.38.2.3 value()

```
StorageType::LIION value (
    "N_STORAGE_TYPES" ,
    StorageType::N_STORAGE_TYPES )
```

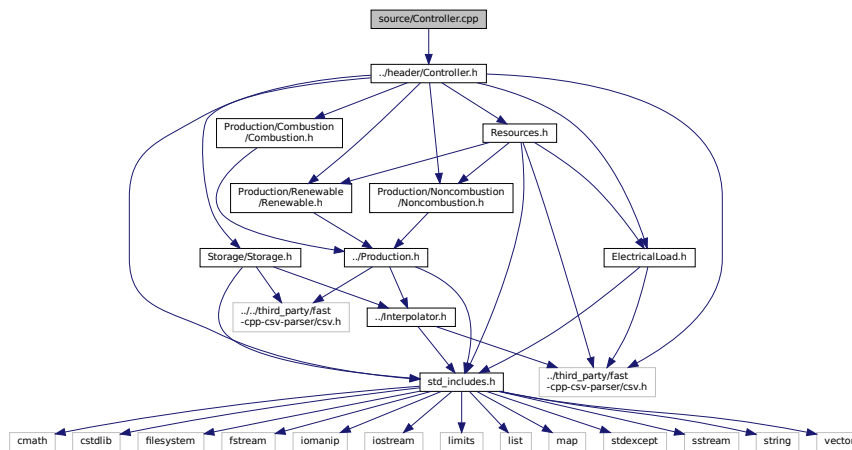
5.38.3 Variable Documentation**5.38.3.1 def_readwrite**

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW & StorageInputs::nominal_inflation_annual
def_readwrite ("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11
& Storage::type & Storage::print_flag & Storage::is_sunk & Storage::n_replacements & Storage::power_capacity_k
& Storage::charge_kWh & Storage::nominal_inflation_annual & Storage::real_discount_annual &
Storage::operation_maintenance_cost_kWh & Storage::total_discharge_kWh & Storage::type_str &
Storage::charging_power_vec_kW def_readwrite("discharging_power_vec_kW", &Storage::discharging_
_power_vec_kW) .def_readwrite("capital_cost_vec" (
    "interpolator" ,
    &Storage::interpolator )
```

5.39 source/Controller.cpp File Reference

Implementation file for the [Controller](#) class.

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



5.39.1 Detailed Description

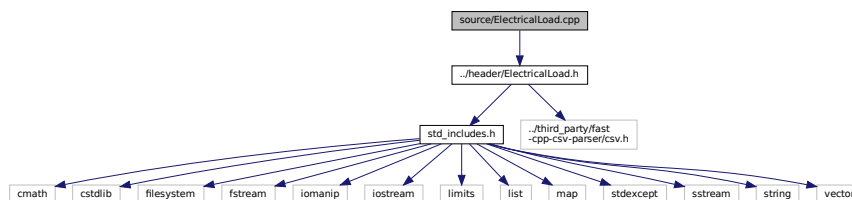
Implementation file for the [Controller](#) class.

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

5.40 source/ElectricalLoad.cpp File Reference

Implementation file for the [ElectricalLoad](#) class.

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



5.40.1 Detailed Description

Implementation file for the [ElectricalLoad](#) class.

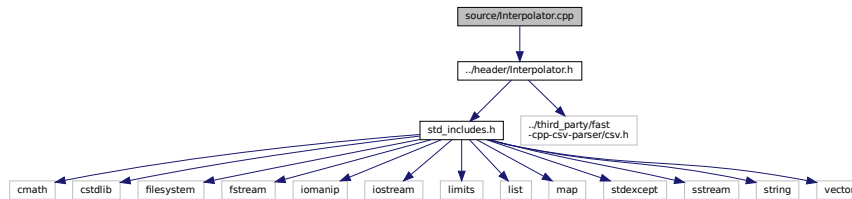
A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

5.41 source/Interpolator.cpp File Reference

Implementation file for the [Interpolator](#) class.

```
#include "../header/Interpolator.h"
```

Include dependency graph for Interpolator.cpp:



5.41.1 Detailed Description

Implementation file for the [Interpolator](#) class.

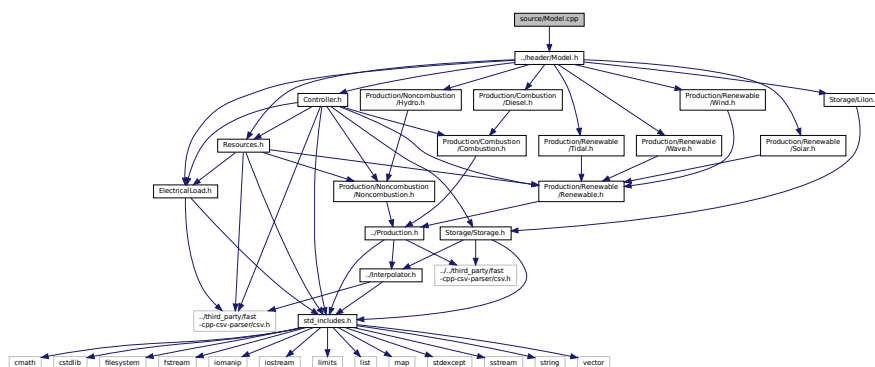
A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

5.42 source/Model.cpp File Reference

Implementation file for the [Model](#) class.

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

Include dependency graph for Model.cpp:



5.42.1 Detailed Description

Implementation file for the [Model](#) class.

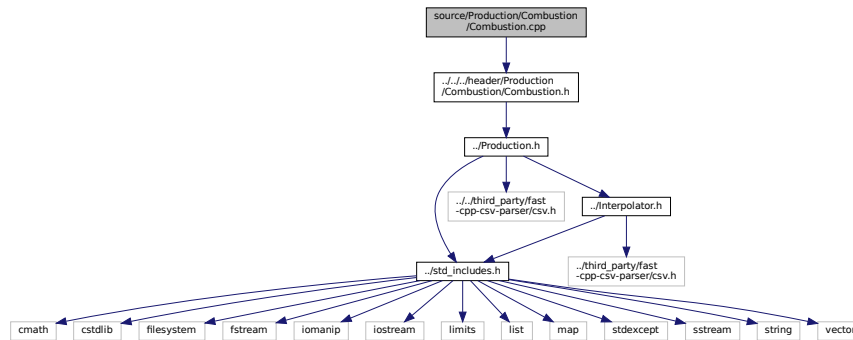
A container class which forms the centre of PGMcpp. The [Model](#) class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

5.43 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:



5.43.1 Detailed Description

Implementation file for the [Combustion](#) class.

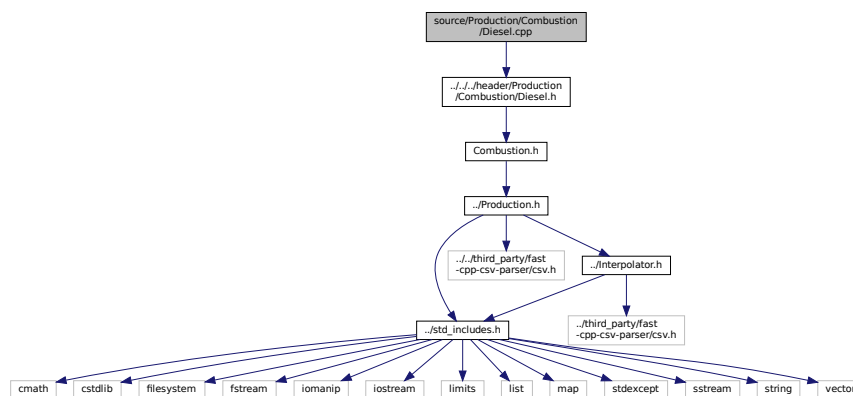
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

5.44 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:



5.44.1 Detailed Description

Implementation file for the [Diesel](#) class.

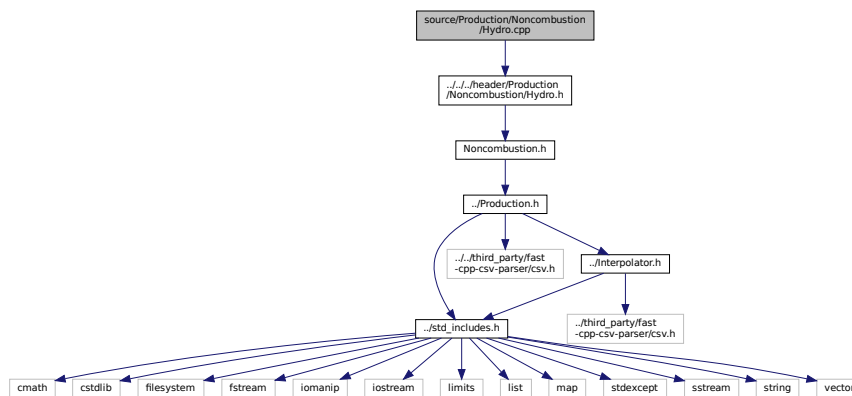
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

5.45 source/Production/Noncombustion/Hydro.cpp File Reference

Implementation file for the [Hydro](#) class.

```
#include "../.../header/Production/Noncombustion/Hydro.h"
```

Include dependency graph for Hydro.cpp:



5.45.1 Detailed Description

Implementation file for the [Hydro](#) class.

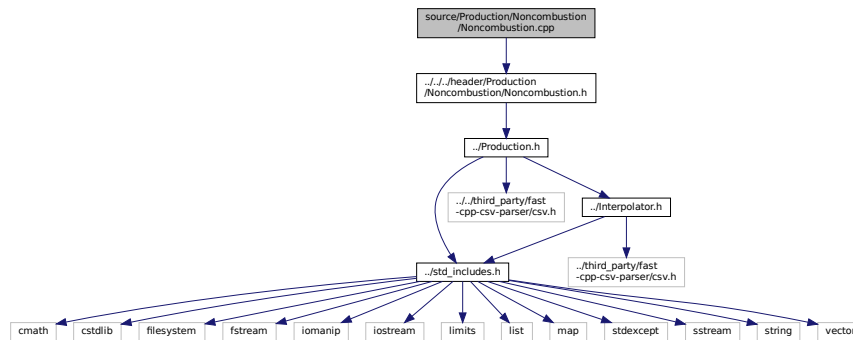
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference

Implementation file for the [Noncombustion](#) class.

```
#include "../../../header/Production/Noncombustion/Noncombustion.h"
```

Include dependency graph for Noncombustion.cpp:



5.46.1 Detailed Description

Implementation file for the [Noncombustion](#) class.

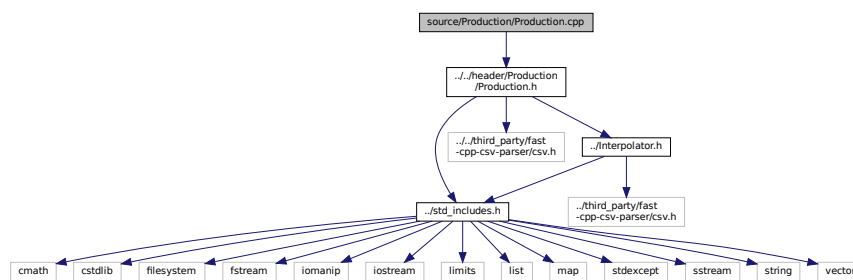
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

5.47 source/Production/Production.cpp File Reference

Implementation file for the [Production](#) class.

```
#include "../../../header/Production/Production.h"
```

Include dependency graph for Production.cpp:



5.47.1 Detailed Description

Implementation file for the [Production](#) class.

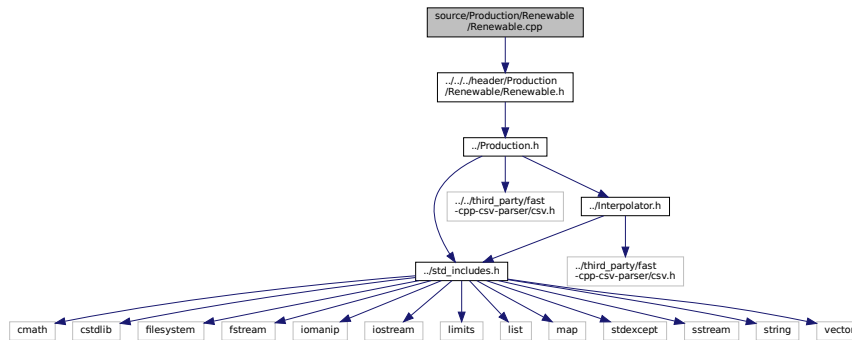
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.48 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:



5.48.1 Detailed Description

Implementation file for the [Renewable](#) class.

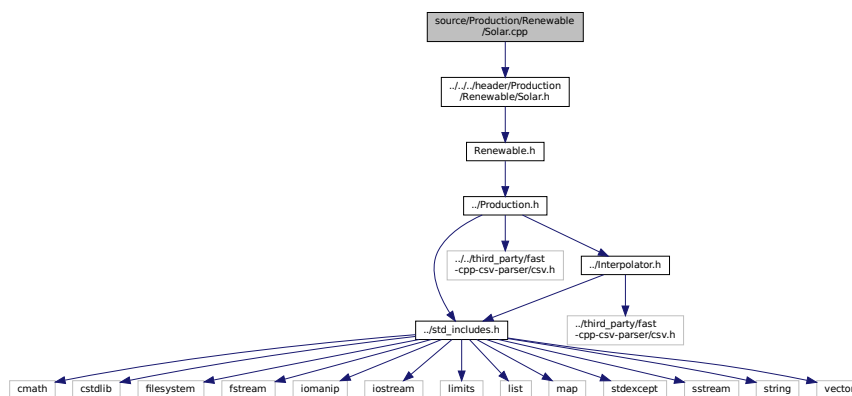
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

5.49 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:



5.49.1 Detailed Description

Implementation file for the [Solar](#) class.

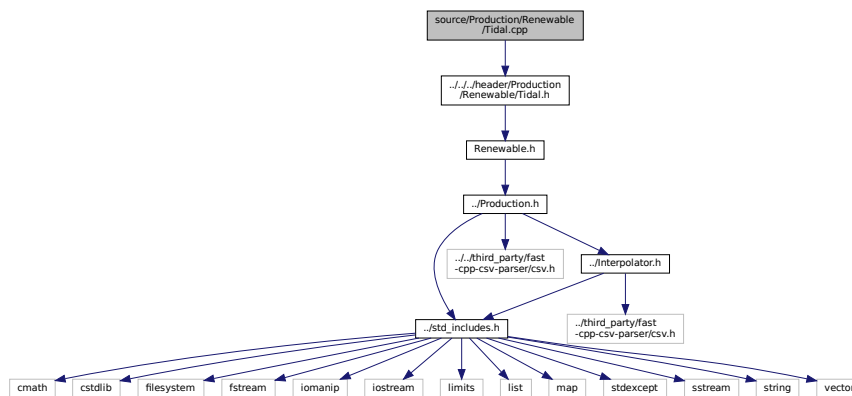
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

5.50 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:



5.50.1 Detailed Description

Implementation file for the [Tidal](#) class.

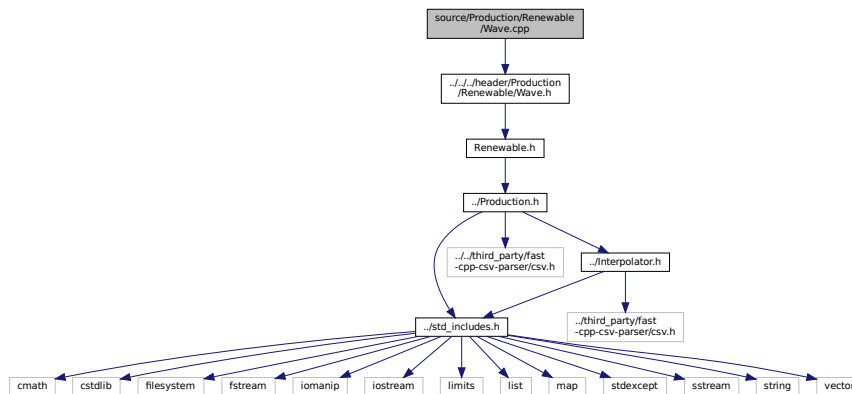
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

5.51 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:



5.51.1 Detailed Description

Implementation file for the [Wave](#) class.

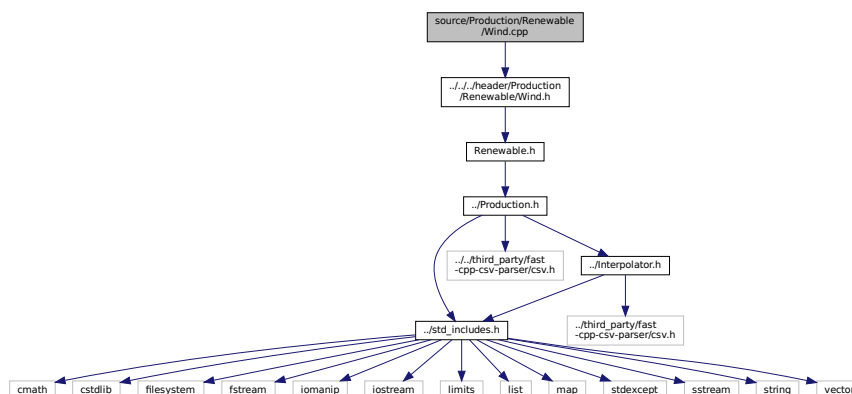
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

5.52 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:



5.52.1 Detailed Description

Implementation file for the [Wind](#) class.

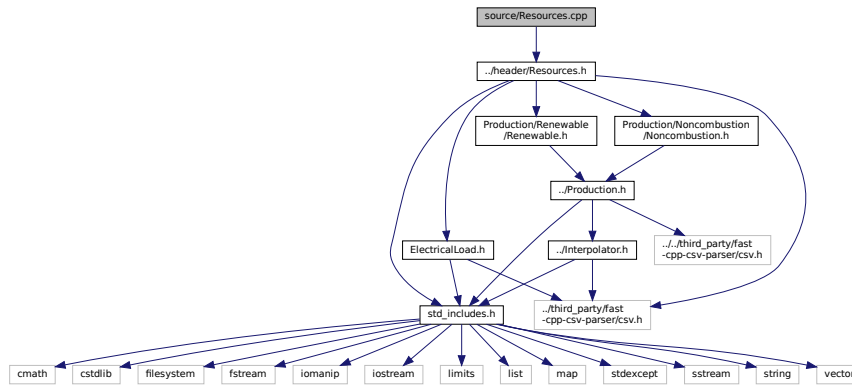
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

5.53 source/Resources.cpp File Reference

Implementation file for the [Resources](#) class.

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

Include dependency graph for Resources.cpp:



5.53.1 Detailed Description

Implementation file for the [Resources](#) class.

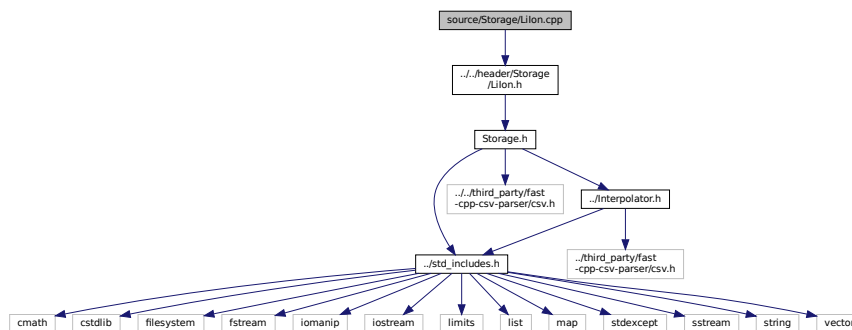
A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

5.54 source/Storage/Lilon.cpp File Reference

Implementation file for the [Lilon](#) class.

```
#include "../../header/Storage/LiIon.h"
```

Include dependency graph for Lilon.cpp:



5.54.1 Detailed Description

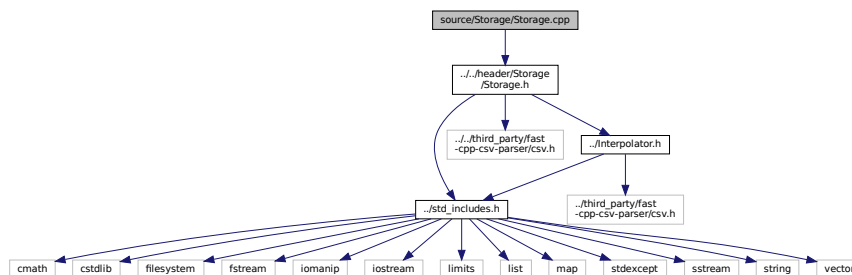
Implementation file for the [Lilon](#) class.

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

5.55 source/Storage/Storage.cpp File Reference

Implementation file for the [Storage](#) class.

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



5.55.1 Detailed Description

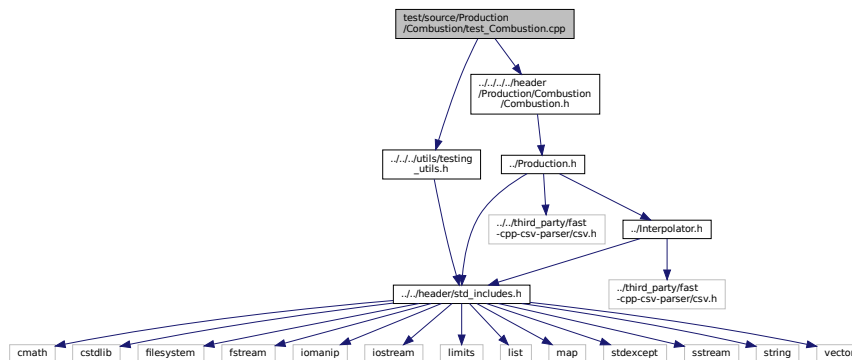
Implementation file for the [Storage](#) class.

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

5.56 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

- [Combustion](#) * [testConstruct_Combustion](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Combustion](#) object and spot check some post-construction attributes.
- int [main](#) (int argc, char **argv)

5.56.1 Detailed Description

Testing suite for [Combustion](#) class.

A suite of tests for the [Combustion](#) class.

5.56.2 Function Documentation

5.56.2.1 main()

```
int main (
    int argc,
    char ** argv )
147 {
148     #ifdef _WIN32
149         activateVirtualTerminal();
150     #endif /* _WIN32 */
151
152     printGold("\tTesting Production <-- Combustion");
153
154     srand(time(NULL));
155
156
157     std::vector<double> time_vec_hrs (8760, 0);
158     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
159         time_vec_hrs[i] = i;
160     }
161
162     Combustion* test_combustion_ptr = testConstruct_Combustion (&time_vec_hrs);
163
164
165     try {
166         //...
167     }
168
169
170     catch (...) {
171         delete test_combustion_ptr;
172
173         printGold(" ..... ");
174         printRed("FAIL");
175         std::cout << std::endl;
176         throw;
177     }
178
179
180     delete test_combustion_ptr;
181
182     printGold(" ..... ");
183     printGreen("PASS");
184     std::cout << std::endl;
185     return 0;
186
187 } /* main() */
```

5.56.2.2 testConstruct_Combustion()

```
Combustion * testConstruct_Combustion (
    std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a [Combustion](#) object and spot check some post-construction attributes.

Parameters

<code>time_vec_hrs_ptr</code>	A pointer to the vector containing the modelling time series.
-------------------------------	---

Returns

A pointer to a test [Combustion](#) object.

```

65 {
66     CombustionInputs combustion_inputs;
67
68     Combustion* test_combustion_ptr = new Combustion(
69         8760,
70         1,
71         combustion_inputs,
72         time_vec_hrs_ptr
73     );
74
75     testTruth(
76         not combustion_inputs.production_inputs.print_flag,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_combustion_ptr->fuel_consumption_vec_L.size(),
83         8760,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         test_combustion_ptr->fuel_cost_vec.size(),
90         8760,
91         __FILE__,
92         __LINE__
93     );
94
95     testFloatEquals(
96         test_combustion_ptr->CO2_emissions_vec_kg.size(),
97         8760,
98         __FILE__,
99         __LINE__
100    );
101
102    testFloatEquals(
103        test_combustion_ptr->CO_emissions_vec_kg.size(),
104        8760,
105        __FILE__,
106        __LINE__
107    );
108
109    testFloatEquals(
110        test_combustion_ptr->NOx_emissions_vec_kg.size(),
111        8760,
112        __FILE__,
113        __LINE__
114    );
115
116    testFloatEquals(
117        test_combustion_ptr->SOx_emissions_vec_kg.size(),
118        8760,
119        __FILE__,
120        __LINE__
121    );
122
123    testFloatEquals(
124        test_combustion_ptr->CH4_emissions_vec_kg.size(),
125        8760,
126        __FILE__,
127        __LINE__
128    );
129
130    testFloatEquals(
131        test_combustion_ptr->PM_emissions_vec_kg.size(),
132        8760,
133        __FILE__,
134        __LINE__
135    );
136
137    return test_combustion_ptr;
138 } /* testConstruct_Combustion() */

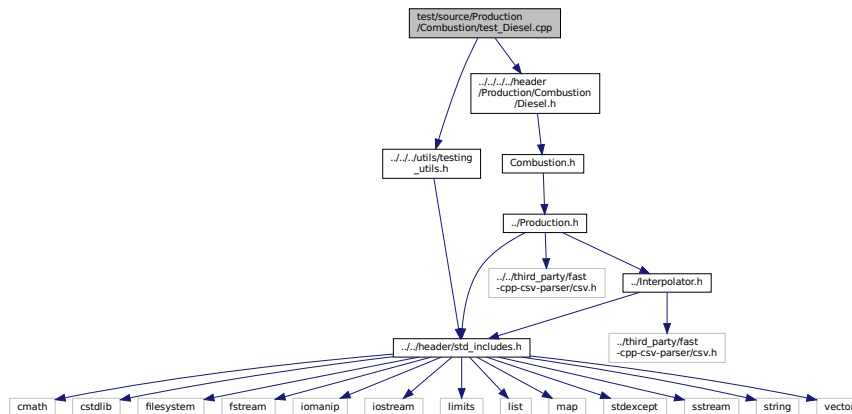
```

5.57 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

- [Combustion](#) * [testConstruct_Diesel](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Diesel](#) object and spot check some post-construction attributes.
- [Combustion](#) * [testConstructLookup_Diesel](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Diesel](#) object using fuel consumption lookup.
- void [testBadConstruct_Diesel](#) (std::vector< double > *time_vec_hrs_ptr)
Function to test the trying to construct a [Diesel](#) object given bad inputs is being handled as expected.
- void [testCapacityConstraint_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Test to check that the installed capacity constraint is active and behaving as expected.
- void [testMinimumLoadRatioConstraint_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Test to check that the minimum load ratio constraint is active and behaving as expected.
- void [testCommit_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Diesel](#) object.
- void [testMinimumRuntimeConstraint_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Function to check that the minimum runtime constraint is active and behaving as expected.
- void [testFuelConsumptionEmissions_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Function to test that post-commit fuel consumption and emissions are > 0 when the test [Diesel](#) object is running, and = 0 when it is not (as expected).
- void [testEconomics_Diesel](#) ([Combustion](#) *test_diesel_ptr)
Function to test that the post-commit model economics for the test [Diesel](#) object are as expected (> 0 when running, = 0 when not).
- void [testFuelLookup_Diesel](#) ([Combustion](#) *test_diesel_lookup_ptr)
Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.
- int [main](#) (int argc, char **argv)

5.57.1 Detailed Description

Testing suite for [Diesel](#) class.

A suite of tests for the [Diesel](#) class.

5.57.2 Function Documentation

5.57.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    730 {
    731     #ifdef _WIN32
    732         activateVirtualTerminal();
    733     #endif /* _WIN32 */
    734
    735     printGold("\tTesting Production <-- Combustion <-- Diesel");
    736
    737     srand(time(NULL));
    738
    739
    740     std::vector<double> time_vec_hrs (8760, 0);
    741     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
    742         time_vec_hrs[i] = i;
    743     }
    744
    745
    746     Combustion* test_diesel_ptr = testConstruct_Diesel(&time_vec_hrs);
    747     Combustion* test_diesel_lookup_ptr = testConstructLookup_Diesel(&time_vec_hrs);
    748
    749     try {
    750         testBadConstruct_Diesel(&time_vec_hrs);
    751
    752         testCapacityConstraint_Diesel(test_diesel_ptr);
    753         testMinimumLoadRatioConstraint_Diesel(test_diesel_ptr);
    754
    755         testCommit_Diesel(test_diesel_ptr);
    756
    757         testMinimumRuntimeConstraint_Diesel(test_diesel_ptr);
    758
    759         testFuelConsumptionEmissions_Diesel(test_diesel_ptr);
    760         testEconomics_Diesel(test_diesel_ptr);
    761
    762         testFuelLookup_Diesel(test_diesel_lookup_ptr);
    763     }
    764
    765
    766     catch (...) {
    767         delete test_diesel_ptr;
    768         delete test_diesel_lookup_ptr;
    769
    770         printGold(" ..... ");
    771         printRed("FAIL");
    772         std::cout << std::endl;
    773         throw;
    774     }
    775
    776
    777     delete test_diesel_ptr;
    778     delete test_diesel_lookup_ptr;
    779
    780     printGold(" ..... ");
    781     printGreen("PASS");
    782     std::cout << std::endl;
    783     return 0;
    784
    785 } /* main() */
```

5.57.2.2 testBadConstruct_Diesel()

```
void testBadConstruct_Diesel (
    std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a [Diesel](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```
203 {
204     bool error_flag = true;
205
206     try {
207         DieselInputs bad_diesel_inputs;
208         bad_diesel_inputs.fuel_cost_L = -1;
209
210         Diesel bad_diesel(
211             8760,
212             1,
213             bad_diesel_inputs,
214             time_vec_hrs_ptr
215         );
216
217         error_flag = false;
218     } catch (...) {
219         // Task failed successfully! =P
220     }
221     if (not error_flag) {
222         expectedErrorNotDetected(__FILE__, __LINE__);
223     }
224
225     return;
226 } /* testBadConstruct_Diesel() */
```

5.57.2.3 testCapacityConstraint_Diesel()

```
void testCapacityConstraint_Diesel (
    Combustion * test_diesel_ptr )
```

Test to check that the installed capacity constraint is active and behaving as expected.

Parameters

<i>test_diesel_ptr</i>	A Combustion pointer to the test Diesel object.
------------------------	---

```
244 {
245     testFloatEquals(
246         test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
247         test_diesel_ptr->capacity_kW,
248         __FILE__,
249         __LINE__
250     );
251
252     return;
253 } /* testCapacityConstraint_Diesel() */
```

5.57.2.4 testCommit_Diesel()

```
void testCommit_Diesel (
    Combustion * test_diesel_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Diesel](#) object.

Parameters

<code>test_diesel_ptr</code>	A Combustion pointer to the test Diesel object.
------------------------------	---

```

303 {
304     std::vector<double> dt_vec_hrs (48, 1);
305
306     std::vector<double> load_vec_kW = {
307         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
308         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
309         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
310         1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
311     };
312
313     double load_kW = 0;
314     double production_kW = 0;
315     double roll = 0;
316
317     for (int i = 0; i < 48; i++) {
318         roll = (double)rand() / RAND_MAX;
319
320         if (roll >= 0.95) {
321             roll = 1.25;
322         }
323
324         load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
325         load_kW = load_vec_kW[i];
326
327         production_kW = test_diesel_ptr->requestProductionkW(
328             i,
329             dt_vec_hrs[i],
330             load_kW
331         );
332
333         load_kW = test_diesel_ptr->commit(
334             i,
335             dt_vec_hrs[i],
336             production_kW,
337             load_kW
338         );
339
340         // load_kW <= load_vec_kW (i.e., after vs before)
341         testLessThanOrEqualTo(
342             load_kW,
343             load_vec_kW[i],
344             __FILE__,
345             __LINE__
346         );
347
348         // production = dispatch + storage + curtailment
349         testFloatEquals(
350             test_diesel_ptr->production_vec_kW[i] -
351             test_diesel_ptr->dispatch_vec_kW[i] -
352             test_diesel_ptr->storage_vec_kW[i] -
353             test_diesel_ptr->curtailment_vec_kW[i],
354             0,
355             __FILE__,
356             __LINE__
357         );
358
359         // capacity constraint
360         if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
361             testFloatEquals(
362                 test_diesel_ptr->production_vec_kW[i],
363                 test_diesel_ptr->capacity_kW,
364                 __FILE__,
365                 __LINE__
366             );
367         }
368
369         // minimum load ratio constraint
370         else if (
371             test_diesel_ptr->is_running and
372             test_diesel_ptr->production_vec_kW[i] > 0 and
373             load_vec_kW[i] <
374             ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
375         ) {
376             testFloatEquals(
377                 test_diesel_ptr->production_vec_kW[i],
378                 ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
379                 test_diesel_ptr->capacity_kW,

```

```

380         __FILE__,
381         __LINE__
382     );
383 }
384 }
385
386 return;
387 } /* testCommit_Diesel() */

```

5.57.2.5 testConstruct_Diesel()

```

Combustion * testConstruct_Diesel (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Diesel](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Combustion](#) pointer to a test [Diesel](#) object.

```

65 {
66     DieselInputs diesel_inputs;
67
68     Combustion* test_diesel_ptr = new Diesel(
69         8760,
70         1,
71         diesel_inputs,
72         time_vec_hrs_ptr
73     );
74
75     testTruth(
76         not diesel_inputs.combustion_inputs.production_inputs.print_flag,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_diesel_ptr->type,
83         CombustionType :: DIESEL,
84         __FILE__,
85         __LINE__
86     );
87
88     testTruth(
89         test_diesel_ptr->type_str == "DIESEL",
90         __FILE__,
91         __LINE__
92     );
93
94     testFloatEquals(
95         test_diesel_ptr->linear_fuel_slope_LkWh,
96         0.265675,
97         __FILE__,
98         __LINE__
99     );
100
101     testFloatEquals(
102         test_diesel_ptr->linear_fuel_intercept_LkWh,
103         0.026676,
104         __FILE__,
105         __LINE__
106     );
107
108     testFloatEquals(
109         test_diesel_ptr->capital_cost,
110         94125.375446,
111         __FILE__,
112         __LINE__

```

```

113     );
114
115     testFloatEquals(
116         test_diesel_ptr->operation_maintenance_cost_kWh,
117         0.069905,
118         __FILE__,
119         __LINE__
120     );
121
122     testFloatEquals(
123         ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
124         0.2,
125         __FILE__,
126         __LINE__
127     );
128
129     testFloatEquals(
130         ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
131         4,
132         __FILE__,
133         __LINE__
134     );
135
136     testFloatEquals(
137         test_diesel_ptr->replace_running_hrs,
138         30000,
139         __FILE__,
140         __LINE__
141     );
142
143     testFloatEquals(
144         test_diesel_ptr->cycle_charging_setpoint,
145         0.85,
146         __FILE__,
147         __LINE__
148     );
149
150     return test_diesel_ptr;
151 } /* testConstruct_Diesel() */

```

5.57.2.6 testConstructLookup_Diesel()

```

Combustion * testConstructLookup_Diesel (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Diesel](#) object using fuel consumption lookup.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Combustion](#) pointer to a test [Diesel](#) object.

```

170 {
171     DieselInputs diesel_inputs;
172
173     diesel_inputs.combustion_inputs.fuel_mode = FuelMode :: FUEL_MODE_LOOKUP;
174     diesel_inputs.combustion_inputs.path_2_fuel_interp_data =
175         "data/test/interpolation/diesel_fuel_curve.csv";
176
177     Combustion* test_diesel_lookup_ptr = new Diesel(
178         8760,
179         1,
180         diesel_inputs,
181         time_vec_hrs_ptr
182     );
183
184     return test_diesel_lookup_ptr;
185 } /* testConstructLookup_Diesel() */

```

5.57.2.7 testEconomics_Diesel()

```
void testEconomics_Diesel (
    Combustion * test_diesel_ptr )
```

Function to test that the post-commit model economics for the test [Diesel](#) object are as expected (> 0 when running, $= 0$ when not).

Parameters

<code>test_diesel_ptr</code>	A Combustion pointer to the test Diesel object.
------------------------------	---

```
607 {
608     std::vector<bool> expected_is_running_vec = {
609         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
610         1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1,
611         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
612         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
613     };
614
615     bool is_running = false;
616
617     for (int i = 0; i < 48; i++) {
618         is_running = test_diesel_ptr->is_running_vec[i];
619
620         testFloatEquals(
621             is_running,
622             expected_is_running_vec[i],
623             __FILE__,
624             __LINE__
625         );
626
627         // O&M, fuel consumption, and emissions > 0 whenever diesel is running
628         if (is_running) {
629             testGreaterThan(
630                 test_diesel_ptr->operation_maintenance_cost_vec[i],
631                 0,
632                 __FILE__,
633                 __LINE__
634             );
635         }
636
637         // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
638         else {
639             testFloatEquals(
640                 test_diesel_ptr->operation_maintenance_cost_vec[i],
641                 0,
642                 __FILE__,
643                 __LINE__
644             );
645         }
646     }
647
648     return;
649 } /* testEconomics_Diesel() */
```

5.57.2.8 testFuelConsumptionEmissions_Diesel()

```
void testFuelConsumptionEmissions_Diesel (
    Combustion * test_diesel_ptr )
```

Function to test that post-commit fuel consumption and emissions are > 0 when the test [Diesel](#) object is running, and $= 0$ when it is not (as expected).

Parameters

<code>test_diesel_ptr</code>	A Combustion pointer to the test Diesel object.
------------------------------	---


```

449 {
450     std::vector<bool> expected_is_running_vec = {
451         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
452         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
453         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
454         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
455     };
456
457     bool is_running = false;
458
459     for (int i = 0; i < 48; i++) {
460         is_running = test_diesel_ptr->is_running_vec[i];
461
462         testFloatEquals(
463             is_running,
464             expected_is_running_vec[i],
465             __FILE__,
466             __LINE__
467         );
468
469         // O&M, fuel consumption, and emissions > 0 whenever diesel is running
470         if (is_running) {
471             testGreaterThan(
472                 test_diesel_ptr->fuel_consumption_vec_L[i],
473                 0,
474                 __FILE__,
475                 __LINE__
476             );
477
478             testGreaterThan(
479                 test_diesel_ptr->fuel_cost_vec[i],
480                 0,
481                 __FILE__,
482                 __LINE__
483             );
484
485             testGreaterThan(
486                 test_diesel_ptr->CO2_emissions_vec_kg[i],
487                 0,
488                 __FILE__,
489                 __LINE__
490             );
491
492             testGreaterThan(
493                 test_diesel_ptr->CO_emissions_vec_kg[i],
494                 0,
495                 __FILE__,
496                 __LINE__
497             );
498
499             testGreaterThan(
500                 test_diesel_ptr->NOx_emissions_vec_kg[i],
501                 0,
502                 __FILE__,
503                 __LINE__
504             );
505
506             testGreaterThan(
507                 test_diesel_ptr->SOx_emissions_vec_kg[i],
508                 0,
509                 __FILE__,
510                 __LINE__
511             );
512
513             testGreaterThan(
514                 test_diesel_ptr->CH4_emissions_vec_kg[i],
515                 0,
516                 __FILE__,
517                 __LINE__
518             );
519
520             testGreaterThan(
521                 test_diesel_ptr->PM_emissions_vec_kg[i],
522                 0,
523                 __FILE__,
524                 __LINE__
525             );
526         }
527
528         // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
529         else {
530             testFloatEquals(
531                 test_diesel_ptr->fuel_consumption_vec_L[i],
532                 0,
533                 __FILE__,
534                 __LINE__
535             );

```

```

536
537     testFloatEquals (
538         test_diesel_ptr->fuel_cost_vec[i],
539         0,
540         __FILE__,
541         __LINE__
542     );
543
544     testFloatEquals (
545         test_diesel_ptr->CO2_emissions_vec_kg[i],
546         0,
547         __FILE__,
548         __LINE__
549     );
550
551     testFloatEquals (
552         test_diesel_ptr->CO_emissions_vec_kg[i],
553         0,
554         __FILE__,
555         __LINE__
556     );
557
558     testFloatEquals (
559         test_diesel_ptr->NOx_emissions_vec_kg[i],
560         0,
561         __FILE__,
562         __LINE__
563     );
564
565     testFloatEquals (
566         test_diesel_ptr->SOx_emissions_vec_kg[i],
567         0,
568         __FILE__,
569         __LINE__
570     );
571
572     testFloatEquals (
573         test_diesel_ptr->CH4_emissions_vec_kg[i],
574         0,
575         __FILE__,
576         __LINE__
577     );
578
579     testFloatEquals (
580         test_diesel_ptr->PM_emissions_vec_kg[i],
581         0,
582         __FILE__,
583         __LINE__
584     );
585 }
586 }
587
588 return;
589 } /* testFuelConsumptionEmissions_Diesel() */

```

5.57.2.9 testFuelLookup_Diesel()

```

void testFuelLookup_Diesel (
    Combustion * test_diesel_lookup_ptr )

```

Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.

Parameters

<code>test_diesel_lookup_ptr</code>	A Combustion pointer to the test Diesel object using fuel consumption lookup.
-------------------------------------	---

```

668 {
669     std::vector<double> load_ratio_vec = {
670         0,
671         0.170812859791767,
672         0.322739274162545,
673         0.369750203682042,
674         0.443532869135929,
675         0.471567864244626,
676         0.536513734479662,

```

```

677         0.586125806988674,
678         0.601101175455075,
679         0.658356862575221,
680         0.70576929893201,
681         0.784069734739331,
682         0.805765927542453,
683         0.884747873186048,
684         0.930870496062112,
685         0.979415217694769,
686         1
687     };
688
689     std::vector<double> expected_fuel_consumption_vec_L = {
690         4.68079520372916,
691         8.35159603357656,
692         11.7422361561399,
693         12.9931187917615,
694         14.8786636301325,
695         15.5746957307243,
696         17.1419229487141,
697         18.3041866133728,
698         18.6530540913696,
699         19.9569217633299,
700         21.012354614584,
701         22.7142305879957,
702         23.1916726441968,
703         24.8602332554707,
704         25.8172124624032,
705         26.8256741279932,
706         27.254952
707     };
708
709     for (size_t i = 0; i < load_ratio_vec.size(); i++) {
710         testFloatEquals(
711             test_diesel_lookup_ptr->getFuelConsumptionL(
712                 1, load_ratio_vec[i] * test_diesel_lookup_ptr->capacity_kW
713             ),
714             expected_fuel_consumption_vec_L[i],
715             __FILE__,
716             __LINE__
717         );
718     }
719
720     return;
721 } /* testFuelLookup_Diesel() */

```

5.57.2.10 testMinimumLoadRatioConstraint_Diesel()

```

void testMinimumLoadRatioConstraint_Diesel (
    Combustion * test_diesel_ptr )

```

Test to check that the minimum load ratio constraint is active and behaving as expected.

Parameters

<i>test_diesel_ptr</i>	A Combustion pointer to the test Diesel object.
------------------------	---

```

271 {
272     testFloatEquals(
273         test_diesel_ptr->requestProductionkW(
274             0,
275             1,
276             0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
277                 test_diesel_ptr->capacity_kW
278         ),
279         ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
280         __FILE__,
281         __LINE__
282     );
283
284     return;
285 } /* testMinimumLoadRatioConstraint_Diesel() */

```

5.57.2.11 testMinimumRuntimeConstraint_Diesel()

```
void testMinimumRuntimeConstraint_Diesel (
    Combustion * test_diesel_ptr )
```

Function to check that the minimum runtime constraint is active and behaving as expected.

Parameters

<code>test_diesel_ptr</code>	A Combustion pointer to the test Diesel object.
------------------------------	---

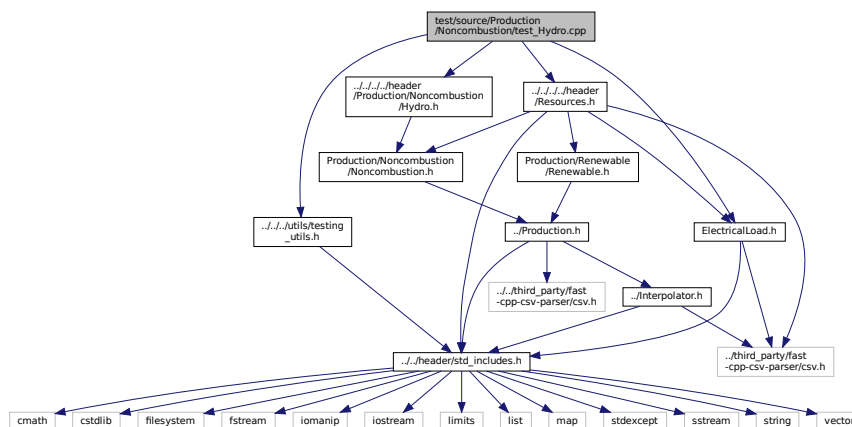
```
405 {
406     std::vector<double> load_vec_kW = {
407         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
408         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
409         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
410         1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
411     };
412
413     std::vector<bool> expected_is_running_vec = {
414         1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
415         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
416         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
417         1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
418     };
419
420     for (int i = 0; i < 48; i++) {
421         testFloatEquals(
422             test_diesel_ptr->is_running_vec[i],
423             expected_is_running_vec[i],
424             __FILE__,
425             __LINE__
426         );
427     }
428
429     return;
430 } /* testMinimumRuntimeConstraint_Diesel() */
```

5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference

Testing suite for [Hydro](#) class.

```
#include "../.../utils/testing_utils.h"
#include "../.../header/Resources.h"
#include "../.../header/ElectricalLoad.h"
#include "../.../header/Production/Noncombustion/Hydro.h"
```

Include dependency graph for test_Hydro.cpp:



Functions

- [Noncombustion](#) * [testConstruct_Hydro](#) ([HydroInputs](#) hydro_inputs, std::vector< double > *time_vec_hrs_↵ ptr)
A function to construct a [Hydro](#) object and spot check some post-construction attributes.
- void [testEfficiencyInterpolation_Hydro](#) ([Noncombustion](#) *test_hydro_ptr)
Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.
- void [testCommit_Hydro](#) ([Noncombustion](#) *test_hydro_ptr, [Resources](#) *test_resources_ptr)
- int [main](#) (int argc, char **argv)

5.58.1 Detailed Description

Testing suite for [Hydro](#) class.

A suite of tests for the [Hydro](#) class.

5.58.2 Function Documentation

5.58.2.1 main()

```

int main (
    int argc,
    char ** argv )
330 {
331     #ifdef _WIN32
332         activateVirtualTerminal();
333     #endif /* _WIN32 */
334
335     printGold("\tTesting Production <-- Noncombustion <-- Hydro");
336
337     srand(time(NULL));
  
```

```

338
339
340     std::vector<double> time_vec_hrs (8760, 0);
341     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
342         time_vec_hrs[i] = i;
343     }
344
345     std::string path_2_electrical_load_time_series =
346         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
347
348     ElectricalLoad* test_electrical_load_ptr =
349         new ElectricalLoad(path_2_electrical_load_time_series);
350
351     Resources* test_resources_ptr = new Resources();
352
353     HydroInputs hydro_inputs;
354     int hydro_resource_key = 0;
355
356     hydro_inputs.reservoir_capacity_m3 = 10000;
357     hydro_inputs.resource_key = hydro_resource_key;
358
359     Noncombustion* test_hydro_ptr = testConstruct_Hydro(hydro_inputs, &time_vec_hrs);
360
361     std::string path_2_hydro_resource_data =
362         "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
363
364     test_resources_ptr->addResource(
365         NoncombustionType::HYDRO,
366         path_2_hydro_resource_data,
367         hydro_resource_key,
368         test_electrical_load_ptr
369     );
370
371
372     try {
373         testEfficiencyInterpolation_Hydro(test_hydro_ptr);
374         testCommit_Hydro(test_hydro_ptr, test_resources_ptr);
375     }
376
377
378     catch (...) {
379         delete test_electrical_load_ptr;
380         delete test_resources_ptr;
381         delete test_hydro_ptr;
382
383         printGold(" ... ");
384         printRed("FAIL");
385         std::cout << std::endl;
386         throw;
387     }
388
389
390     delete test_electrical_load_ptr;
391     delete test_resources_ptr;
392     delete test_hydro_ptr;
393
394     printGold(" ... ");
395     printGreen("PASS");
396     std::cout << std::endl;
397     return 0;
398
399 } /* main() */

```

5.58.2.2 testCommit_Hydro()

```

void testCommit_Hydro (
    Noncombustion * test_hydro_ptr,
    Resources * test_resources_ptr )
247 {
248     double load_kW = 100 * (double)rand() / RAND_MAX;
249     double production_kW = 0;
250
251     for (int i = 0; i < 8760; i++) {
252         production_kW = test_hydro_ptr->requestProductionkW(
253             i,
254             1,
255             load_kW,
256             test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
257         );

```

```

258
259     load_kW = test_hydro_ptr->commit(
260         i,
261         1,
262         production_kW,
263         load_kW,
264         test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
265     );
266
267     testGreaterThanOrEqualTo(
268         test_hydro_ptr->production_vec_kW[i],
269         0,
270         __FILE__,
271         __LINE__
272     );
273
274     testLessThanOrEqualTo(
275         test_hydro_ptr->production_vec_kW[i],
276         test_hydro_ptr->capacity_kW,
277         __FILE__,
278         __LINE__
279     );
280
281     testFloatEquals(
282         test_hydro_ptr->production_vec_kW[i] -
283         test_hydro_ptr->dispatch_vec_kW[i] -
284         test_hydro_ptr->curtailment_vec_kW[i] -
285         test_hydro_ptr->storage_vec_kW[i],
286         0,
287         __FILE__,
288         __LINE__
289     );
290
291     testGreaterThanOrEqualTo(
292         ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
293         0,
294         __FILE__,
295         __LINE__
296     );
297
298     testLessThanOrEqualTo(
299         ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
300         ((Hydro*)test_hydro_ptr)->maximum_flow_m3hr,
301         __FILE__,
302         __LINE__
303     );
304
305     testGreaterThanOrEqualTo(
306         ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
307         0,
308         __FILE__,
309         __LINE__
310     );
311
312     testLessThanOrEqualTo(
313         ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
314         ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
315         __FILE__,
316         __LINE__
317     );
318 }
319
320 return;
321 } /* testCommit_Hydro() */

```

5.58.2.3 testConstruct_Hydro()

```

Hydro *Noncombustion * testConstruct_Hydro (
    HydroInputs hydro_inputs,
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Hydro](#) object and spot check some post-construction attributes.

Returns

A [Noncombustion](#) pointer to a test [Hydro](#) object.

```

72 {
73     Noncombustion* test_hydro_ptr = new Hydro(
74         8760,
75         1,
76         hydro_inputs,
77         time_vec_hrs_ptr
78     );
79
80     testTruth(
81         not hydro_inputs.noncombustion_inputs.production_inputs.print_flag,
82         __FILE__,
83         __LINE__
84     );
85
86     testFloatEquals(
87         test_hydro_ptr->n_points,
88         8760,
89         __FILE__,
90         __LINE__
91     );
92
93     testFloatEquals(
94         test_hydro_ptr->type,
95         NoncombustionType :: HYDRO,
96         __FILE__,
97         __LINE__
98     );
99
100    testTruth(
101        test_hydro_ptr->type_str == "HYDRO",
102        __FILE__,
103        __LINE__
104    );
105
106    testFloatEquals(
107        ((Hydro*)test_hydro_ptr)->turbine_type,
108        HydroTurbineType :: HYDRO_TURBINE_PELTON,
109        __FILE__,
110        __LINE__
111    );
112
113    testFloatEquals(
114        ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
115        10000,
116        __FILE__,
117        __LINE__
118    );
119
120    return test_hydro_ptr;
121 } /* testConstruct_Hydro() */

```

5.58.2.4 testEfficiencyInterpolation_Hydro()

```

void testEfficiencyInterpolation_Hydro (
    Noncombustion * test_hydro_ptr )

```

Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.

Parameters

<i>test_hydro_ptr</i>	A Noncombustion pointer to the test Hydro object.
-----------------------	---

```

140 {
141     std::vector<double> expected_gen_power_ratios = {
142         0, 0.1, 0.2, 0.3, 0.4, 0.5,
143         0.6, 0.7, 0.75, 0.8, 0.9, 1
144     };
145
146     std::vector<double> expected_gen_efficiencies = {

```



```

147         0.000, 0.800, 0.900, 0.913,
148         0.925, 0.943, 0.947, 0.950,
149         0.953, 0.954, 0.956, 0.958
150     };
151
152     double query = 0;
153     for (size_t i = 0; i < expected_gen_power_ratios.size(); i++) {
154         testFloatEquals(
155             test_hydro_ptr->interpolator.interp_map_1D[
156                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
157             ].x_vec[i],
158             expected_gen_power_ratios[i],
159             __FILE__,
160             __LINE__
161         );
162
163         testFloatEquals(
164             test_hydro_ptr->interpolator.interp_map_1D[
165                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
166             ].y_vec[i],
167             expected_gen_efficiencies[i],
168             __FILE__,
169             __LINE__
170         );
171
172         if (i < expected_gen_power_ratios.size() - 1) {
173             query = expected_gen_power_ratios[i] + ((double)rand() / RAND_MAX) *
174                 (expected_gen_power_ratios[i + 1] - expected_gen_power_ratios[i]);
175
176             test_hydro_ptr->interpolator.interp1D(
177                 HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
178                 query
179             );
180         }
181     }
182
183     std::vector<double> expected_turb_power_ratios = {
184         0, 0.1, 0.2, 0.3, 0.4,
185         0.5, 0.6, 0.7, 0.8, 0.9,
186         1
187     };
188
189     std::vector<double> expected_turb_efficiencies = {
190         0.000, 0.780, 0.855, 0.875, 0.890,
191         0.900, 0.908, 0.913, 0.918, 0.908,
192         0.880
193     };
194
195     for (size_t i = 0; i < expected_turb_power_ratios.size(); i++) {
196         testFloatEquals(
197             test_hydro_ptr->interpolator.interp_map_1D[
198                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
199             ].x_vec[i],
200             expected_turb_power_ratios[i],
201             __FILE__,
202             __LINE__
203         );
204
205         testFloatEquals(
206             test_hydro_ptr->interpolator.interp_map_1D[
207                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
208             ].y_vec[i],
209             expected_turb_efficiencies[i],
210             __FILE__,
211             __LINE__
212         );
213
214         if (i < expected_turb_power_ratios.size() - 1) {
215             query = expected_turb_power_ratios[i] + ((double)rand() / RAND_MAX) *
216                 (expected_turb_power_ratios[i + 1] - expected_turb_power_ratios[i]);
217
218             test_hydro_ptr->interpolator.interp1D(
219                 HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
220                 query
221             );
222         }
223     }
224
225     return;
226 } /* testEfficiencyInterpolation_Hydro() */

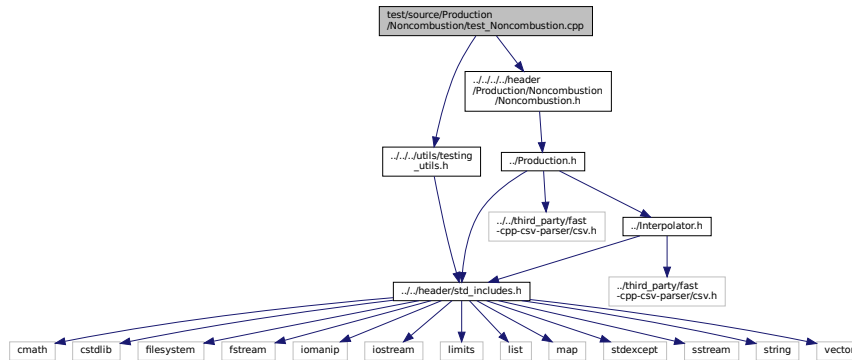
```

5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp

File Reference

Testing suite for [Noncombustion](#) class.

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



Functions

- [Noncombustion](#) * [testConstruct_Noncombustion](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Noncombustion](#) object and spot check some post-construction attributes.
- int [main](#) (int argc, char **argv)

5.59.1 Detailed Description

Testing suite for [Noncombustion](#) class.

A suite of tests for the [Noncombustion](#) class.

5.59.2 Function Documentation

5.59.2.1 main()

```

int main (
    int argc,
    char ** argv )
99 {
100     #ifdef _WIN32
101         activateVirtualTerminal();
102     #endif /* _WIN32 */
103
104     printGold("\tTesting Production <-- Noncombustion");
105
106     srand(time(NULL));
107
108
109     std::vector<double> time_vec_hrs (8760, 0);
110     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
111         time_vec_hrs[i] = i;
112     }
113
114     Noncombustion* test_noncombustion_ptr = testConstruct_Noncombustion(&time_vec_hrs);
115
116     try {
117         //...
118     }
119
120
121
122     catch (...) {
123         delete test_noncombustion_ptr;
124
125         printGold(" ..... ");
126         printRed("FAIL");
127         std::cout << std::endl;
128         throw;
129     }
130
131     delete test_noncombustion_ptr;
132
133     printGold(" ..... ");
134     printGreen("PASS");
135     std::cout << std::endl;
136     return 0;
137
138
139 } /* main() */

```

5.59.2.2 testConstruct_Noncombustion()

```

Noncombustion * testConstruct_Noncombustion (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Noncombustion](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A pointer to a test [Noncombustion](#) object.

```

65 {
66     NoncombustionInputs noncombustion_inputs;
67
68     Noncombustion* test_noncombustion_ptr =
69         new Noncombustion(
70             8760,
71             1,

```

```

72         noncombustion_inputs,
73         time_vec_hrs_ptr
74     );
75
76     testTruth(
77         not noncombustion_inputs.production_inputs.print_flag,
78         __FILE__,
79         __LINE__
80     );
81
82     testFloatEquals(
83         test_noncombustion_ptr->n_points,
84         8760,
85         __FILE__,
86         __LINE__
87     );
88
89     return test_noncombustion_ptr;
90 } /* testConstruct_Noncombustion() */

```

5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference

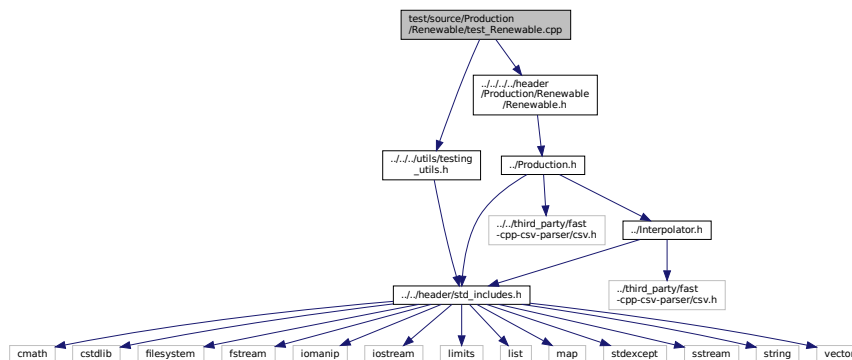
Testing suite for [Renewable](#) class.

```

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

```

Include dependency graph for test_Renewable.cpp:



Functions

- [Renewable](#) * [testConstruct_Renewable](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Renewable](#) object and spot check some post-construction attributes.
- int [main](#) (int argc, char **argv)

5.60.1 Detailed Description

Testing suite for [Renewable](#) class.

A suite of tests for the [Renewable](#) class.

5.60.2 Function Documentation

5.60.2.1 main()

```

int main (
    int argc,
    char ** argv )
98 {
99     #ifdef _WIN32
100         activateVirtualTerminal();
101     #endif /* _WIN32 */
102
103     printGold("\tTesting Production <-- Renewable");
104
105     srand(time(NULL));
106
107
108     std::vector<double> time_vec_hrs (8760, 0);
109     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
110         time_vec_hrs[i] = i;
111     }
112
113     Renewable* test_renewable_ptr = testConstruct_Renewable(&time_vec_hrs);
114
115
116     try {
117         //...
118     }
119
120
121     catch (...) {
122         delete test_renewable_ptr;
123
124         printGold(" ..... ");
125         printRed("FAIL");
126         std::cout << std::endl;
127         throw;
128     }
129
130
131     delete test_renewable_ptr;
132
133     printGold(" ..... ");
134     printGreen("PASS");
135     std::cout << std::endl;
136     return 0;
137
138 } /* main() */

```

5.60.2.2 testConstruct_Renewable()

```

Renewable * testConstruct_Renewable (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Renewable](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A pointer to a test [Renewable](#) object.

```

65 {
66     RenewableInputs renewable_inputs;
67
68     Renewable* test_renewable_ptr = new Renewable(
69         8760,
70         1,
71         renewable_inputs,
72         time_vec_hrs_ptr
73     );
74
75     testTruth(
76         not renewable_inputs.production_inputs.print_flag,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_renewable_ptr->n_points,
83         8760,
84         __FILE__,
85         __LINE__
86     );
87
88     return test_renewable_ptr;
89 } /* testConstruct_Renewable() */

```

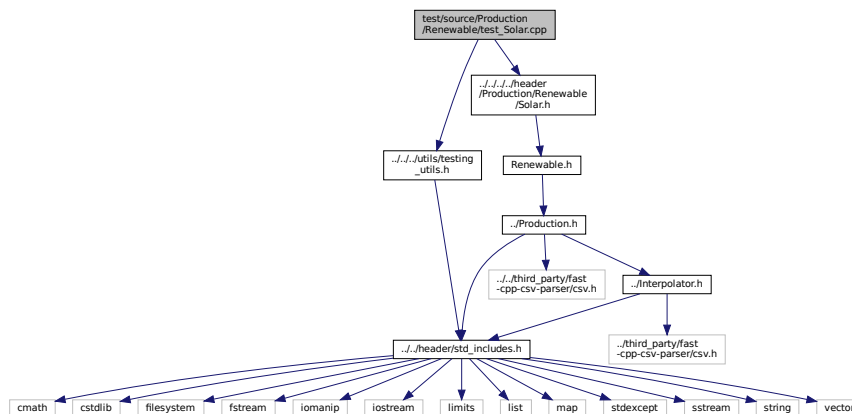
5.61 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for [Solar](#) class.

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

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

Include dependency graph for test_Solar.cpp:



Functions

- [Renewable](#) * [testConstruct_Solar](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Solar](#) object and spot check some post-construction attributes.
- void [testBadConstruct_Solar](#) (std::vector< double > *time_vec_hrs_ptr)
Function to test the trying to construct a [Solar](#) object given bad inputs is being handled as expected.
- void [testProductionOverride_Solar](#) (std::string path_2_normalized_production_time_series, std::vector< double > *time_vec_hrs_ptr)

Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.

- void `testProductionConstraint_Solar` (`Renewable` *test_solar_ptr)

Function to test that the production constraint is active and behaving as expected.

- void `testCommit_Solar` (`Renewable` *test_solar_ptr)

Function to test if the commit method is working as expected, by checking some post-call attributes of the test `Solar` object. Uses a randomized resource input.

- void `testEconomics_Solar` (`Renewable` *test_solar_ptr)
- int `main` (int argc, char **argv)

5.61.1 Detailed Description

Testing suite for `Solar` class.

A suite of tests for the `Solar` class.

5.61.2 Function Documentation

5.61.2.1 `main()`

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

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

    srand(time(NULL));

    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_solar_ptr = testConstruct_Solar(&time_vec_hrs);

    try {
        testBadConstruct_Solar(&time_vec_hrs);

        std::string path_2_normalized_production_time_series =
            "data/test/normalized_production/normalized_solar_production.csv";

        testProductionOverride_Solar(
            path_2_normalized_production_time_series,
            &time_vec_hrs
        );

        testProductionConstraint_Solar(test_solar_ptr);

        testCommit_Solar(test_solar_ptr);
        testEconomics_Solar(test_solar_ptr);
    }

    catch (...) {
        delete test_solar_ptr;

        printGold(" ..... ");
        printRed("FAIL");
    }
}
```

```

506         std::cout << std::endl;
507         throw;
508     }
509
510
511     delete test_solar_ptr;
512
513     printGold(" ..... ");
514     printGreen("PASS");
515     std::cout << std::endl;
516     return 0;
517
518 } /* main() */

```

5.61.2.2 testBadConstruct_Solar()

```

void testBadConstruct_Solar (
    std::vector< double > * time_vec_hrs_ptr )

```

Function to test the trying to construct a [Solar](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```

134 {
135     bool error_flag = true;
136
137     try {
138         SolarInputs bad_solar_inputs;
139         bad_solar_inputs.derating = -1;
140
141         Solar bad_solar(8760, 1, bad_solar_inputs, time_vec_hrs_ptr);
142
143         error_flag = false;
144     } catch (...) {
145         // Task failed successfully! =P
146     }
147     if (not error_flag) {
148         expectedErrorNotDetected(__FILE__, __LINE__);
149     }
150
151     return;
152 } /* testBadConstruct_Solar() */

```

5.61.2.3 testCommit_Solar()

```

void testCommit_Solar (
    Renewable * test_solar_ptr )

```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Solar](#) object. Uses a randomized resource input.

Parameters

<i>test_solar_ptr</i>	A Renewable pointer to the test Solar object.
-----------------------	---

```

314 {
315     std::vector<double> dt_vec_hrs (48, 1);
316
317     std::vector<double> load_vec_kW = {

```



```

318         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
319         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
320         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
321         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
322     };
323
324     double load_kW = 0;
325     double production_kW = 0;
326     double roll = 0;
327     double solar_resource_kWm2 = 0;
328
329     for (int i = 0; i < 48; i++) {
330         roll = (double)rand() / RAND_MAX;
331
332         solar_resource_kWm2 = roll;
333
334         roll = (double)rand() / RAND_MAX;
335
336         if (roll <= 0.1) {
337             solar_resource_kWm2 = 0;
338         }
339
340         else if (roll >= 0.95) {
341             solar_resource_kWm2 = 1.25;
342         }
343
344         roll = (double)rand() / RAND_MAX;
345
346         if (roll >= 0.95) {
347             roll = 1.25;
348         }
349
350         load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
351         load_kW = load_vec_kW[i];
352
353         production_kW = test_solar_ptr->computeProductionkW(
354             i,
355             dt_vec_hrs[i],
356             solar_resource_kWm2
357         );
358
359         load_kW = test_solar_ptr->commit(
360             i,
361             dt_vec_hrs[i],
362             production_kW,
363             load_kW
364         );
365
366         // is running (or not) as expected
367         if (solar_resource_kWm2 > 0) {
368             testTruth(
369                 test_solar_ptr->is_running,
370                 __FILE__,
371                 __LINE__
372             );
373         }
374
375         else {
376             testTruth(
377                 not test_solar_ptr->is_running,
378                 __FILE__,
379                 __LINE__
380             );
381         }
382
383         // load_kW <= load_vec_kW (i.e., after vs before)
384         testLessThanOrEqualTo(
385             load_kW,
386             load_vec_kW[i],
387             __FILE__,
388             __LINE__
389         );
390
391         // production = dispatch + storage + curtailment
392         testFloatEquals(
393             test_solar_ptr->production_vec_kW[i] -
394             test_solar_ptr->dispatch_vec_kW[i] -
395             test_solar_ptr->storage_vec_kW[i] -
396             test_solar_ptr->curtailment_vec_kW[i],
397             0,
398             __FILE__,
399             __LINE__
400         );
401
402         // capacity constraint
403         if (solar_resource_kWm2 > 1) {
404             testFloatEquals(

```

```

405             test_solar_ptr->production_vec_kW[i],
406             test_solar_ptr->capacity_kW,
407             __FILE__,
408             __LINE__
409         );
410     }
411 }
412
413 return;
414 } /* testCommit_Solar() */

```

5.61.2.4 testConstruct_Solar()

```

Renewable * testConstruct_Solar (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Solar](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Renewable](#) pointer to a test [Solar](#) object.

```

65 {
66     SolarInputs solar_inputs;
67
68     Renewable* test_solar_ptr = new Solar(
69         8760,
70         1,
71         solar_inputs,
72         time_vec_hrs_ptr
73     );
74
75     testTruth(
76         not solar_inputs.renewable_inputs.production_inputs.print_flag,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_solar_ptr->n_points,
83         8760,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         test_solar_ptr->type,
90         RenewableType :: SOLAR,
91         __FILE__,
92         __LINE__
93     );
94
95     testTruth(
96         test_solar_ptr->type_str == "SOLAR",
97         __FILE__,
98         __LINE__
99     );
100
101     testFloatEquals(
102         test_solar_ptr->capital_cost,
103         350118.723363,
104         __FILE__,
105         __LINE__
106     );
107
108     testFloatEquals(
109         test_solar_ptr->operation_maintenance_cost_kWh,
110         0.01,

```

```

111     __FILE__,
112     __LINE__
113 );
114
115 return test_solar_ptr;
116 } /* testConstruct_Solar() */

```

5.61.2.5 testEconomics_Solar()

```

void testEconomics_Solar (
    Renewable * test_solar_ptr )
432 {
433     for (int i = 0; i < 48; i++) {
434         // resource, O&M > 0 whenever solar is running (i.e., producing)
435         if (test_solar_ptr->is_running_vec[i]) {
436             testGreaterThan(
437                 test_solar_ptr->operation_maintenance_cost_vec[i],
438                 0,
439                 __FILE__,
440                 __LINE__
441             );
442         }
443
444         // resource, O&M = 0 whenever solar is not running (i.e., not producing)
445         else {
446             testFloatEquals(
447                 test_solar_ptr->operation_maintenance_cost_vec[i],
448                 0,
449                 __FILE__,
450                 __LINE__
451             );
452         }
453     }
454
455     return;
456 } /* testEconomics_Solar() */

```

5.61.2.6 testProductionConstraint_Solar()

```

void testProductionConstraint_Solar (
    Renewable * test_solar_ptr )

```

Function to test that the production constraint is active and behaving as expected.

Parameters

<i>test_solar_ptr</i>	A Renewable pointer to the test Solar object.
-----------------------	---

```

279 {
280     testFloatEquals(
281         test_solar_ptr->computeProductionkW(0, 1, 2),
282         100,
283         __FILE__,
284         __LINE__
285     );
286
287     testFloatEquals(
288         test_solar_ptr->computeProductionkW(0, 1, -1),
289         0,
290         __FILE__,
291         __LINE__
292     );
293
294     return;
295 } /* testProductionConstraint_Solar() */

```

5.61.2.7 testProductionOverride_Solar()

```
void testProductionOverride_Solar (
    std::string path_2_normalized_production_time_series,
    std::vector< double > * time_vec_hrs_ptr )
```

Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.

Parameters

<i>path_2_normalized_production_time_series</i>	A path (either relative or absolute) to the given normalized production time series data.
<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.

```
179 {
180     SolarInputs solar_inputs;
181
182     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
183         path_2_normalized_production_time_series;
184
185     Solar test_solar_override(
186         time_vec_hrs_ptr->size(),
187         1,
188         solar_inputs,
189         time_vec_hrs_ptr
190     );
191
192
193     std::vector<double> expected_normalized_production_vec = {
194         0.916955708517556,
195         0.90947506148393,
196         0.38425267564517,
197         0.191510884037643,
198         0.803361391862077,
199         0.261511294927198,
200         0.221944653883198,
201         0.858495335855501,
202         0.0162863861443092,
203         0.774345409915512,
204         0.354898664149867,
205         0.11158009453439,
206         0.191670176408956,
207         0.0149072402795702,
208         0.30174228469322,
209         0.0815062957850151,
210         0.776404660266821,
211         0.207069187162109,
212         0.518926216750454,
213         0.148538109788597,
214         0.443035200791027,
215         0.62119079547209,
216         0.270792717524391,
217         0.761074879460849,
218         0.0545251308358993,
219         0.0895417089500092,
220         0.21787190761933,
221         0.834403724509682,
222         0.908807953036246,
223         0.815888965292123,
224         0.416663215314571,
225         0.523649705576525,
226         0.490890480401437,
227         0.28317138282312,
228         0.877382682055847,
229         0.14972090597986,
230         0.480161632646382,
231         0.0655830129932816,
232         0.41802666403448,
233         0.48692477737368,
234         0.275957323208066,
235         0.228651250718341,
236         0.574371311550247,
237         0.251872481275769,
```

```

238         0.802697508767121,
239         0.00130607304363551,
240         0.481240172488057,
241         0.702527508293784
242     };
243
244     for (size_t i = 0; i < expected_normalized_production_vec.size(); i++) {
245         testFloatEquals(
246             test_solar_override.normalized_production_vec[i],
247             expected_normalized_production_vec[i],
248             __FILE__,
249             __LINE__
250         );
251
252         testFloatEquals(
253             test_solar_override.computeProductionkW(i, rand(), rand()),
254             test_solar_override.capacity_kW * expected_normalized_production_vec[i],
255             __FILE__,
256             __LINE__
257         );
258     }
259
260     return;
261 } /* testProductionOverride_Solar() */

```

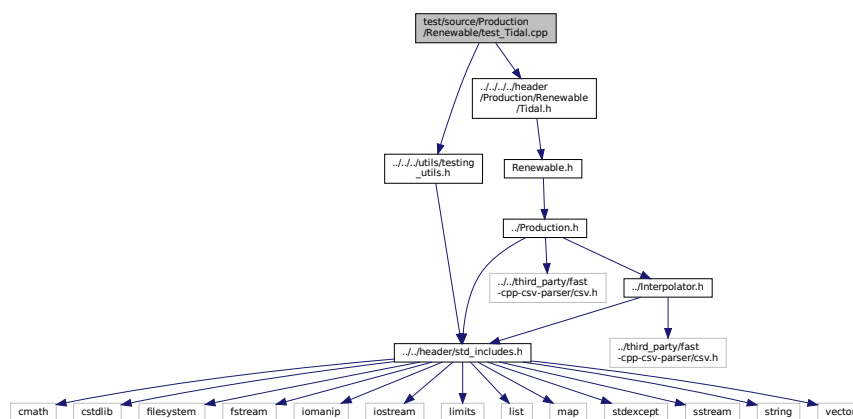
5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference

Testing suite for [Tidal](#) class.

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

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

Include dependency graph for test_Tidal.cpp:



Functions

- [Renewable](#) * [testConstruct_Tidal](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Tidal](#) object and spot check some post-construction attributes.
- void [testBadConstruct_Tidal](#) (std::vector< double > *time_vec_hrs_ptr)
Function to test the trying to construct a [Tidal](#) object given bad inputs is being handled as expected.
- void [testProductionConstraint_Tidal](#) ([Renewable](#) *test_tidal_ptr)
Function to test that the production constraint is active and behaving as expected.
- void [testCommit_Tidal](#) ([Renewable](#) *test_tidal_ptr)
Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Tidal](#) object. Uses a randomized resource input.
- void [testEconomics_Tidal](#) ([Renewable](#) *test_tidal_ptr)
- int [main](#) (int argc, char **argv)

5.62.1 Detailed Description

Testing suite for [Tidal](#) class.

A suite of tests for the [Tidal](#) class.

5.62.2 Function Documentation

5.62.2.1 main()

```
int main (
    int argc,
    char ** argv )
352 {
353     #ifdef _WIN32
354         activateVirtualTerminal();
355     #endif /* _WIN32 */
356
357     printGold("\tTesting Production <-- Renewable <-- Tidal");
358
359     srand(time(NULL));
360
361
362     std::vector<double> time_vec_hrs (8760, 0);
363     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
364         time_vec_hrs[i] = i;
365     }
366
367     Renewable* test_tidal_ptr = testConstruct_Tidal(&time_vec_hrs);
368
369
370     try {
371         testBadConstruct_Tidal(&time_vec_hrs);
372
373         testProductionConstraint_Tidal(test_tidal_ptr);
374
375         testCommit_Tidal(test_tidal_ptr);
376         testEconomics_Tidal(test_tidal_ptr);
377     }
378
379
380     catch (...) {
381         delete test_tidal_ptr;
382
383         printGold(" ..... ");
384         printRed("FAIL");
385         std::cout << std::endl;
386         throw;
387     }
388
389
390     delete test_tidal_ptr;
391
392     printGold(" ..... ");
393     printGreen("PASS");
394     std::cout << std::endl;
395     return 0;
396
397 } /* main() */
```

5.62.2.2 testBadConstruct_Tidal()

```
void testBadConstruct_Tidal (
    std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a [Tidal](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```

129 {
130     bool error_flag = true;
131
132     try {
133         TidalInputs bad_tidal_inputs;
134         bad_tidal_inputs.design_speed_ms = -1;
135
136         Tidal bad_tidal(8760, 1, bad_tidal_inputs, time_vec_hrs_ptr);
137
138         error_flag = false;
139     } catch (...) {
140         // Task failed successfully! =P
141     }
142     if (not error_flag) {
143         expectedErrorNotDetected(__FILE__, __LINE__);
144     }
145
146     return;
147 } /* testBadConstruct_Tidal() */

```

5.62.2.3 testCommit_Tidal()

```

void testCommit_Tidal (
    Renewable * test_tidal_ptr )

```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Tidal](#) object. Uses a randomized resource input.

Parameters

<i>test_tidal_ptr</i>	A Renewable pointer to the test Tidal object.
-----------------------	---

```

211 {
212     std::vector<double> dt_vec_hrs (48, 1);
213
214     std::vector<double> load_vec_kW = {
215         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
216         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
217         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
218         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
219     };
220
221     double load_kW = 0;
222     double production_kW = 0;
223     double roll = 0;
224     double tidal_resource_ms = 0;
225
226     for (int i = 0; i < 48; i++) {
227         roll = (double)rand() / RAND_MAX;
228
229         tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
230
231         roll = (double)rand() / RAND_MAX;
232
233         if (roll <= 0.1) {
234             tidal_resource_ms = 0;
235         }
236
237         else if (roll >= 0.95) {
238             tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
239         }
240
241         roll = (double)rand() / RAND_MAX;
242
243         if (roll >= 0.95) {
244             roll = 1.25;
245         }
246

```

```

247     load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
248     load_kW = load_vec_kW[i];
249
250     production_kW = test_tidal_ptr->computeProductionkW(
251         i,
252         dt_vec_hrs[i],
253         tidal_resource_ms
254     );
255
256     load_kW = test_tidal_ptr->commit(
257         i,
258         dt_vec_hrs[i],
259         production_kW,
260         load_kW
261     );
262
263     // is running (or not) as expected
264     if (production_kW > 0) {
265         testTruth(
266             test_tidal_ptr->is_running,
267             __FILE__,
268             __LINE__
269         );
270     }
271
272     else {
273         testTruth(
274             not test_tidal_ptr->is_running,
275             __FILE__,
276             __LINE__
277         );
278     }
279
280     // load_kW <= load_vec_kW (i.e., after vs before)
281     testLessThanOrEqualTo(
282         load_kW,
283         load_vec_kW[i],
284         __FILE__,
285         __LINE__
286     );
287
288     // production = dispatch + storage + curtailment
289     testFloatEquals(
290         test_tidal_ptr->production_vec_kW[i] -
291         test_tidal_ptr->dispatch_vec_kW[i] -
292         test_tidal_ptr->storage_vec_kW[i] -
293         test_tidal_ptr->curtailment_vec_kW[i],
294         0,
295         __FILE__,
296         __LINE__
297     );
298 }
299
300 return;
301 } /* testCommit_Tidal() */

```

5.62.2.4 testConstruct_Tidal()

```

Renewable * testConstruct_Tidal (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Tidal](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Renewable](#) pointer to a test [Tidal](#) object.

```

65 {

```



```

66     TidalInputs tidal_inputs;
67
68     Renewable* test_tidal_ptr = new Tidal(8760, 1, tidal_inputs, time_vec_hrs_ptr);
69
70     testTruth(
71         not tidal_inputs.renewable_inputs.production_inputs.print_flag,
72         __FILE__,
73         __LINE__
74     );
75
76     testFloatEquals(
77         test_tidal_ptr->n_points,
78         8760,
79         __FILE__,
80         __LINE__
81     );
82
83     testFloatEquals(
84         test_tidal_ptr->type,
85         RenewableType :: TIDAL,
86         __FILE__,
87         __LINE__
88     );
89
90     testTruth(
91         test_tidal_ptr->type_str == "TIDAL",
92         __FILE__,
93         __LINE__
94     );
95
96     testFloatEquals(
97         test_tidal_ptr->capital_cost,
98         500237.446725,
99         __FILE__,
100        __LINE__
101    );
102
103    testFloatEquals(
104        test_tidal_ptr->operation_maintenance_cost_kWh,
105        0.069905,
106        __FILE__,
107        __LINE__
108    );
109
110    return test_tidal_ptr;
111 } /* testConstruct_Tidal() */

```

5.62.2.5 testEconomics_Tidal()

```

void testEconomics_Tidal (
    Renewable * test_tidal_ptr )
319 {
320     for (int i = 0; i < 48; i++) {
321         // resource, O&M > 0 whenever tidal is running (i.e., producing)
322         if (test_tidal_ptr->is_running_vec[i]) {
323             testGreaterThan(
324                 test_tidal_ptr->operation_maintenance_cost_vec[i],
325                 0,
326                 __FILE__,
327                 __LINE__
328             );
329         }
330
331         // resource, O&M = 0 whenever tidal is not running (i.e., not producing)
332         else {
333             testFloatEquals(
334                 test_tidal_ptr->operation_maintenance_cost_vec[i],
335                 0,
336                 __FILE__,
337                 __LINE__
338             );
339         }
340     }
341
342     return;
343 } /* testEconomics_Tidal() */

```

5.62.2.6 testProductionConstraint_Tidal()

```
void testProductionConstraint_Tidal (
    Renewable * test_tidal_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

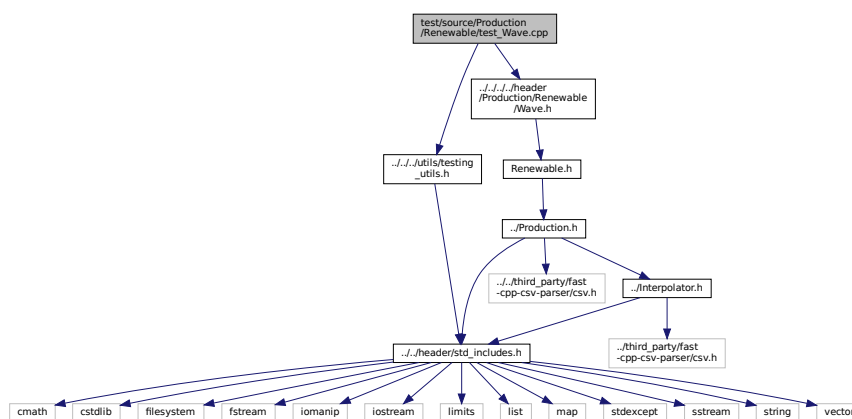
<code>test_tidal_ptr</code>	A Renewable pointer to the test Tidal object.
-----------------------------	---

```
165 {
166     testFloatEquals(
167         test_tidal_ptr->computeProductionkW(0, 1, 1e6),
168         0,
169         __FILE__,
170         __LINE__
171     );
172
173     testFloatEquals(
174         test_tidal_ptr->computeProductionkW(
175             0,
176             1,
177             ((Tidal*)test_tidal_ptr)->design_speed_ms
178         ),
179         test_tidal_ptr->capacity_kW,
180         __FILE__,
181         __LINE__
182     );
183
184     testFloatEquals(
185         test_tidal_ptr->computeProductionkW(0, 1, -1),
186         0,
187         __FILE__,
188         __LINE__
189     );
190
191     return;
192 } /* testProductionConstraint_Tidal() */
```

5.63 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

- [Renewable * testConstruct_Wave](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Wave](#) object and spot check some post-construction attributes.
- [Renewable * testConstructLookup_Wave](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Wave](#) object using production lookup.
- void [testBadConstruct_Wave](#) (std::vector< double > *time_vec_hrs_ptr)
Function to test the trying to construct a [Wave](#) object given bad inputs is being handled as expected.
- void [testProductionConstraint_Wave](#) ([Renewable](#) *test_wave_ptr)
Function to test that the production constraint is active and behaving as expected.
- void [testCommit_Wave](#) ([Renewable](#) *test_wave_ptr)
Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Wave](#) object. Uses a randomized resource input.
- void [testEconomics_Wave](#) ([Renewable](#) *test_wave_ptr)
- void [testProductionLookup_Wave](#) ([Renewable](#) *test_wave_lookup_ptr)
Function to test that production lookup (i.e., interpolation) is returning the expected values.
- int [main](#) (int argc, char **argv)

5.63.1 Detailed Description

Testing suite for [Wave](#) class.

A suite of tests for the [Wave](#) class.

5.63.2 Function Documentation

5.63.2.1 main()

```
int main (
    int argc,
    char ** argv )
467 {
468     #ifdef _WIN32
469         activateVirtualTerminal();
470     #endif /* _WIN32 */
471
472     printGold("\tTesting Production <-- Renewable <-- Wave");
473
474     srand(time(NULL));
475
476
477     std::vector<double> time_vec_hrs (8760, 0);
478     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
479         time_vec_hrs[i] = i;
480     }
481
482     Renewable* test_wave_ptr = testConstruct_Wave(&time_vec_hrs);
483     Renewable* test_wave_lookup_ptr = testConstructLookup_Wave(&time_vec_hrs);
484
485
486     try {
487         testBadConstruct_Wave(&time_vec_hrs);
488
489         testProductionConstraint_Wave(test_wave_ptr);
490
491         testCommit_Wave(test_wave_ptr);
492         testEconomics_Wave(test_wave_ptr);
493     }
```

```

494     testProductionLookup_Wave(test_wave_lookup_ptr);
495 }
496
497
498 catch (...) {
499     delete test_wave_ptr;
500     delete test_wave_lookup_ptr;
501
502     printGold(" ..... ");
503     printRed("FAIL");
504     std::cout << std::endl;
505     throw;
506 }
507
508
509 delete test_wave_ptr;
510 delete test_wave_lookup_ptr;
511
512 printGold(" ..... ");
513 printGreen("PASS");
514 std::cout << std::endl;
515 return 0;
516
517 } /* main() */

```

5.63.2.2 testBadConstruct_Wave()

```

void testBadConstruct_Wave (
    std::vector< double > * time_vec_hrs_ptr )

```

Function to test the trying to construct a [Wave](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```

158 {
159     bool error_flag = true;
160
161     try {
162         WaveInputs bad_wave_inputs;
163         bad_wave_inputs.design_significant_wave_height_m = -1;
164
165         Wave bad_wave(8760, 1, bad_wave_inputs, time_vec_hrs_ptr);
166
167         error_flag = false;
168     } catch (...) {
169         // Task failed successfully! =P
170     }
171     if (not error_flag) {
172         expectedErrorNotDetected(__FILE__, __LINE__);
173     }
174
175     return;
176 } /* testBadConstruct_Wave() */

```

5.63.2.3 testCommit_Wave()

```

void testCommit_Wave (
    Renewable * test_wave_ptr )

```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Wave](#) object. Uses a randomized resource input.

Parameters

<code>test_wave_ptr</code>	A Renewable pointer to the test Wave object.
----------------------------	--

```

229 {
230     std::vector<double> dt_vec_hrs (48, 1);
231
232     std::vector<double> load_vec_kW = {
233         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
234         1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0,
235         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
236         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
237     };
238
239     double load_kW = 0;
240     double production_kW = 0;
241     double roll = 0;
242     double significant_wave_height_m = 0;
243     double energy_period_s = 0;
244
245     for (int i = 0; i < 48; i++) {
246         roll = (double)rand() / RAND_MAX;
247
248         if (roll <= 0.05) {
249             roll = 0;
250         }
251
252         significant_wave_height_m = roll *
253             ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
254
255         roll = (double)rand() / RAND_MAX;
256
257         if (roll <= 0.05) {
258             roll = 0;
259         }
260
261         energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
262
263         roll = (double)rand() / RAND_MAX;
264
265         if (roll >= 0.95) {
266             roll = 1.25;
267         }
268
269         load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
270         load_kW = load_vec_kW[i];
271
272         production_kW = test_wave_ptr->computeProductionkW(
273             i,
274             dt_vec_hrs[i],
275             significant_wave_height_m,
276             energy_period_s
277         );
278
279         load_kW = test_wave_ptr->commit(
280             i,
281             dt_vec_hrs[i],
282             production_kW,
283             load_kW
284         );
285
286         // is running (or not) as expected
287         if (production_kW > 0) {
288             testTruth(
289                 test_wave_ptr->is_running,
290                 __FILE__,
291                 __LINE__
292             );
293         }
294
295         else {
296             testTruth(
297                 not test_wave_ptr->is_running,
298                 __FILE__,
299                 __LINE__
300             );
301         }
302
303         // load_kW <= load_vec_kW (i.e., after vs before)
304         testLessThanOrEqualTo(
305             load_kW,
306             load_vec_kW[i],
307             __FILE__,
308             __LINE__
309         );
310     }

```

```

311         // production = dispatch + storage + curtailment
312         testFloatEquals(
313             test_wave_ptr->production_vec_kW[i] -
314             test_wave_ptr->dispatch_vec_kW[i] -
315             test_wave_ptr->storage_vec_kW[i] -
316             test_wave_ptr->curtailment_vec_kW[i],
317             0,
318             __FILE__,
319             __LINE__
320         );
321     }
322
323     return;
324 } /* testCommit_Wave() */

```

5.63.2.4 testConstruct_Wave()

```

Renewable * testConstruct_Wave (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Wave](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Renewable](#) pointer to a test [Wave](#) object.

```

65 {
66     WaveInputs wave_inputs;
67
68     Renewable* test_wave_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);
69
70     testTruth(
71         not wave_inputs.renewable_inputs.production_inputs.print_flag,
72         __FILE__,
73         __LINE__
74     );
75
76     testFloatEquals(
77         test_wave_ptr->n_points,
78         8760,
79         __FILE__,
80         __LINE__
81     );
82
83     testFloatEquals(
84         test_wave_ptr->type,
85         RenewableType :: WAVE,
86         __FILE__,
87         __LINE__
88     );
89
90     testTruth(
91         test_wave_ptr->type_str == "WAVE",
92         __FILE__,
93         __LINE__
94     );
95
96     testFloatEquals(
97         test_wave_ptr->capital_cost,
98         850831.063539,
99         __FILE__,
100        __LINE__
101    );
102
103    testFloatEquals(
104        test_wave_ptr->operation_maintenance_cost_kWh,
105        0.069905,
106        __FILE__,

```

```

107     __LINE__
108 );
109
110 return test_wave_ptr;
111 } /* testConstruct_Wave() */

```

5.63.2.5 testConstructLookup_Wave()

```

Renewable * testConstructLookup_Wave (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Wave](#) object using production lookup.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Renewable](#) pointer to a test [Wave](#) object.

```

130 {
131     WaveInputs wave_inputs;
132
133     wave_inputs.power_model = WavePowerProductionModel :: WAVE_POWER_LOOKUP;
134     wave_inputs.path_2_normalized_performance_matrix =
135         "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
136
137     Renewable* test_wave_lookup_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);
138
139     return test_wave_lookup_ptr;
140 } /* testConstructLookup_Wave() */

```

5.63.2.6 testEconomics_Wave()

```

void testEconomics_Wave (
    Renewable * test_wave_ptr )
{
    for (int i = 0; i < 48; i++) {
        // resource, O&M > 0 whenever wave is running (i.e., producing)
        if (test_wave_ptr->is_running_vec[i]) {
            testGreaterThan(
                test_wave_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
        // resource, O&M = 0 whenever wave is not running (i.e., not producing)
        else {
            testFloatEquals(
                test_wave_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }
    return;
} /* testEconomics_Wave() */

```

5.63.2.7 testProductionConstraint_Wave()

```
void testProductionConstraint_Wave (
    Renewable * test_wave_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

<i>test_wave_ptr</i>	A Renewable pointer to the test Wave object.
----------------------	--

```
194 {
195     testFloatEquals(
196         test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
197         0,
198         __FILE__,
199         __LINE__
200     );
201
202     testFloatEquals(
203         test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
204         0,
205         __FILE__,
206         __LINE__
207     );
208
209     return;
210 } /* testProductionConstraint_Wave() */
```

5.63.2.8 testProductionLookup_Wave()

```
void testProductionLookup_Wave (
    Renewable * test_wave_lookup_ptr )
```

Function to test that production lookup (i.e., interpolation) is returning the expected values.

Parameters

<i>test_wave_lookup_ptr</i>	A Renewable pointer to the test Wave object using production lookup.
-----------------------------	--

```
385 {
386     std::vector<double> significant_wave_height_vec_m = {
387         0.389211848822208,
388         0.836477431896843,
389         1.52738334015579,
390         1.92640601114508,
391         2.27297317532019,
392         2.87416589636605,
393         3.72275770908175,
394         3.95063175885536,
395         4.68097139867404,
396         4.97775020449812,
397         5.55184219980547,
398         6.06566629451658,
399         6.27927876785062,
400         6.96218133671013,
401         7.51754442460228
402     };
403
404     std::vector<double> energy_period_vec_s = {
405         5.45741899698926,
406         6.00101329139007,
407         7.50567689404182,
408         8.77681262912881,
409         9.45143678206774,
410         10.7767876462885,
411         11.4795760857165,
412         12.9430684577599,
```



```

413         13.303544885703,
414         14.5069863517863,
415         15.1487890438045,
416         16.086524049077,
417         17.176609978648,
418         18.4155153740256,
419         19.1704554940162
420     };
421
422     std::vector<std::vector<double>> expected_normalized_performance_matrix = {
423
424 {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
425 {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
426 {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
427 {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
428 {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
429 {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
430 {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
431 {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
432 {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
433 {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
434 {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
435 {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
436 {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
437 {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
438 {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
439     };
440     for (size_t i = 0; i < energy_period_vec_s.size(); i++) {
441         for (size_t j = 0; j < significant_wave_height_vec_m.size(); j++) {
442             testFloatEquals(
443                 test_wave_lookup_ptr->computeProductionkW(
444                     0,
445                     1,
446                     significant_wave_height_vec_m[j],
447                     energy_period_vec_s[i]
448                 ),
449                 expected_normalized_performance_matrix[i][j] *
450                 test_wave_lookup_ptr->capacity_kW,
451                 __FILE__,
452                 __LINE__
453             );
454         }
455     }
456     return;
457 } /* testProductionLookup_Wave() */
458 }

```

5.64 test/source/Production/Renewable/test_Wind.cpp File Reference

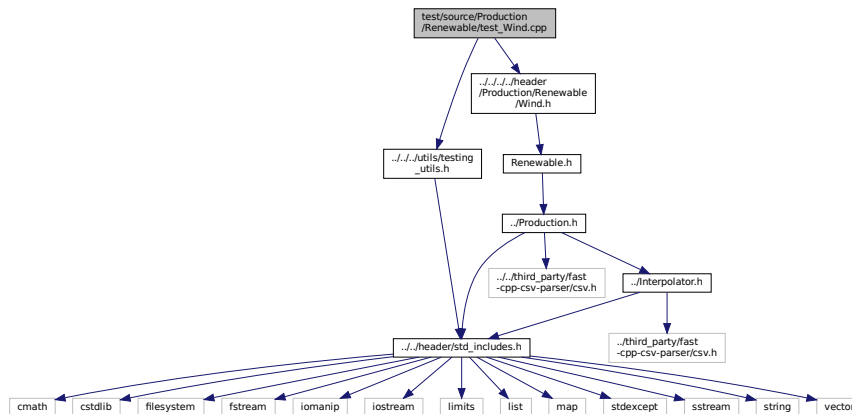
Testing suite for [Wind](#) class.

```

#include ".../.../utils/testing_utils.h"
#include ".../.../.../header/Production/Renewable/Wind.h"

```

Include dependency graph for test_Wind.cpp:



Functions

- `Renewable * testConstruct_Wind (std::vector< double > *time_vec_hrs_ptr)`
A function to construct a [Wind](#) object and spot check some post-construction attributes.
- `void testBadConstruct_Wind (std::vector< double > *time_vec_hrs_ptr)`
Function to test the trying to construct a [Wind](#) object given bad inputs is being handled as expected.
- `void testProductionConstraint_Wind (Renewable *test_wind_ptr)`
Function to test that the production constraint is active and behaving as expected.
- `void testCommit_Wind (Renewable *test_wind_ptr)`
Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Wind](#) object. Uses a randomized resource input.
- `void testEconomics_Wind (Renewable *test_wind_ptr)`
- `int main (int argc, char **argv)`

5.64.1 Detailed Description

Testing suite for [Wind](#) class.

A suite of tests for the [Wind](#) class.

5.64.2 Function Documentation

5.64.2.1 main()

```

int main (
    int argc,
    char ** argv )
352 {
353     #ifdef _WIN32
354         activateVirtualTerminal();
355     #endif /* _WIN32 */
356
357     printGold("\tTesting Production <-- Renewable <-- Wind");
358
359     srand(time(NULL));
360
361
362     std::vector<double> time_vec_hrs (8760, 0);
363     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
364         time_vec_hrs[i] = i;
365     }
366
367     Renewable* test_wind_ptr = testConstruct_Wind(&time_vec_hrs);
368
369     try {
370         testBadConstruct_Wind(&time_vec_hrs);
371
372         testProductionConstraint_Wind(test_wind_ptr);
373
374         testCommit_Wind(test_wind_ptr);
375         testEconomics_Wind(test_wind_ptr);
376     }
377
378     catch (...) {
379         delete test_wind_ptr;
380
381         printGold(" ..... ");
382         printRed("FAIL");
383         std::cout << std::endl;
384         throw;
385     }
386
387     delete test_wind_ptr;
388
389     printGold(" ..... ");
390     printGreen("PASS");
391     std::cout << std::endl;
392     return 0;
393 } /* main() */

```

5.64.2.2 testBadConstruct_Wind()

```

void testBadConstruct_Wind (
    std::vector< double > * time_vec_hrs_ptr )

```

Function to test the trying to construct a [Wind](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```

129 {
130     bool error_flag = true;
131
132     try {
133         WindInputs bad_wind_inputs;
134         bad_wind_inputs.design_speed_ms = -1;
135
136         Wind bad_wind(8760, 1, bad_wind_inputs, time_vec_hrs_ptr);
137

```

```

138         error_flag = false;
139     } catch (...) {
140         // Task failed successfully! =P
141     }
142     if (not error_flag) {
143         expectedErrorNotDetected(__FILE__, __LINE__);
144     }
145
146     return;
147 } /* testBadConstruct_Wind() */

```

5.64.2.3 testCommit_Wind()

```

void testCommit_Wind (
    Renewable * test_wind_ptr )

```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test [Wind](#) object. Uses a randomized resource input.

Parameters

<i>test_wind_ptr</i>	A Renewable pointer to the test Wind object.
----------------------	--

```

211 {
212     std::vector<double> dt_vec_hrs (48, 1);
213
214     std::vector<double> load_vec_kW = {
215         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
216         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
217         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
218         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
219     };
220
221     double load_kW = 0;
222     double production_kW = 0;
223     double roll = 0;
224     double wind_resource_ms = 0;
225
226     for (int i = 0; i < 48; i++) {
227         roll = (double)rand() / RAND_MAX;
228
229         wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
230
231         roll = (double)rand() / RAND_MAX;
232
233         if (roll <= 0.1) {
234             wind_resource_ms = 0;
235         }
236
237         else if (roll >= 0.95) {
238             wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
239         }
240
241         roll = (double)rand() / RAND_MAX;
242
243         if (roll >= 0.95) {
244             roll = 1.25;
245         }
246
247         load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
248         load_kW = load_vec_kW[i];
249
250         production_kW = test_wind_ptr->computeProductionkW(
251             i,
252             dt_vec_hrs[i],
253             wind_resource_ms
254         );
255
256         load_kW = test_wind_ptr->commit(
257             i,
258             dt_vec_hrs[i],
259             production_kW,
260             load_kW
261         );

```

```

262
263     // is running (or not) as expected
264     if (production_kW > 0) {
265         testTruth(
266             test_wind_ptr->is_running,
267             __FILE__,
268             __LINE__
269         );
270     }
271
272     else {
273         testTruth(
274             not test_wind_ptr->is_running,
275             __FILE__,
276             __LINE__
277         );
278     }
279
280     // load_kW <= load_vec_kW (i.e., after vs before)
281     testLessThanOrEqualTo(
282         load_kW,
283         load_vec_kW[i],
284         __FILE__,
285         __LINE__
286     );
287
288     // production = dispatch + storage + curtailment
289     testFloatEquals(
290         test_wind_ptr->production_vec_kW[i] -
291         test_wind_ptr->dispatch_vec_kW[i] -
292         test_wind_ptr->storage_vec_kW[i] -
293         test_wind_ptr->curtailment_vec_kW[i],
294         0,
295         __FILE__,
296         __LINE__
297     );
298 }
299
300 return;
301 } /* testCommit_Wind() */

```

5.64.2.4 testConstruct_Wind()

```

Renewable * testConstruct_Wind (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Wind](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A [Renewable](#) pointer to a test [Wind](#) object.

```

65 {
66     WindInputs wind_inputs;
67
68     Renewable* test_wind_ptr = new Wind(8760, 1, wind_inputs, time_vec_hrs_ptr);
69
70     testTruth(
71         not wind_inputs.renewable_inputs.production_inputs.print_flag,
72         __FILE__,
73         __LINE__
74     );
75
76     testFloatEquals(
77         test_wind_ptr->n_points,
78         8760,
79         __FILE__,
80         __LINE__

```

```

81     );
82
83     testFloatEquals(
84         test_wind_ptr->type,
85         RenewableType :: WIND,
86         __FILE__,
87         __LINE__
88     );
89
90     testTruth(
91         test_wind_ptr->type_str == "WIND",
92         __FILE__,
93         __LINE__
94     );
95
96     testFloatEquals(
97         test_wind_ptr->capital_cost,
98         450356.170088,
99         __FILE__,
100        __LINE__
101    );
102
103    testFloatEquals(
104        test_wind_ptr->operation_maintenance_cost_kWh,
105        0.034953,
106        __FILE__,
107        __LINE__
108    );
109
110    return test_wind_ptr;
111 } /* testConstruct_Wind() */

```

5.64.2.5 testEconomics_Wind()

```

void testEconomics_Wind (
    Renewable * test_wind_ptr )
{
    319 {
    320     for (int i = 0; i < 48; i++) {
    321         // resource, O&M > 0 whenever wind is running (i.e., producing)
    322         if (test_wind_ptr->is_running_vec[i]) {
    323             testGreaterThan(
    324                 test_wind_ptr->operation_maintenance_cost_vec[i],
    325                 0,
    326                 __FILE__,
    327                 __LINE__
    328             );
    329         }
    330
    331         // resource, O&M = 0 whenever wind is not running (i.e., not producing)
    332         else {
    333             testFloatEquals(
    334                 test_wind_ptr->operation_maintenance_cost_vec[i],
    335                 0,
    336                 __FILE__,
    337                 __LINE__
    338             );
    339         }
    340     }
    341
    342     return;
    343 } /* testEconomics_Wind() */

```

5.64.2.6 testProductionConstraint_Wind()

```

void testProductionConstraint_Wind (
    Renewable * test_wind_ptr )

```

Function to test that the production constraint is active and behaving as expected.

Parameters

<code>test_wind_ptr</code>	A Renewable pointer to the test Wind object.
----------------------------	--

```

165 {
166     testFloatEquals(
167         test_wind_ptr->computeProductionkW(0, 1, 1e6),
168         0,
169         __FILE__,
170         __LINE__
171     );
172
173     testFloatEquals(
174         test_wind_ptr->computeProductionkW(
175             0,
176             1,
177             ((Wind*)test_wind_ptr)->design_speed_ms
178         ),
179         test_wind_ptr->capacity_kW,
180         __FILE__,
181         __LINE__
182     );
183
184     testFloatEquals(
185         test_wind_ptr->computeProductionkW(0, 1, -1),
186         0,
187         __FILE__,
188         __LINE__
189     );
190
191     return;
192 } /* testProductionConstraint_Wind() */

```

5.65 test/source/Production/test_Production.cpp File Reference

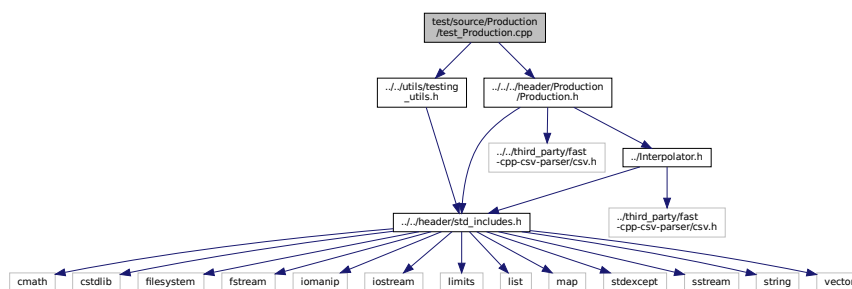
Testing suite for [Production](#) class.

```

#include "../utils/testing_utils.h"
#include "../../../../header/Production/Production.h"

```

Include dependency graph for test_Production.cpp:



Functions

- [Production](#) * [testConstruct_Production](#) (std::vector< double > *time_vec_hrs_ptr)
A function to construct a [Production](#) object and spot check some post-construction attributes.
- void [testBadConstruct_Production](#) (std::vector< double > *time_vec_hrs_ptr)
Function to test the trying to construct a [Production](#) object given bad inputs is being handled as expected.
- int [main](#) (int argc, char **argv)

5.65.1 Detailed Description

Testing suite for [Production](#) class.

A suite of tests for the [Production](#) class.

5.65.2 Function Documentation

5.65.2.1 main()

```

int main (
    int argc,
    char ** argv )
203 {
204     #ifdef _WIN32
205         activateVirtualTerminal();
206     #endif /* _WIN32 */
207
208     printGold("\tTesting Production");
209
210     srand(time(NULL));
211
212
213     std::vector<double> time_vec_hrs (8760, 0);
214     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
215         time_vec_hrs[i] = i;
216     }
217
218     Production* test_production_ptr = testConstruct_Production(&time_vec_hrs);
219
220
221     try {
222         testBadConstruct_Production(&time_vec_hrs);
223     }
224
225
226     catch (...) {
227         delete test_production_ptr;
228
229         printGold(" ..... ");
230         printRed("FAIL");
231         std::cout << std::endl;
232         throw;
233     }
234
235
236     delete test_production_ptr;
237
238     printGold(" ..... ");
239     printGreen("PASS");
240     std::cout << std::endl;
241     return 0;
242
243 } /* main() */

```

5.65.2.2 testBadConstruct_Production()

```

void testBadConstruct_Production (
    std::vector< double > * time_vec_hrs_ptr )

```

Function to test the trying to construct a [Production](#) object given bad inputs is being handled as expected.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

```

177 {
178     bool error_flag = true;
179
180     try {
181         ProductionInputs production_inputs;
182
183         Production bad_production(0, 1, production_inputs, time_vec_hrs_ptr);
184
185         error_flag = false;
186     } catch (...) {
187         // Task failed successfully! =P
188     }
189     if (not error_flag) {
190         expectedErrorNotDetected(__FILE__, __LINE__);
191     }
192
193     return;
194 } /* testBadConstruct_Production() */

```

5.65.2.3 testConstruct_Production()

```

Production * testConstruct_Production (
    std::vector< double > * time_vec_hrs_ptr )

```

A function to construct a [Production](#) object and spot check some post-construction attributes.

Parameters

<i>time_vec_hrs_ptr</i>	A pointer to the vector containing the modelling time series.
-------------------------	---

Returns

A pointer to a test [Production](#) object.

```

65 {
66     ProductionInputs production_inputs;
67
68     Production* test_production_ptr = new Production(
69         8760,
70         1,
71         production_inputs,
72         time_vec_hrs_ptr
73     );
74
75     testTruth(
76         not production_inputs.print_flag,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         production_inputs.nominal_inflation_annual,
83         0.02,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         production_inputs.nominal_discount_annual,
90         0.04,
91         __FILE__,
92         __LINE__
93     );
94
95     testFloatEquals(
96         test_production_ptr->n_points,

```

```

97         8760,
98         __FILE__,
99         __LINE__
100    );
101
102    testFloatEquals (
103        test_production_ptr->capacity_kW,
104        100,
105        __FILE__,
106        __LINE__
107    );
108
109    testFloatEquals (
110        test_production_ptr->real_discount_annual,
111        0.0196078431372549,
112        __FILE__,
113        __LINE__
114    );
115
116    testFloatEquals (
117        test_production_ptr->production_vec_kW.size(),
118        8760,
119        __FILE__,
120        __LINE__
121    );
122
123    testFloatEquals (
124        test_production_ptr->dispatch_vec_kW.size(),
125        8760,
126        __FILE__,
127        __LINE__
128    );
129
130    testFloatEquals (
131        test_production_ptr->storage_vec_kW.size(),
132        8760,
133        __FILE__,
134        __LINE__
135    );
136
137    testFloatEquals (
138        test_production_ptr->curtailment_vec_kW.size(),
139        8760,
140        __FILE__,
141        __LINE__
142    );
143
144    testFloatEquals (
145        test_production_ptr->capital_cost_vec.size(),
146        8760,
147        __FILE__,
148        __LINE__
149    );
150
151    testFloatEquals (
152        test_production_ptr->operation_maintenance_cost_vec.size(),
153        8760,
154        __FILE__,
155        __LINE__
156    );
157
158    return test_production_ptr;
159 } /* testConstruct_Production() */

```

5.66 test/source/Storage/test_Lilon.cpp File Reference

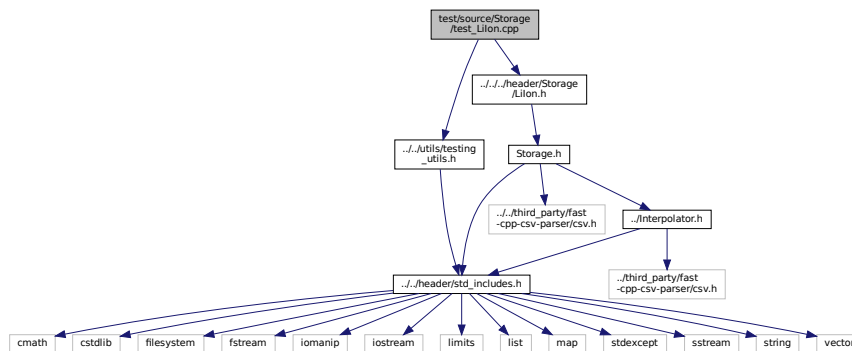
Testing suite for [Lilon](#) class.

```

#include "../utils/testing_utils.h"
#include "../header/Storage/LiIon.h"

```

Include dependency graph for test_Lilon.cpp:



Functions

- [Storage * testConstruct_Lilon](#) (void)
A function to construct a [Lilon](#) object and spot check some post-construction attributes.
- void [testBadConstruct_Lilon](#) (void)
Function to test the trying to construct a [Lilon](#) object given bad inputs is being handled as expected.
- void [testCommitCharge_Lilon](#) (Storage *test_liion_ptr)
A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.
- void [testCommitDischarge_Lilon](#) (Storage *test_liion_ptr)
A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.
- int [main](#) (int argc, char **argv)

5.66.1 Detailed Description

Testing suite for [Lilon](#) class.

A suite of tests for the [Lilon](#) class.

5.66.2 Function Documentation

5.66.2.1 main()

```

int main (
    int argc,
    char ** argv )
331 {
332     #ifdef _WIN32
333         activateVirtualTerminal();
334     #endif /* _WIN32 */
335
336     printGold("\tTesting Storage <-- LiIon");
337
338     srand(time(NULL));
339

```

```

340
341     Storage* test_liion_ptr = testConstruct_LiIon();
342
343
344     try {
345         testBadConstruct_LiIon();
346
347         testCommitCharge_LiIon(test_liion_ptr);
348         testCommitDischarge_LiIon(test_liion_ptr);
349     }
350
351
352     catch (...) {
353         delete test_liion_ptr;
354
355         printGold(" ..... ");
356         printRed("FAIL");
357         std::cout << std::endl;
358         throw;
359     }
360
361
362     delete test_liion_ptr;
363
364     printGold(" ..... ");
365     printGreen("PASS");
366     std::cout << std::endl;
367     return 0;
368
369 } /* main() */

```

5.66.2.2 testBadConstruct_LiIon()

```

void testBadConstruct_LiIon (
    void )

```

Function to test the trying to construct a [LiIon](#) object given bad inputs is being handled as expected.

```

174 {
175     bool error_flag = true;
176
177     try {
178         LiIonInputs bad_liion_inputs;
179         bad_liion_inputs.min_SOC = -1;
180
181         LiIon bad_liion(8760, 1, bad_liion_inputs);
182
183         error_flag = false;
184     } catch (...) {
185         // Task failed successfully! =P
186     }
187     if (not error_flag) {
188         expectedErrorNotDetected(__FILE__, __LINE__);
189     }
190
191     return;
192 } /* testBadConstruct_LiIon() */

```

5.66.2.3 testCommitCharge_LiIon()

```

void testCommitCharge_LiIon (
    Storage * test_liion_ptr )

```

A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.

Parameters

<i>test_liion_ptr</i>	A Storage pointer to a test LiIon object.
-----------------------	---

```

210 {
211     double dt_hrs = 1;
212
213     testFloatEquals(
214         test_liion_ptr->getAvailablekW(dt_hrs),
215         100, // hits power capacity constraint
216         __FILE__,
217         __LINE__
218     );
219
220     testFloatEquals(
221         test_liion_ptr->getAcceptablekW(dt_hrs),
222         100, // hits power capacity constraint
223         __FILE__,
224         __LINE__
225     );
226
227     test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing in
228
229     testFloatEquals(
230         test_liion_ptr->getAvailablekW(dt_hrs),
231         0, // is already hitting power capacity constraint
232         __FILE__,
233         __LINE__
234     );
235
236     testFloatEquals(
237         test_liion_ptr->getAcceptablekW(dt_hrs),
238         0, // is already hitting power capacity constraint
239         __FILE__,
240         __LINE__
241     );
242
243     test_liion_ptr->commitCharge(0, dt_hrs, 100);
244
245     testFloatEquals(
246         test_liion_ptr->power_kW,
247         0,
248         __FILE__,
249         __LINE__
250     );
251
252     return;
253 } /* testCommitCharge_LiIon() */

```

5.66.2.4 testCommitDischarge_LiIon()

```

void testCommitDischarge_LiIon (
    Storage * test_liion_ptr )

```

A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.

Parameters

<i>test_liion_ptr</i>	A Storage pointer to a test Lilon object.
-----------------------	---

```

271 {
272     double dt_hrs = 1;
273     double load_kW = 100;
274
275     testFloatEquals(
276         test_liion_ptr->getAvailablekW(dt_hrs),
277         100, // hits power capacity constraint
278         __FILE__,
279         __LINE__
280     );
281
282     testFloatEquals(
283         test_liion_ptr->getAcceptablekW(dt_hrs),
284         100, // hits power capacity constraint
285         __FILE__,
286         __LINE__
287     );
288
289     test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing out

```

```

290
291     testFloatEquals(
292         test_liion_ptr->getAvailablekW(dt_hrs),
293         0, // is already hitting power capacity constraint
294         __FILE__,
295         __LINE__
296     );
297
298     testFloatEquals(
299         test_liion_ptr->getAcceptablekW(dt_hrs),
300         0, // is already hitting power capacity constraint
301         __FILE__,
302         __LINE__
303     );
304
305     load_kW = test_liion_ptr->commitDischarge(0, dt_hrs, 100, load_kW);
306
307     testFloatEquals(
308         load_kW,
309         0,
310         __FILE__,
311         __LINE__
312     );
313
314     testFloatEquals(
315         test_liion_ptr->power_kW,
316         0,
317         __FILE__,
318         __LINE__
319     );
320
321     return;
322 } /* testCommitDischarge_LiIon() */

```

5.66.2.5 testConstruct_LiIon()

```

Storage * testConstruct_LiIon (
    void )

```

A function to construct a [LiIon](#) object and spot check some post-construction attributes.

Returns

A [Storage](#) pointer to a test [LiIon](#) object.

```

63 {
64     LiIonInputs liion_inputs;
65
66     Storage* test_liion_ptr = new LiIon(8760, 1, liion_inputs);
67
68     testTruth(
69         test_liion_ptr->type_str == "LIION",
70         __FILE__,
71         __LINE__
72     );
73
74     testFloatEquals(
75         ((LiIon*)test_liion_ptr)->init_SOC,
76         0.5,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         ((LiIon*)test_liion_ptr)->min_SOC,
83         0.15,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         ((LiIon*)test_liion_ptr)->hysteresis_SOC,
90         0.5,
91         __FILE__,
92         __LINE__
93     );

```

```

94
95     testFloatEquals(
96         ((LiIon*)test_liion_ptr)->max_SOC,
97         0.9,
98         __FILE__,
99         __LINE__
100    );
101
102     testFloatEquals(
103         ((LiIon*)test_liion_ptr)->charging_efficiency,
104         0.9,
105         __FILE__,
106         __LINE__
107    );
108
109     testFloatEquals(
110         ((LiIon*)test_liion_ptr)->discharging_efficiency,
111         0.9,
112         __FILE__,
113         __LINE__
114    );
115
116     testFloatEquals(
117         ((LiIon*)test_liion_ptr)->replace_SOH,
118         0.8,
119         __FILE__,
120         __LINE__
121    );
122
123     testFloatEquals(
124         ((LiIon*)test_liion_ptr)->power_kW,
125         0,
126         __FILE__,
127         __LINE__
128    );
129
130     testFloatEquals(
131         ((LiIon*)test_liion_ptr)->SOH_vec.size(),
132         8760,
133         __FILE__,
134         __LINE__
135    );
136
137     testTruth(
138         not ((LiIon*)test_liion_ptr)->power_degradation_flag,
139         __FILE__,
140         __LINE__
141    );
142
143     testFloatEquals(
144         test_liion_ptr->energy_capacity_kWh,
145         ((LiIon*)test_liion_ptr)->dynamic_energy_capacity_kWh,
146         __FILE__,
147         __LINE__
148    );
149
150     testFloatEquals(
151         test_liion_ptr->power_capacity_kW,
152         ((LiIon*)test_liion_ptr)->dynamic_power_capacity_kW,
153         __FILE__,
154         __LINE__
155    );
156
157     return test_liion_ptr;
158 } /* testConstruct_LiIon() */

```

5.67 test/source/Storage/test_Storage.cpp File Reference

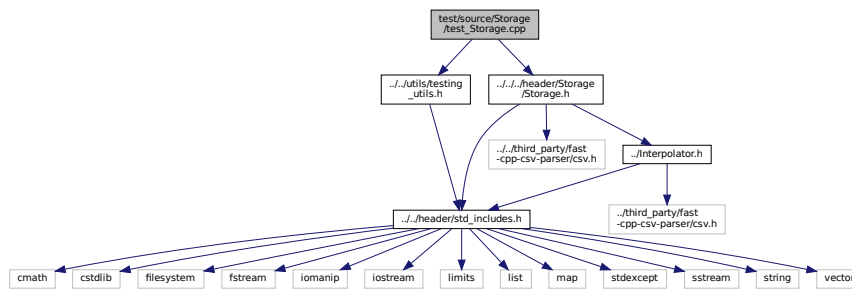
Testing suite for [Storage](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Storage/Storage.h"

```

Include dependency graph for test_Storage.cpp:



Functions

- [Storage](#) * [testConstruct_Storage](#) (void)
A function to construct a [Storage](#) object and spot check some post-construction attributes.
- void [testBadConstruct_Storage](#) (void)
Function to test the trying to construct a [Storage](#) object given bad inputs is being handled as expected.
- int [main](#) (int argc, char **argv)

5.67.1 Detailed Description

Testing suite for [Storage](#) class.

A suite of tests for the [Storage](#) class.

5.67.2 Function Documentation

5.67.2.1 main()

```

int main (
    int argc,
    char ** argv )
161 {
162     #ifdef _WIN32
163         activateVirtualTerminal();
164     #endif /* _WIN32 */
165
166     printGold("\tTesting Storage");
167
168     srand(time(NULL));
169
170     Storage* test_storage_ptr = testConstruct_Storage();
171
172     try {
173         testBadConstruct_Storage();
174     }
175
176     catch (...) {
177
178
179

```



```

180         delete test_storage_ptr;
181
182         printGold(" ..... ");
183         printRed("FAIL");
184         std::cout << std::endl;
185         throw;
186     }
187
188
189     delete test_storage_ptr;
190
191     printGold(" ..... ");
192     printGreen("PASS");
193     std::cout << std::endl;
194     return 0;
195
196 } /* main() */

```

5.67.2.2 testBadConstruct_Storage()

```

void testBadConstruct_Storage (
    void )

```

Function to test the trying to construct a [Storage](#) object given bad inputs is being handled as expected.

```

134 {
135     bool error_flag = true;
136
137     try {
138         StorageInputs bad_storage_inputs;
139         bad_storage_inputs.energy_capacity_kWh = 0;
140
141         Storage bad_storage(8760, 1, bad_storage_inputs);
142
143         error_flag = false;
144     } catch (...) {
145         // Task failed successfully! =P
146     }
147     if (not error_flag) {
148         expectedErrorNotDetected(__FILE__, __LINE__);
149     }
150
151     return;
152 } /* testBadConstruct_Storage() */

```

5.67.2.3 testConstruct_Storage()

```

Storage * testConstruct_Storage (
    void )

```

A function to construct a [Storage](#) object and spot check some post-construction attributes.

Returns

A [Renewable](#) pointer to a test [Storage](#) object.

```

63 {
64     StorageInputs storage_inputs;
65
66     Storage* test_storage_ptr = new Storage(8760, 1, storage_inputs);
67
68     testFloatEquals(
69         test_storage_ptr->power_capacity_kW,
70         100,
71         __FILE__,
72         __LINE__
73     );
74

```

```

75     testFloatEquals(
76         test_storage_ptr->energy_capacity_kWh,
77         1000,
78         __FILE__,
79         __LINE__
80     );
81
82     testFloatEquals(
83         test_storage_ptr->charge_vec_kWh.size(),
84         8760,
85         __FILE__,
86         __LINE__
87     );
88
89     testFloatEquals(
90         test_storage_ptr->charging_power_vec_kW.size(),
91         8760,
92         __FILE__,
93         __LINE__
94     );
95
96     testFloatEquals(
97         test_storage_ptr->discharging_power_vec_kW.size(),
98         8760,
99         __FILE__,
100        __LINE__
101    );
102
103    testFloatEquals(
104        test_storage_ptr->capital_cost_vec.size(),
105        8760,
106        __FILE__,
107        __LINE__
108    );
109
110    testFloatEquals(
111        test_storage_ptr->operation_maintenance_cost_vec.size(),
112        8760,
113        __FILE__,
114        __LINE__
115    );
116
117    return test_storage_ptr;
118 } /* testConstruct_Storage() */

```

5.68 test/source/test_Controller.cpp File Reference

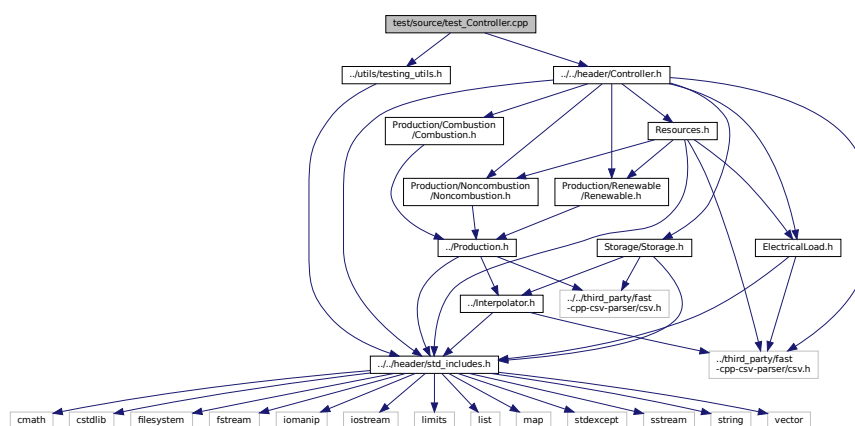
Testing suite for [Controller](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Controller.h"

```

Include dependency graph for test_Controller.cpp:



Functions

- `Controller * testConstruct_Controller` (void)
A function to construct a `Controller` object.
- `int main` (int argc, char **argv)

5.68.1 Detailed Description

Testing suite for `Controller` class.

A suite of tests for the `Controller` class.

5.68.2 Function Documentation

5.68.2.1 `main()`

```
int main (
    int argc,
    char ** argv )
75 {
76     #ifdef _WIN32
77         activateVirtualTerminal();
78     #endif /* _WIN32 */
79     printGold("\tTesting Controller");
80
81     srand(time(NULL));
82
83
84     Controller* test_controller_ptr = testConstruct_Controller();
85
86
87     try {
88         //...
89     }
90
91
92     catch (...) {
93         delete test_controller_ptr;
94
95         printGold(" ..... ");
96         printRed("FAIL");
97         std::cout << std::endl;
98         throw;
99     }
100
101
102     delete test_controller_ptr;
103
104     printGold(" ..... ");
105     printGreen("PASS");
106     std::cout << std::endl;
107     return 0;
108 } /* main() */
```

5.68.2.2 testConstruct_Controller()

```
Controller * testConstruct_Controller (
    void )
```

A function to construct a [Controller](#) object.

Returns

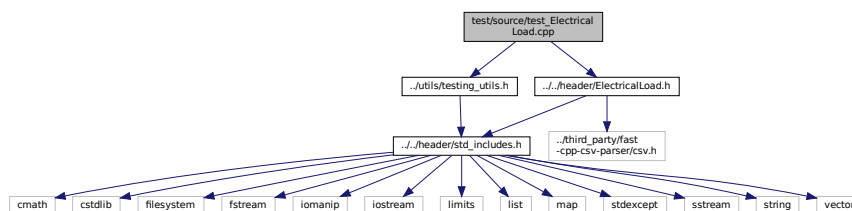
A pointer to a test [Controller](#) object.

```
62 {
63     Controller* test_controller_ptr = new Controller();
64
65     return test_controller_ptr;
66 } /* testConstruct_Controller() */
```

5.69 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

- [ElectricalLoad](#) * [testConstruct_ElectricalLoad](#) (void)
A function to construct an [ElectricalLoad](#) object.
- void [testPostConstructionAttributes_ElectricalLoad](#) ([ElectricalLoad](#) *test_electrical_load_ptr)
A function to check the values of various post-construction attributes.
- void [testDataRead_ElectricalLoad](#) ([ElectricalLoad](#) *test_electrical_load_ptr)
A function to check the values read into the test [ElectricalLoad](#) object.
- int [main](#) (int argc, char **argv)

5.69.1 Detailed Description

Testing suite for [ElectricalLoad](#) class.

A suite of tests for the [ElectricalLoad](#) class.

5.69.2 Function Documentation

5.69.2.1 main()

```

int main (
    int argc,
    char ** argv )
248 {
249     #ifdef _WIN32
250         activateVirtualTerminal();
251     #endif /* _WIN32 */
252
253     printGold("\tTesting ElectricalLoad");
254
255     srand(time(NULL));
256
257     ElectricalLoad* test_electrical_load_ptr = testConstruct_ElectricalLoad();
258
259     try {
260         testPostConstructionAttributes_ElectricalLoad(test_electrical_load_ptr);
261         testDataRead_ElectricalLoad(test_electrical_load_ptr);
262     }
263
264     catch (...) {
265         delete test_electrical_load_ptr;
266
267         printGold(" ..... ");
268         printRed("FAIL");
269         std::cout << std::endl;
270         throw;
271     }
272
273     delete test_electrical_load_ptr;
274
275     printGold(" ..... ");
276     printGreen("PASS");
277     std::cout << std::endl;
278     return 0;
279 } /* main() */

```

5.69.2.2 testConstruct_ElectricalLoad()

```

ElectricalLoad * testConstruct_ElectricalLoad (
    void )

```

A function to construct an [ElectricalLoad](#) object.

Returns

A pointer to a test [ElectricalLoad](#) object.

```

62 {
63     std::string path_2_electrical_load_time_series =
64         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
65
66     ElectricalLoad* test_electrical_load_ptr =
67         new ElectricalLoad(path_2_electrical_load_time_series);
68
69     testTruth(
70         test_electrical_load_ptr->path_2_electrical_load_time_series ==
71         path_2_electrical_load_time_series,
72         __FILE__,
73         __LINE__
74     );
75
76     return test_electrical_load_ptr;
77 } /* testConstruct_ElectricalLoad() */

```

5.69.2.3 testDataRead_ElectricalLoad()

```
void testDataRead_ElectricalLoad (
    ElectricalLoad * test_electrical_load_ptr )
```

A function to check the values read into the test [ElectricalLoad](#) object.

Parameters

<code>test_electrical_load_ptr</code>	A pointer to the test ElectricalLoad object.
---------------------------------------	--

```
153 {
154     std::vector<double> expected_dt_vec_hrs (48, 1);
155
156     std::vector<double> expected_time_vec_hrs = {
157         0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
158         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
159         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
160         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
161     };
162
163     std::vector<double> expected_load_vec_kW = {
164         360.253836463674,
165         355.171277826775,
166         353.776453532298,
167         353.75405737934,
168         346.592867404975,
169         340.132411175118,
170         337.354867340578,
171         340.644115618736,
172         363.639028500678,
173         378.787797779238,
174         372.215798201712,
175         395.093925731298,
176         402.325427142659,
177         386.907725462306,
178         380.709170928091,
179         372.062070914977,
180         372.328646856954,
181         391.841444284136,
182         394.029351759596,
183         383.369407765254,
184         381.093099675206,
185         382.604158946193,
186         390.744843709034,
187         383.13949492437,
188         368.150393976985,
189         364.629744480226,
190         363.572736804082,
191         359.854924202248,
192         355.207590170267,
193         349.094656012401,
194         354.365935871597,
195         343.380608328546,
196         404.673065729266,
197         486.296896820126,
198         480.225974100847,
199         457.318764401085,
200         418.177339948609,
201         414.399018364126,
202         409.678420185754,
203         404.768766016563,
204         401.699589920585,
205         402.44339040654,
206         398.138372541906,
207         396.010498627646,
208         390.165117432277,
209         375.850429417013,
210         365.567100746484,
211         365.429624610923
212     };
213
214     for (int i = 0; i < 48; i++) {
215         testFloatEquals(
216             test_electrical_load_ptr->dt_vec_hrs[i],
217             expected_dt_vec_hrs[i],
218             __FILE__,
219             __LINE__
220         );
221
222         testFloatEquals(
```

```

223         test_electrical_load_ptr->time_vec_hrs[i],
224         expected_time_vec_hrs[i],
225         __FILE__,
226         __LINE__
227     );
228
229     testFloatEquals(
230         test_electrical_load_ptr->load_vec_kW[i],
231         expected_load_vec_kW[i],
232         __FILE__,
233         __LINE__
234     );
235
236 }
237
238 return;
239 } /* testDataRead_ElectricalLoad() */

```

5.69.2.4 testPostConstructionAttributes_ElectricalLoad()

```

void testPostConstructionAttributes_ElectricalLoad (
    ElectricalLoad * test_electrical_load_ptr )

```

A function to check the values of various post-construction attributes.

Parameters

<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
---------------------------------	--

```

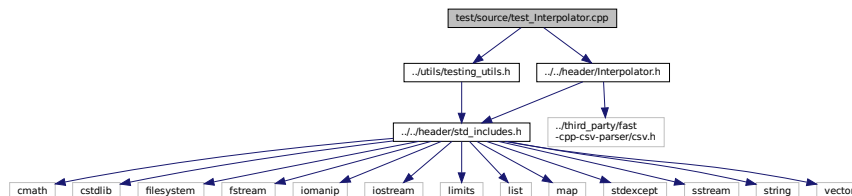
98 {
99     testFloatEquals(
100         test_electrical_load_ptr->n_points,
101         8760,
102         __FILE__,
103         __LINE__
104     );
105
106     testFloatEquals(
107         test_electrical_load_ptr->n_years,
108         0.999886,
109         __FILE__,
110         __LINE__
111     );
112
113     testFloatEquals(
114         test_electrical_load_ptr->min_load_kW,
115         82.1211213927802,
116         __FILE__,
117         __LINE__
118     );
119
120     testFloatEquals(
121         test_electrical_load_ptr->mean_load_kW,
122         258.373472633202,
123         __FILE__,
124         __LINE__
125     );
126
127
128     testFloatEquals(
129         test_electrical_load_ptr->max_load_kW,
130         500,
131         __FILE__,
132         __LINE__
133     );
134
135     return;
136 } /* testPostConstructionAttributes_ElectricalLoad() */

```

5.70 test/source/test_Interpolator.cpp File Reference

Testing suite for [Interpolator](#) class.

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



Functions

- [Interpolator](#) * [testConstruct_Interpolator](#) (void)
A function to construct an [Interpolator](#) object.
- void [testDataRead1D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_1D, std::string path_2↵_data_1D)
A function to check the 1D data values read into the [Interpolator](#) object.
- void [testBadIndexing1D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_bad)
A function to check if bad key errors are being handled properly.
- void [testInvalidInterpolation1D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_1D)
Function to check if attempting to interpolate outside the given 1D data domain is handled properly.
- void [testInterpolation1D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_1D)
Function to check that the [Interpolator](#) object is returning the expected 1D interpolation values.
- void [testDataRead2D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_2D, std::string path_2↵_data_2D)
A function to check the 2D data values read into the [Interpolator](#) object.
- void [testInvalidInterpolation2D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_2D)
Function to check if attempting to interpolate outside the given 2D data domain is handled properly.
- void [testInterpolation2D_Interpolator](#) ([Interpolator](#) *test_interpolator_ptr, int data_key_2D)
Function to check that the [Interpolator](#) object is returning the expected 2D interpolation values.
- int [main](#) (int argc, char **argv)

5.70.1 Detailed Description

Testing suite for [Interpolator](#) class.

A suite of tests for the [Interpolator](#) class.

5.70.2 Function Documentation

5.70.2.1 main()

```

int main (
    int argc,
    char ** argv )
725 {
726     #ifdef _WIN32
727         activateVirtualTerminal();
728     #endif /* _WIN32 */
729
730     printGold("\n\tTesting Interpolator");
731
732     srand(time(NULL));
733
734
735     Interpolator* test_interpolator_ptr = testConstruct_Interpolator();
736
737
738     try {
739         int data_key_1D = 1;
740         std::string path_2_data_1D =
741             "data/test/interpolation/diesel_fuel_curve.csv";
742
743         testDataRead1D_Interpolator(test_interpolator_ptr, data_key_1D, path_2_data_1D);
744         testBadIndexing1D_Interpolator(test_interpolator_ptr, -99);
745         testInvalidInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
746         testInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
747
748
749         int data_key_2D = 2;
750         std::string path_2_data_2D =
751             "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
752
753         testDataRead2D_Interpolator(test_interpolator_ptr, data_key_2D, path_2_data_2D);
754         testInvalidInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
755         testInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
756     }
757
758
759     catch (...) {
760         delete test_interpolator_ptr;
761
762         printGold(" ..... ");
763         printRed("FAIL");
764         std::cout << std::endl;
765         throw;
766     }
767
768
769     delete test_interpolator_ptr;
770
771     printGold(" ..... ");
772     printGreen("PASS");
773     std::cout << std::endl;
774     return 0;
775 } /* main() */

```

5.70.2.2 testBadIndexing1D_Interpolator()

```

void testBadIndexing1D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key_bad )

```

A function to check if bad key errors are being handled properly.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_bad</i>	A key used to index into the Interpolator object.

```

212 {

```

```

213     bool error_flag = true;
214
215     try {
216         test_interpolator_ptr->interp1D(data_key_bad, 0);
217         error_flag = false;
218     } catch (...) {
219         // Task failed successfully! =P
220     }
221     if (not error_flag) {
222         expectedErrorNotDetected(__FILE__, __LINE__);
223     }
224
225     return;
226 } /* testBadIndexing1D_Interpolator() */

```

5.70.2.3 testConstruct_Interpolator()

```

Interpolator * testConstruct_Interpolator (
    void )

```

A function to construct an [Interpolator](#) object.

Returns

A pointer to a test [Interpolator](#) object.

```

62 {
63     Interpolator* test_interpolator_ptr = new Interpolator();
64
65     return test_interpolator_ptr;
66 } /* testConstruct_Interpolator() */

```

5.70.2.4 testDataRead1D_Interpolator()

```

void testDataRead1D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key,
    std::string path_2_data_1D )

```

A function to check the 1D data values read into the [Interpolator](#) object.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_1D</i>	A key used to index into the Interpolator object.
<i>path_2_data_1D</i>	A path (either relative or absolute) to the interpolation data.

```

95 {
96     test_interpolator_ptr->addData1D(data_key_1D, path_2_data_1D);
97
98     testTruth(
99         test_interpolator_ptr->path_map_1D[data_key_1D] == path_2_data_1D,
100         __FILE__,
101         __LINE__
102     );
103
104     testFloatEquals(
105         test_interpolator_ptr->interp_map_1D[data_key_1D].n_points,
106         16,
107         __FILE__,

```

```

108     __LINE__
109 );
110
111 testFloatEquals (
112     test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec.size(),
113     16,
114     __FILE__,
115     __LINE__
116 );
117
118 std::vector<double> expected_x_vec = {
119     0,
120     0.3,
121     0.35,
122     0.4,
123     0.45,
124     0.5,
125     0.55,
126     0.6,
127     0.65,
128     0.7,
129     0.75,
130     0.8,
131     0.85,
132     0.9,
133     0.95,
134     1
135 };
136
137 std::vector<double> expected_y_vec = {
138     4.68079520372916,
139     11.1278522361839,
140     12.4787834830748,
141     13.7808847600209,
142     15.0417468303382,
143     16.277263,
144     17.4612831516442,
145     18.6279054806525,
146     19.7698039220515,
147     20.8893499214868,
148     21.955378,
149     23.0690535155297,
150     24.1323614374927,
151     25.1797231192866,
152     26.2122451458747,
153     27.254952
154 };
155
156 for (int i = 0; i < test_interpolator_ptr->interp_map_1D[data_key_1D].n_points; i++) {
157     testFloatEquals (
158         test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec[i],
159         expected_x_vec[i],
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals (
165         test_interpolator_ptr->interp_map_1D[data_key_1D].y_vec[i],
166         expected_y_vec[i],
167         __FILE__,
168         __LINE__
169     );
170 }
171
172 testFloatEquals (
173     test_interpolator_ptr->interp_map_1D[data_key_1D].min_x,
174     expected_x_vec[0],
175     __FILE__,
176     __LINE__
177 );
178
179 testFloatEquals (
180     test_interpolator_ptr->interp_map_1D[data_key_1D].max_x,
181     expected_x_vec[expected_x_vec.size() - 1],
182     __FILE__,
183     __LINE__
184 );
185
186 return;
187 } /* testDataRead1D_Interpolator() */

```

5.70.2.5 testDataRead2D_Interpolator()

```
void testDataRead2D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key,
    std::string path_2_data_2D )
```

A function to check the 2D data values read into the [Interpolator](#) object.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_2D</i>	A key used to index into the Interpolator object.
<i>path_2_data_2D</i>	A path (either relative or absolute) to the interpolation data.

```
402 {
403     test_interpolator_ptr->addData2D(data_key_2D, path_2_data_2D);
404
405     testTruth(
406         test_interpolator_ptr->path_map_2D[data_key_2D] == path_2_data_2D,
407         __FILE__,
408         __LINE__
409     );
410
411     testFloatEquals(
412         test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows,
413         16,
414         __FILE__,
415         __LINE__
416     );
417
418     testFloatEquals(
419         test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols,
420         16,
421         __FILE__,
422         __LINE__
423     );
424
425     testFloatEquals(
426         test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec.size(),
427         16,
428         __FILE__,
429         __LINE__
430     );
431
432     testFloatEquals(
433         test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec.size(),
434         16,
435         __FILE__,
436         __LINE__
437     );
438
439     testFloatEquals(
440         test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix.size(),
441         16,
442         __FILE__,
443         __LINE__
444     );
445
446     testFloatEquals(
447         test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[0].size(),
448         16,
449         __FILE__,
450         __LINE__
451     );
452
453     std::vector<double> expected_x_vec = {
454         0.25, 0.75, 1.25, 1.75, 2.25, 2.75, 3.25, 3.75, 4.25, 4.75, 5.25, 5.75, 6.25, 6.75, 7.25, 7.75
455     };
456
457     std::vector<double> expected_y_vec = {
458         5,
459         6,
460         7,
461         8,
462         9,
463         10,
464         11,
```

```

465         12,
466         13,
467         14,
468         15,
469         16,
470         17,
471         18,
472         19,
473         20
474     };
475
476     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; i++) {
477         testFloatEquals(
478             test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec[i],
479             expected_x_vec[i],
480             __FILE__,
481             __LINE__
482         );
483     }
484
485     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
486         testFloatEquals(
487             test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec[i],
488             expected_y_vec[i],
489             __FILE__,
490             __LINE__
491         );
492     }
493
494     testFloatEquals(
495         test_interpolator_ptr->interp_map_2D[data_key_2D].min_x,
496         expected_x_vec[0],
497         __FILE__,
498         __LINE__
499     );
500
501     testFloatEquals(
502         test_interpolator_ptr->interp_map_2D[data_key_2D].max_x,
503         expected_x_vec[expected_x_vec.size() - 1],
504         __FILE__,
505         __LINE__
506     );
507
508     testFloatEquals(
509         test_interpolator_ptr->interp_map_2D[data_key_2D].min_y,
510         expected_y_vec[0],
511         __FILE__,
512         __LINE__
513     );
514
515     testFloatEquals(
516         test_interpolator_ptr->interp_map_2D[data_key_2D].max_y,
517         expected_y_vec[expected_y_vec.size() - 1],
518         __FILE__,
519         __LINE__
520     );
521
522     std::vector<std::vector<double>> expected_z_matrix = {
523         {0, 0.129128125, 0.268078125, 0.404253125, 0.537653125, 0.668278125, 0.796128125, 0.921203125,
524         1, 1, 0, 0, 0, 0, 0},
525         {0, 0.11160375, 0.24944375, 0.38395375, 0.51513375, 0.64298375, 0.76750375, 0.88869375, 1, 1, 1,
526         1, 1, 1, 1},
527         {0, 0.094079375, 0.230809375, 0.363654375, 0.492614375, 0.617689375, 0.738879375, 0.856184375,
528         0.969604375, 1, 1, 1, 1, 1, 1, 1},
529         {0, 0.076555, 0.212175, 0.343355, 0.470095, 0.592395, 0.710255, 0.823675, 0.932655, 1, 1, 1, 1,
530         1, 1, 1},
531         {0, 0.059030625, 0.193540625, 0.323055625, 0.447575625, 0.567100625, 0.681630625, 0.791165625,
532         0.895705625, 0.995250625, 1, 1, 1, 1, 1, 1},
533         {0, 0.04150625, 0.17490625, 0.30275625, 0.42505625, 0.54180625, 0.65300625, 0.75865625,
534         0.85875625, 0.95330625, 1, 1, 1, 1, 1, 1},
535         {0, 0.023981875, 0.156271875, 0.282456875, 0.402536875, 0.516511875, 0.624381875, 0.726146875,
536         0.821806875, 0.911361875, 0.994811875, 1, 1, 1, 1, 1},
537         {0, 0.0064575, 0.1376375, 0.2621575, 0.3800175, 0.4912175, 0.5957575, 0.6936375, 0.7848575,
538         0.8694175, 0.9473175, 1, 1, 1, 1, 1},
539         {0, 0, 0.119003125, 0.241858125, 0.357498125, 0.465923125, 0.567133125, 0.661128125,
540         0.747908125, 0.827473125, 0.899823125, 0.964958125, 1, 1, 1, 1},
541         {0, 0, 0.10036875, 0.22155875, 0.33497875, 0.44062875, 0.53850875, 0.62861875, 0.71095875,
542         0.78552875, 0.85232875, 0.91135875, 0.96261875, 1, 1, 1},
543         {0, 0, 0.081734375, 0.201259375, 0.312459375, 0.415334375, 0.509884375, 0.596109375,
544         0.674009375, 0.743584375, 0.804834375, 0.857759375, 0.902359375, 0.938634375, 0.966584375,
545         0.986209375},
546         {0, 0, 0.0631, 0.18096, 0.28994, 0.39004, 0.48126, 0.5636, 0.63706, 0.70164, 0.75734, 0.80416,
547         0.8421, 0.87116, 0.89134, 0.90264},
548         {0, 0, 0.044465625, 0.160660625, 0.267420625, 0.364745625, 0.452635625, 0.531090625,
549         0.600110625, 0.659695625, 0.709845625, 0.750560625, 0.781840625, 0.8036856249999999, 0.816095625,
550         0.819070625},
551         {0, 0, 0.02583125, 0.14036125, 0.24490125, 0.33945125, 0.42401125, 0.49858125, 0.56316125,

```

```

0.61775125, 0.66235125, 0.69696125, 0.72158125, 0.73621125, 0.74085125, 0.73550125},
537 {0, 0, 0.007196875, 0.120061875, 0.222381875, 0.314156875, 0.395386875, 0.466071875,
0.526211875, 0.575806875, 0.614856875, 0.643361875, 0.661321875, 0.668736875, 0.665606875,
0.651931875},
538 {0, 0, 0, 0.0997625, 0.1998625, 0.2888625, 0.3667625, 0.4335625, 0.4892625, 0.5338625,
0.5673625, 0.5897625, 0.6010625, 0.6012625, 0.5903625, 0.5683625}
539 };
540
541 for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
542     for (int j = 0; j < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; j++) {
543         testFloatEquals(
544             test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[i][j],
545             expected_z_matrix[i][j],
546             __FILE__,
547             __LINE__
548         );
549     }
550 }
551
552 return;
553 } /* testDataRead2D_Interpolator() */

```

5.70.2.6 testInterpolation1D_Interpolator()

```

void testInterpolation1D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key_1D )

```

Function to check that the [Interpolator](#) object is returning the expected 1D interpolation values.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_1D</i>	A key used to index into the Interpolator object.

```

322 {
323     std::vector<double> interp_x_vec = {
324         0,
325         0.170812859791767,
326         0.322739274162545,
327         0.369750203682042,
328         0.443532869135929,
329         0.471567864244626,
330         0.536513734479662,
331         0.586125806988674,
332         0.601101175455075,
333         0.658356862575221,
334         0.70576929893201,
335         0.784069734739331,
336         0.805765927542453,
337         0.884747873186048,
338         0.930870496062112,
339         0.979415217694769,
340         1
341     };
342
343     std::vector<double> expected_interp_y_vec = {
344         4.68079520372916,
345         8.35159603357656,
346         11.7422361561399,
347         12.9931187917615,
348         14.8786636301325,
349         15.5746957307243,
350         17.1419229487141,
351         18.3041866133728,
352         18.6530540913696,
353         19.9569217633299,
354         21.012354614584,
355         22.7142305879957,
356         23.1916726441968,
357         24.8602332554707,
358         25.8172124624032,
359         26.8256741279932,
360         27.254952

```

```

361     };
362
363     for (size_t i = 0; i < interp_x_vec.size(); i++) {
364         testFloatEquals(
365             test_interpolator_ptr->interp1D(data_key_1D, interp_x_vec[i]),
366             expected_interp_y_vec[i],
367             __FILE__,
368             __LINE__
369         );
370     }
371
372     return;
373 } /* testInterpolation1D_Interpolator() */

```

5.70.2.7 testInterpolation2D_Interpolator()

```

void testInterpolation2D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key_2D )

```

Function to check that the [Interpolator](#) object is returning the expected 2D interpolation values.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_2D</i>	A key used to index into the Interpolator object.

```

649 {
650     std::vector<double> interp_x_vec = {
651         0.389211848822208,
652         0.836477431896843,
653         1.52738334015579,
654         1.92640601114508,
655         2.27297317532019,
656         2.87416589636605,
657         3.72275770908175,
658         3.95063175885536,
659         4.68097139867404,
660         4.97775020449812,
661         5.55184219980547,
662         6.06566629451658,
663         6.27927876785062,
664         6.96218133671013,
665         7.51754442460228
666     };
667
668     std::vector<double> interp_y_vec = {
669         5.45741899698926,
670         6.00101329139007,
671         7.50567689404182,
672         8.77681262912881,
673         9.45143678206774,
674         10.7767876462885,
675         11.4795760857165,
676         12.9430684577599,
677         13.303544885703,
678         14.5069863517863,
679         15.1487890438045,
680         16.086524049077,
681         17.176609978648,
682         18.4155153740256,
683         19.1704554940162
684     };
685
686     std::vector<std::vector<double>> expected_interp_z_matrix = {
687
688         {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
689
690         {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
691
692         {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
693
694         {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240

```

```

691 {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.770611
692 {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
693 {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
694 {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
695 {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.685
696 {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
697 {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
698 {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
699 {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
700 {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
701 {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
702 };
703
704 for (size_t i = 0; i < interp_y_vec.size(); i++) {
705     for (size_t j = 0; j < interp_x_vec.size(); j++) {
706         testFloatEquals(
707             test_interpolator_ptr->interp2D(data_key_2D, interp_x_vec[j], interp_y_vec[i]),
708             expected_interp_z_matrix[i][j],
709             __FILE__,
710             __LINE__
711         );
712     }
713 }
714
715 return;
716 } /* testInterpolation2D_Interpolator() */

```

5.70.2.8 testInvalidInterpolation1D_Interpolator()

```

void testInvalidInterpolation1D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key_1D )

```

Function to check if attempting to interpolate outside the given 1D data domain is handled properly.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_1D</i>	A key used to index into the Interpolator object.

```

252 {
253     bool error_flag = true;
254
255     try {
256         test_interpolator_ptr->interp1D(data_key_1D, -1);
257         error_flag = false;
258     } catch (...) {
259         // Task failed successfully! =P
260     }
261     if (not error_flag) {
262         expectedErrorNotDetected(__FILE__, __LINE__);
263     }
264
265     try {
266         test_interpolator_ptr->interp1D(data_key_1D, 2);
267         error_flag = false;
268     } catch (...) {
269         // Task failed successfully! =P
270     }
271     if (not error_flag) {
272         expectedErrorNotDetected(__FILE__, __LINE__);
273     }
274 }

```



```

275     try {
276         test_interpolator_ptr->interp1D(data_key_1D, 0 - FLOAT_TOLERANCE);
277         error_flag = false;
278     } catch (...) {
279         // Task failed successfully! =P
280     }
281     if (not error_flag) {
282         expectedErrorNotDetected(__FILE__, __LINE__);
283     }
284
285     try {
286         test_interpolator_ptr->interp1D(data_key_1D, 1 + FLOAT_TOLERANCE);
287         error_flag = false;
288     } catch (...) {
289         // Task failed successfully! =P
290     }
291     if (not error_flag) {
292         expectedErrorNotDetected(__FILE__, __LINE__);
293     }
294
295     return;
296 } /* testInvalidInterpolation1D_Interpolator() */

```

5.70.2.9 testInvalidInterpolation2D_Interpolator()

```

void testInvalidInterpolation2D_Interpolator (
    Interpolator * test_interpolator_ptr,
    int data_key_2D )

```

Function to check if attempting to interpolate outside the given 2D data domain is handled properly.

Parameters

<i>test_interpolator_ptr</i>	A pointer to the test Interpolator object.
<i>data_key_2D</i>	A key used to index into the Interpolator object.

```

579 {
580     bool error_flag = true;
581
582     try {
583         test_interpolator_ptr->interp2D(data_key_2D, -1, 6);
584         error_flag = false;
585     } catch (...) {
586         // Task failed successfully! =P
587     }
588     if (not error_flag) {
589         expectedErrorNotDetected(__FILE__, __LINE__);
590     }
591
592     try {
593         test_interpolator_ptr->interp2D(data_key_2D, 99, 6);
594         error_flag = false;
595     } catch (...) {
596         // Task failed successfully! =P
597     }
598     if (not error_flag) {
599         expectedErrorNotDetected(__FILE__, __LINE__);
600     }
601
602     try {
603         test_interpolator_ptr->interp2D(data_key_2D, 0.75, -1);
604         error_flag = false;
605     } catch (...) {
606         // Task failed successfully! =P
607     }
608     if (not error_flag) {
609         expectedErrorNotDetected(__FILE__, __LINE__);
610     }
611
612     try {
613         test_interpolator_ptr->interp2D(data_key_2D, 0.75, 99);
614         error_flag = false;
615     } catch (...) {
616         // Task failed successfully! =P

```


- Function to test adding a wind resource and then check the values read into the [Resources](#) component of the test [Model](#) object.*
- void [testAddHydroResource_Model](#) ([Model](#) *test_model_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)
Function to test adding a hydro resource and then check the values read into the [Resources](#) component of the test [Model](#) object.
 - void [testAddHydro_Model](#) ([Model](#) *test_model_ptr, int hydro_resource_key)
Function to test adding a hydroelectric asset to the test [Model](#) object, and then spot check some post-add attributes.
 - void [testAddDiesel_Model](#) ([Model](#) *test_model_ptr)
Function to test adding a suite of diesel generators to the test [Model](#) object, and then spot check some post-add attributes.
 - void [testAddSolar_Model](#) ([Model](#) *test_model_ptr, int solar_resource_key)
Function to test adding a solar PV array to the test [Model](#) object and then spot check some post-add attributes.
 - void [testAddSolar_productionOverride_Model](#) ([Model](#) *test_model_ptr, std::string path_2_normalized_↔ production_time_series)
Function to test adding a solar PV array to the test [Model](#) object using the production override feature, and then spot check some post-add attributes.
 - void [testAddTidal_Model](#) ([Model](#) *test_model_ptr, int tidal_resource_key)
Function to test adding a tidal turbine to the test [Model](#) object and then spot check some post-add attributes.
 - void [testAddWave_Model](#) ([Model](#) *test_model_ptr, int wave_resource_key)
Function to test adding a wave energy converter to the test [Model](#) object and then spot check some post-add attributes.
 - void [testAddWind_Model](#) ([Model](#) *test_model_ptr, int wind_resource_key)
Function to test adding a wind turbine to the test [Model](#) object and then spot check some post-add attributes.
 - void [testAddLilon_Model](#) ([Model](#) *test_model_ptr)
Function to test adding a lithium ion battery energy storage system to the test [Model](#) object and then spot check some post-add attributes.
 - void [testLoadBalance_Model](#) ([Model](#) *test_model_ptr)
Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the [Controller](#) as expected.
 - void [testEconomics_Model](#) ([Model](#) *test_model_ptr)
Function to check that the modelled economic metrics are > 0.
 - void [testFuelConsumptionEmissions_Model](#) ([Model](#) *test_model_ptr)
Function to check that the modelled fuel consumption and emissions are > 0.
 - int [main](#) (int argc, char **argv)

5.71.1 Detailed Description

Testing suite for [Model](#) class.

A suite of tests for the [Model](#) class.

5.71.2 Function Documentation

5.71.2.1 main()

```

int main (
    int argc,
    char ** argv )

1490 {
1491     #ifdef _WIN32
1492         activateVirtualTerminal();
1493     #endif /* _WIN32 */
1494
1495     printGold("\tTesting Model");
1496     std::cout << std::flush;
1497
1498     srand(time(NULL));
1499
1500
1501     std::string path_2_electrical_load_time_series =
1502         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
1503
1504     ModelInputs test_model_inputs;
1505     test_model_inputs.path_2_electrical_load_time_series =
1506         path_2_electrical_load_time_series;
1507
1508     Model* test_model_ptr = testConstruct_Model(test_model_inputs);
1509
1510
1511     try {
1512         testBadConstruct_Model();
1513         testPostConstructionAttributes_Model(test_model_ptr);
1514         testElectricalLoadData_Model(test_model_ptr);
1515
1516
1517         int solar_resource_key = 0;
1518         std::string path_2_solar_resource_data =
1519             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
1520
1521         testAddSolarResource_Model(
1522             test_model_ptr,
1523             path_2_solar_resource_data,
1524             solar_resource_key
1525         );
1526
1527
1528         int tidal_resource_key = 1;
1529         std::string path_2_tidal_resource_data =
1530             "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
1531
1532         testAddTidalResource_Model(
1533             test_model_ptr,
1534             path_2_tidal_resource_data,
1535             tidal_resource_key
1536         );
1537
1538
1539         int wave_resource_key = 2;
1540         std::string path_2_wave_resource_data =
1541             "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
1542
1543         testAddWaveResource_Model(
1544             test_model_ptr,
1545             path_2_wave_resource_data,
1546             wave_resource_key
1547         );
1548
1549
1550         int wind_resource_key = 3;
1551         std::string path_2_wind_resource_data =
1552             "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
1553
1554         testAddWindResource_Model(
1555             test_model_ptr,
1556             path_2_wind_resource_data,
1557             wind_resource_key
1558         );
1559
1560
1561         int hydro_resource_key = 4;
1562         std::string path_2_hydro_resource_data =
1563             "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
1564
1565         testAddHydroResource_Model(
1566             test_model_ptr,
1567             path_2_hydro_resource_data,
1568             hydro_resource_key
1569         );

```

```

1570
1571
1572     std::string path_2_normalized_production_time_series =
1573         "data/test/normalized_production/normalized_solar_production.csv";
1574
1575     // looping solely for the sake of profiling (also tests reset(), which is
1576     // needed for wrapping PGMcpp in an optimizer)
1577     for (int i = 0; i < 1000; i++) {
1578         test_model_ptr->reset();
1579
1580
1581         testAddHydro_Model(test_model_ptr, hydro_resource_key);
1582         testAddDiesel_Model(test_model_ptr);
1583         testAddSolar_Model(test_model_ptr, solar_resource_key);
1584
1585         testAddSolar_productionOverride_Model(
1586             test_model_ptr,
1587             path_2_normalized_production_time_series
1588         );
1589
1590         testAddTidal_Model(test_model_ptr, tidal_resource_key);
1591         testAddWave_Model(test_model_ptr, wave_resource_key);
1592         testAddWind_Model(test_model_ptr, wind_resource_key);
1593
1594
1595         test_model_ptr->run();
1596     }
1597
1598     testLoadBalance_Model(test_model_ptr);
1599     testEconomics_Model(test_model_ptr);
1600     testFuelConsumptionEmissions_Model(test_model_ptr);
1601
1602     test_model_ptr->writeResults("test/test_results/");
1603 }
1604
1605 catch (...) {
1606     delete test_model_ptr;
1607
1608     printGold(" ..... ");
1609     printRed("FAIL");
1610     std::cout << std::endl;
1611     throw;
1612 }
1613
1614 delete test_model_ptr;
1615
1616 printGold(" ..... ");
1617 printGreen("PASS");
1618 std::cout << std::endl;
1619 return 0;
1620 }
1621 /* main() */

```

5.71.2.2 testAddDiesel_Model()

```

void testAddDiesel_Model (
    Model * test_model_ptr )

```

Function to test adding a suite of diesel generators to the test [Model](#) object, and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

918 {
919     DieselInputs diesel_inputs;
920     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
921     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
922
923     test_model_ptr->addDiesel(diesel_inputs);
924
925     testFloatEquals (

```

```

926         test_model_ptr->combustion_ptr_vec.size(),
927         1,
928         __FILE__,
929         __LINE__
930     );
931
932     testFloatEquals (
933         test_model_ptr->combustion_ptr_vec[0]->type,
934         CombustionType :: DIESEL,
935         __FILE__,
936         __LINE__
937     );
938
939     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
940
941     test_model_ptr->addDiesel (diesel_inputs);
942
943     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
944
945     test_model_ptr->addDiesel (diesel_inputs);
946
947     testFloatEquals (
948         test_model_ptr->combustion_ptr_vec.size(),
949         3,
950         __FILE__,
951         __LINE__
952     );
953
954     std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
955
956     for (int i = 0; i < 3; i++) {
957         testFloatEquals (
958             test_model_ptr->combustion_ptr_vec[i]->capacity_kW,
959             expected_diesel_capacity_vec_kW[i],
960             __FILE__,
961             __LINE__
962         );
963     }
964
965     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
966
967     for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
968         test_model_ptr->addDiesel (diesel_inputs);
969     }
970
971     return;
972 } /* testAddDiesel_Model() */

```

5.71.2.3 testAddHydro_Model()

```

void testAddHydro_Model (
    Model * test_model_ptr,
    int hydro_resource_key )

```

Function to test adding a hydroelectric asset to the test [Model](#) object, and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>hydro_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

868 {
869     HydroInputs hydro_inputs;
870     hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
871     hydro_inputs.reservoir_capacity_m3 = 100000;
872     hydro_inputs.init_reservoir_state = 0.5;
873     hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
874     hydro_inputs.resource_key = hydro_resource_key;
875
876     test_model_ptr->addHydro (hydro_inputs);
877
878     testFloatEquals (
879         test_model_ptr->noncombustion_ptr_vec.size(),

```

```

880         1,
881         __FILE__,
882         __LINE__
883     );
884
885     testFloatEquals (
886         test_model_ptr->noncombustion_ptr_vec[0]->type,
887         NoncombustionType :: HYDRO,
888         __FILE__,
889         __LINE__
890     );
891
892     testFloatEquals (
893         test_model_ptr->noncombustion_ptr_vec[0]->resource_key,
894         hydro_resource_key,
895         __FILE__,
896         __LINE__
897     );
898
899     return;
900 } /* testAddHydro_Model() */

```

5.71.2.4 testAddHydroResource_Model()

```

void testAddHydroResource_Model (
    Model * test_model_ptr,
    std::string path_2_hydro_resource_data,
    int hydro_resource_key )

```

Function to test adding a hydro resource and then check the values read into the [Resources](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_hydro_resource_data</i>	A path (either relative or absolute) to the hydro resource data.
<i>hydro_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

773 {
774     test_model_ptr->addResource (
775         NoncombustionType :: HYDRO,
776         path_2_hydro_resource_data,
777         hydro_resource_key
778     );
779
780     std::vector<double> expected_hydro_resource_vec_ms = {
781         2167.91531556942,
782         2046.58261560569,
783         2007.85941123153,
784         2000.11477247929,
785         1917.50527264453,
786         1963.97311577093,
787         1908.46985899809,
788         1886.5267112678,
789         1965.26388854254,
790         1953.64692935289,
791         2084.01504296306,
792         2272.46796101188,
793         2520.29645627096,
794         2715.203242423,
795         2720.36633563203,
796         3130.83228077221,
797         3289.59741021591,
798         3981.45195965772,
799         5295.45929491303,
800         7084.47124360523,
801         7709.20557708454,
802         7436.85238642936,
803         7235.49173429668,
804         6710.14695517339,
805         6015.71085806577,
806         5279.97001316337,

```

```

807         4877.24870889801,
808         4421.60569340303,
809         3919.49483690424,
810         3498.70270322341,
811         3274.10813058883,
812         3147.61233529349,
813         2904.94693324343,
814         2805.55738101,
815         2418.32535637171,
816         2398.96375630723,
817         2260.85100182222,
818         2157.58912702878,
819         2019.47637254377,
820         1913.63295220712,
821         1863.29279076589,
822         1748.41395678279,
823         1695.49224555317,
824         1599.97501375715,
825         1559.96103873397,
826         1505.74855473274,
827         1438.62833664765,
828         1384.41585476901
829     };
830
831     for (size_t i = 0; i < expected_hydro_resource_vec_ms.size(); i++) {
832         testFloatEquals(
833             test_model_ptr->resources.resource_map_1D[hydro_resource_key][i],
834             expected_hydro_resource_vec_ms[i],
835             __FILE__,
836             __LINE__
837         );
838     }
839
840     return;
841 } /* testAddHydroResource_Model() */

```

5.71.2.5 testAddLilon_Model()

```

void testAddLiIon_Model (
    Model * test_model_ptr )

```

Function to test adding a lithium ion battery energy storage system to the test [Model](#) object and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

1244 {
1245     LiIonInputs liion_inputs;
1246
1247     test_model_ptr->addLiIon(liion_inputs);
1248
1249     testFloatEquals(
1250         test_model_ptr->storage_ptr_vec.size(),
1251         1,
1252         __FILE__,
1253         __LINE__
1254     );
1255
1256     testFloatEquals(
1257         test_model_ptr->storage_ptr_vec[0]->type,
1258         StorageType :: LIION,
1259         __FILE__,
1260         __LINE__
1261     );
1262
1263     return;
1264 } /* testAddLiIon_Model() */

```


5.71.2.6 testAddSolar_Model()

```
void testAddSolar_Model (
    Model * test_model_ptr,
    int solar_resource_key )
```

Function to test adding a solar PV array to the test [Model](#) object and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>solar_resource_key</i>	A key used to index into the Resources component of the test Model object.

```
999 {
1000     SolarInputs solar_inputs;
1001     solar_inputs.resource_key = solar_resource_key;
1002
1003     test_model_ptr->addSolar(solar_inputs);
1004
1005     testFloatEquals(
1006         test_model_ptr->renewable_ptr_vec.size(),
1007         1,
1008         __FILE__,
1009         __LINE__
1010     );
1011
1012     testFloatEquals(
1013         test_model_ptr->renewable_ptr_vec[0]->type,
1014         RenewableType :: SOLAR,
1015         __FILE__,
1016         __LINE__
1017     );
1018
1019     return;
1020 } /* testAddSolar_Model() */
```

5.71.2.7 testAddSolar_productionOverride_Model()

```
void testAddSolar_productionOverride_Model (
    Model * test_model_ptr,
    std::string path_2_normalized_production_time_series )
```

Function to test adding a solar PV array to the test [Model](#) object using the production override feature, and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_normalized_production_time_series</i>	A path (either relative or absolute) to the given normalized production time series data.

```
1047 {
1048     SolarInputs solar_inputs;
1049     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
1050         path_2_normalized_production_time_series;
1051
1052     test_model_ptr->addSolar(solar_inputs);
1053
1054     testFloatEquals(
1055         test_model_ptr->renewable_ptr_vec.size(),
1056         2,
1057         __FILE__,
1058         __LINE__
1059     );
1060 }
```

```

1061     testFloatEquals(
1062         test_model_ptr->renewable_ptr_vec[1]->type,
1063         RenewableType :: SOLAR,
1064         __FILE__,
1065         __LINE__
1066     );
1067
1068     testTruth(
1069         test_model_ptr->renewable_ptr_vec[1]->normalized_production_series_given,
1070         __FILE__,
1071         __LINE__
1072     );
1073
1074     testTruth(
1075         test_model_ptr->renewable_ptr_vec[1]->path_2_normalized_production_time_series ==
1076         path_2_normalized_production_time_series,
1077         __FILE__,
1078         __LINE__
1079     );
1080
1081     return;
1082 } /* testAddSolar_productionOverride_Model() */

```

5.71.2.8 testAddSolarResource_Model()

```

void testAddSolarResource_Model (
    Model * test_model_ptr,
    std::string path_2_solar_resource_data,
    int solar_resource_key )

```

Function to test adding a solar resource and then check the values read into the [Resources](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_solar_resource_data</i>	A path (either relative or absolute) to the solar resource data.
<i>solar_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

315 {
316     test_model_ptr->addResource(
317         RenewableType :: SOLAR,
318         path_2_solar_resource_data,
319         solar_resource_key
320     );
321
322     std::vector<double> expected_solar_resource_vec_kWm2 = {
323         0,
324         0,
325         0,
326         0,
327         0,
328         0,
329         8.51702662684015E-05,
330         0.000348341567045,
331         0.00213793728593,
332         0.004099863613322,
333         0.000997135230553,
334         0.009534527624657,
335         0.022927996790616,
336         0.0136071715294,
337         0.002535134127751,
338         0.005206897515821,
339         0.005627658648597,
340         0.000701186722215,
341         0.00017119827089,
342         0,
343         0,
344         0,
345         0,
346         0,
347         0,

```

```

348         0,
349         0,
350         0,
351         0,
352         0,
353         0,
354         0.000141055102242,
355         0.00084525014743,
356         0.024893647822702,
357         0.091245556190749,
358         0.158722176731637,
359         0.152859680515876,
360         0.149922903895116,
361         0.13049996570866,
362         0.03081254222795,
363         0.001218928911125,
364         0.000206092647423,
365         0,
366         0,
367         0,
368         0,
369         0,
370         0
371     };
372
373     for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
374         testFloatEquals(
375             test_model_ptr->resources.resource_map_1D[solar_resource_key][i],
376             expected_solar_resource_vec_kWm2[i],
377             __FILE__,
378             __LINE__
379         );
380     }
381
382     return;
383 } /* testAddSolarResource_Model() */

```

5.71.2.9 testAddTidal_Model()

```

void testAddTidal_Model (
    Model * test_model_ptr,
    int tidal_resource_key )

```

Function to test adding a tidal turbine to the test [Model](#) object and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>tidal_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

1109 {
1110     TidalInputs tidal_inputs;
1111     tidal_inputs.resource_key = tidal_resource_key;
1112
1113     test_model_ptr->addTidal(tidal_inputs);
1114
1115     testFloatEquals(
1116         test_model_ptr->renewable_ptr_vec.size(),
1117         3,
1118         __FILE__,
1119         __LINE__
1120     );
1121
1122     testFloatEquals(
1123         test_model_ptr->renewable_ptr_vec[2]->type,
1124         RenewableType :: TIDAL,
1125         __FILE__,
1126         __LINE__
1127     );
1128
1129     return;
1130 } /* testAddTidal_Model() */

```

5.71.2.10 testAddTidalResource_Model()

```
void testAddTidalResource_Model (
    Model * test_model_ptr,
    std::string path_2_tidal_resource_data,
    int tidal_resource_key )
```

Function to test adding a tidal resource and then check the values read into the [Resources](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_tidal_resource_data</i>	A path (either relative or absolute) to the tidal resource data.
<i>tidal_resource_key</i>	A key used to index into the Resources component of the test Model object.

```
415 {
416     test_model_ptr->addResource(
417         RenewableType :: TIDAL,
418         path_2_tidal_resource_data,
419         tidal_resource_key
420     );
421
422     std::vector<double> expected_tidal_resource_vec_ms = {
423         0.347439913040533,
424         0.770545522195602,
425         0.731352084836198,
426         0.293389814389542,
427         0.209959110813115,
428         0.610609623896497,
429         1.78067162013604,
430         2.53522775118089,
431         2.75966627832024,
432         2.52101111143895,
433         2.05389330201031,
434         1.3461515862445,
435         0.28909254878384,
436         0.897754086048563,
437         1.71406453837407,
438         1.85047408742869,
439         1.71507908595979,
440         1.33540349705416,
441         0.434586143463003,
442         0.500623815700637,
443         1.37172172646733,
444         1.68294125491228,
445         1.56101300975417,
446         1.04925834219412,
447         0.211395463930223,
448         1.03720048903385,
449         1.85059536356448,
450         1.85203242794517,
451         1.4091471616277,
452         0.767776539039899,
453         0.251464906990961,
454         1.47018469375652,
455         2.36260493698197,
456         2.46653750048625,
457         2.12851908739291,
458         1.62783753197988,
459         0.734594890957439,
460         0.441886297300355,
461         1.6574418350918,
462         2.0684558286637,
463         1.87717416992136,
464         1.58871262337931,
465         1.03451227609235,
466         0.193371305159817,
467         0.976400122458815,
468         1.6583227369707,
469         1.76690616570953,
470         1.54801328553115
471     };
```

```

472
473     for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
474         testFloatEquals(
475             test_model_ptr->resources.resource_map_1D[tidal_resource_key][i],
476             expected_tidal_resource_vec_ms[i],
477             __FILE__,
478             __LINE__
479         );
480     }
481
482     return;
483 } /* testAddTidalResource_Model() */

```

5.71.2.11 testAddWave_Model()

```

void testAddWave_Model (
    Model * test_model_ptr,
    int wave_resource_key )

```

Function to test adding a wave energy converter to the test [Model](#) object and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>wave_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

1157 {
1158     WaveInputs wave_inputs;
1159     wave_inputs.resource_key = wave_resource_key;
1160
1161     test_model_ptr->addWave(wave_inputs);
1162
1163     testFloatEquals(
1164         test_model_ptr->renewable_ptr_vec.size(),
1165         4,
1166         __FILE__,
1167         __LINE__
1168     );
1169
1170     testFloatEquals(
1171         test_model_ptr->renewable_ptr_vec[3]->type,
1172         RenewableType :: WAVE,
1173         __FILE__,
1174         __LINE__
1175     );
1176
1177     return;
1178 } /* testAddWave_Model() */

```

5.71.2.12 testAddWaveResource_Model()

```

void testAddWaveResource_Model (
    Model * test_model_ptr,
    std::string path_2_wave_resource_data,
    int wave_resource_key )

```

Function to test adding a wave resource and then check the values read into the [Resources](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_wave_resource_data</i>	A path (either relative or absolute) to the wave resource data.
<i>wave_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

515 {
516     test_model_ptr->addResource(
517         RenewableType :: WAVE,
518         path_2_wave_resource_data,
519         wave_resource_key
520     );
521
522     std::vector<double> expected_significant_wave_height_vec_m = {
523         4.26175222125028,
524         4.25020976167872,
525         4.25656524330349,
526         4.27193854786718,
527         4.28744955711233,
528         4.29421815278154,
529         4.2839937266082,
530         4.25716982457976,
531         4.22419391611483,
532         4.19588925217606,
533         4.17338788587412,
534         4.14672746914214,
535         4.10560041173665,
536         4.05074966447193,
537         3.9953696962433,
538         3.95316976150866,
539         3.92771018142378,
540         3.91129562488595,
541         3.89558312094911,
542         3.87861093931749,
543         3.86538307240754,
544         3.86108961027929,
545         3.86459448853189,
546         3.86796474016882,
547         3.86357412779993,
548         3.85554872014731,
549         3.86044266668675,
550         3.89445961915999,
551         3.95554798115731,
552         4.02265508610476,
553         4.07419587011404,
554         4.10314247143958,
555         4.11738045085928,
556         4.12554995596708,
557         4.12923992001675,
558         4.1229292327442,
559         4.10123955307441,
560         4.06748827895363,
561         4.0336230651344,
562         4.01134236393876,
563         4.00136570034559,
564         3.99368787690411,
565         3.97820924247644,
566         3.95369335178055,
567         3.92742545608532,
568         3.90683362771686,
569         3.89331520944006,
570         3.88256045801583
571     };
572
573     std::vector<double> expected_energy_period_vec_s = {
574         10.4456008226821,
575         10.4614151137651,
576         10.4462827795433,
577         10.4127692097884,
578         10.3734397942723,
579         10.3408599227669,
580         10.32637292093,
581         10.3245412676322,
582         10.310409818185,
583         10.2589529840966,
584         10.1728100603103,
585         10.0862908658929,
586         10.03480243813,
587         10.023673635806,
588         10.0243418565116,
589         10.0063487117653,
590         9.96050302286607,
591         9.9011999635568,
592         9.84451822125472,

```

```

593         9.79726875879626,
594         9.75614594835158,
595         9.7173447961368,
596         9.68342904390577,
597         9.66380508567062,
598         9.6674009575699,
599         9.68927134575103,
600         9.70979984863046,
601         9.70967357906908,
602         9.68983025704562,
603         9.6722855524805,
604         9.67973599910003,
605         9.71977125328293,
606         9.78450442291421,
607         9.86532355233449,
608         9.96158937600019,
609         10.0807018356507,
610         10.2291022504937,
611         10.39458528356,
612         10.5464393581004,
613         10.6553277500484,
614         10.7245553190084,
615         10.7893127285064,
616         10.8846512240849,
617         11.0148158739075,
618         11.1544325654719,
619         11.2772785848343,
620         11.3744362756187,
621         11.4533643503183
622     };
623
624     for (size_t i = 0; i < expected_energy_period_vec_s.size(); i++) {
625         testFloatEquals(
626             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][0],
627             expected_significant_wave_height_vec_m[i],
628             __FILE__,
629             __LINE__
630         );
631
632         testFloatEquals(
633             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][1],
634             expected_energy_period_vec_s[i],
635             __FILE__,
636             __LINE__
637         );
638     }
639
640     return;
641 } /* testAddWaveResource_Model() */

```

5.71.2.13 testAddWind_Model()

```

void testAddWind_Model (
    Model * test_model_ptr,
    int wind_resource_key )

```

Function to test adding a wind turbine to the test [Model](#) object and then spot check some post-add attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>wind_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

1205 {
1206     WindInputs wind_inputs;
1207     wind_inputs.resource_key = wind_resource_key;
1208
1209     test_model_ptr->addWind(wind_inputs);
1210
1211     testFloatEquals(
1212         test_model_ptr->renewable_ptr_vec.size(),
1213         5,
1214         __FILE__,

```

```

1215     __LINE__
1216 );
1217
1218 testFloatEquals(
1219     test_model_ptr->renewable_ptr_vec[4]->type,
1220     RenewableType :: WIND,
1221     __FILE__,
1222     __LINE__
1223 );
1224
1225 return;
1226 } /* testAddWind_Model() */

```

5.71.2.14 testAddWindResource_Model()

```

void testAddWindResource_Model (
    Model * test_model_ptr,
    std::string path_2_wind_resource_data,
    int wind_resource_key )

```

Function to test adding a wind resource and then check the values read into the [Resources](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
<i>path_2_wind_resource_data</i>	A path (either relative or absolute) to the wind resource data.
<i>wind_resource_key</i>	A key used to index into the Resources component of the test Model object.

```

673 {
674     test_model_ptr->addResource(
675         RenewableType :: WIND,
676         path_2_wind_resource_data,
677         wind_resource_key
678     );
679
680     std::vector<double> expected_wind_resource_vec_ms = {
681         6.88566688469997,
682         5.02177105466549,
683         3.74211715899568,
684         5.67169579985362,
685         4.90670669971858,
686         4.29586955031368,
687         7.41155377205065,
688         10.2243290476943,
689         13.1258696725555,
690         13.7016198628274,
691         16.2481482330233,
692         16.5096744355418,
693         13.4354482206162,
694         14.0129230731609,
695         14.5554549260515,
696         13.4454539065912,
697         13.3447169512094,
698         11.7372615098554,
699         12.7200070078013,
700         10.6421127908149,
701         6.09869498990661,
702         5.66355596602321,
703         4.97316966910831,
704         3.48937138360567,
705         2.15917470979169,
706         1.29061103587027,
707         3.43475751425219,
708         4.11706326260927,
709         4.28905275747408,
710         5.75850263196241,
711         8.98293663055264,
712         11.7069822941315,
713         12.4031987075858,
714         15.4096570910089,
715         16.6210843829552,

```



```

716         13.3421219142573,
717         15.2112831900548,
718         18.350864533037,
719         15.8751799822971,
720         15.3921198799796,
721         15.9729192868434,
722         12.4728950178772,
723         10.177050481096,
724         10.7342247355551,
725         8.98846695631389,
726         4.14671169124739,
727         3.17256452697149,
728         3.40036336968628
729     };
730
731     for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
732         testFloatEquals(
733             test_model_ptr->resources.resource_map_1D[wind_resource_key][i],
734             expected_wind_resource_vec_ms[i],
735             __FILE__,
736             __LINE__
737         );
738     }
739
740     return;
741 } /* testAddWindResource_Model() */

```

5.71.2.15 testBadConstruct_Model()

```

void testBadConstruct_Model (
    void )

```

Function to check if passing bad [ModelInputs](#) to the [Model](#) constructor is handled appropriately.

```

91 {
92     bool error_flag = true;
93
94     try {
95         ModelInputs bad_model_inputs; // path_2_electrical_load_time_series left empty
96
97         Model bad_model(bad_model_inputs);
98
99         error_flag = false;
100     } catch (...) {
101         // Task failed successfully! =P
102     }
103     if (not error_flag) {
104         expectedErrorNotDetected(__FILE__, __LINE__);
105     }
106
107     try {
108         ModelInputs bad_model_inputs;
109         bad_model_inputs.path_2_electrical_load_time_series =
110             "data/test/electrical_load/bad_path_";
111         bad_model_inputs.path_2_electrical_load_time_series += std::to_string(rand());
112         bad_model_inputs.path_2_electrical_load_time_series += ".csv";
113
114         Model bad_model(bad_model_inputs);
115
116         error_flag = false;
117     } catch (...) {
118         // Task failed successfully! =P
119     }
120     if (not error_flag) {
121         expectedErrorNotDetected(__FILE__, __LINE__);
122     }
123
124     return;
125 }

```

5.71.2.16 testConstruct_Model()

```

Model* testConstruct_Model (
    ModelInputs test_model_inputs )
64 {
65     Model* test_model_ptr = new Model(test_model_inputs);
66
67     testTruth(
68         test_model_ptr->electrical_load.path_2_electrical_load_time_series ==
69         test_model_inputs.path_2_electrical_load_time_series,
70         __FILE__,
71         __LINE__
72     );
73
74     return test_model_ptr;
75 } /* testConstruct_Model() */

```

5.71.2.17 testEconomics_Model()

```

void testEconomics_Model (
    Model * test_model_ptr )

```

Function to check that the modelled economic metrics are > 0 .

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

1397 {
1398     testGreaterThan(
1399         test_model_ptr->net_present_cost,
1400         0,
1401         __FILE__,
1402         __LINE__
1403     );
1404
1405     testGreaterThan(
1406         test_model_ptr->levellized_cost_of_energy_kWh,
1407         0,
1408         __FILE__,
1409         __LINE__
1410     );
1411
1412     return;
1413 } /* testEconomics_Model() */

```

5.71.2.18 testElectricalLoadData_Model()

```

void testElectricalLoadData_Model (
    Model * test_model_ptr )

```

Function to check the values read into the [ElectricalLoad](#) component of the test [Model](#) object.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

198 {
199     std::vector<double> expected_dt_vec_hrs (48, 1);
200

```

```

201     std::vector<double> expected_time_vec_hrs = {
202         0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
203         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
204         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
205         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
206     };
207
208     std::vector<double> expected_load_vec_kW = {
209         360.253836463674,
210         355.171277826775,
211         353.776453532298,
212         353.75405737934,
213         346.592867404975,
214         340.132411175118,
215         337.354867340578,
216         340.644115618736,
217         363.639028500678,
218         378.787797779238,
219         372.215798201712,
220         395.093925731298,
221         402.325427142659,
222         386.907725462306,
223         380.709170928091,
224         372.062070914977,
225         372.328646856954,
226         391.841444284136,
227         394.029351759596,
228         383.369407765254,
229         381.093099675206,
230         382.604158946193,
231         390.744843709034,
232         383.13949492437,
233         368.150393976985,
234         364.629744480226,
235         363.572736804082,
236         359.854924202248,
237         355.207590170267,
238         349.094656012401,
239         354.365935871597,
240         343.380608328546,
241         404.673065729266,
242         486.296896820126,
243         480.225974100847,
244         457.318764401085,
245         418.177339948609,
246         414.399018364126,
247         409.678420185754,
248         404.768766016563,
249         401.699589920585,
250         402.44339040654,
251         398.138372541906,
252         396.010498627646,
253         390.165117432277,
254         375.850429417013,
255         365.567100746484,
256         365.429624610923
257     };
258
259     for (int i = 0; i < 48; i++) {
260         testFloatEquals(
261             test_model_ptr->electrical_load.dt_vec_hrs[i],
262             expected_dt_vec_hrs[i],
263             __FILE__,
264             __LINE__
265         );
266
267         testFloatEquals(
268             test_model_ptr->electrical_load.time_vec_hrs[i],
269             expected_time_vec_hrs[i],
270             __FILE__,
271             __LINE__
272         );
273
274         testFloatEquals(
275             test_model_ptr->electrical_load.load_vec_kW[i],
276             expected_load_vec_kW[i],
277             __FILE__,
278             __LINE__
279         );
280     }
281
282     return;
283 } /* testElectricalLoadData_Model() */

```

5.71.2.19 testFuelConsumptionEmissions_Model()

```
void testFuelConsumptionEmissions_Model (
    Model * test_model_ptr )
```

Function to check that the modelled fuel consumption and emissions are > 0 .

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```
1430 {
1431     testGreaterThan(
1432         test_model_ptr->total_fuel_consumed_L,
1433         0,
1434         __FILE__,
1435         __LINE__
1436     );
1437
1438     testGreaterThan(
1439         test_model_ptr->total_emissions.CO2_kg,
1440         0,
1441         __FILE__,
1442         __LINE__
1443     );
1444
1445     testGreaterThan(
1446         test_model_ptr->total_emissions.CO_kg,
1447         0,
1448         __FILE__,
1449         __LINE__
1450     );
1451
1452     testGreaterThan(
1453         test_model_ptr->total_emissions.NOx_kg,
1454         0,
1455         __FILE__,
1456         __LINE__
1457     );
1458
1459     testGreaterThan(
1460         test_model_ptr->total_emissions.SOx_kg,
1461         0,
1462         __FILE__,
1463         __LINE__
1464     );
1465
1466     testGreaterThan(
1467         test_model_ptr->total_emissions.CH4_kg,
1468         0,
1469         __FILE__,
1470         __LINE__
1471     );
1472
1473     testGreaterThan(
1474         test_model_ptr->total_emissions.PM_kg,
1475         0,
1476         __FILE__,
1477         __LINE__
1478     );
1479
1480     return;
1481 } /* testFuelConsumptionEmissions_Model() */
```

5.71.2.20 testLoadBalance_Model()

```
void testLoadBalance_Model (
    Model * test_model_ptr )
```

Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the [Controller](#) as expected.

Parameters

<code>test_model_ptr</code>	A pointer to the test Model object.
-----------------------------	---

```

1283 {
1284     double net_load_kW = 0;
1285
1286     Combustion* combustion_ptr;
1287     Noncombustion* noncombustion_ptr;
1288     Renewable* renewable_ptr;
1289     Storage* storage_ptr;
1290
1291     for (int i = 0; i < test_model_ptr->electrical_load.n_points; i++) {
1292         net_load_kW = test_model_ptr->controller.net_load_vec_kW[i];
1293
1294         testLessThanOrEqualTo(
1295             test_model_ptr->controller.net_load_vec_kW[i],
1296             test_model_ptr->electrical_load.max_load_kW,
1297             __FILE__,
1298             __LINE__
1299         );
1300
1301         for (size_t j = 0; j < test_model_ptr->combustion_ptr_vec.size(); j++) {
1302             combustion_ptr = test_model_ptr->combustion_ptr_vec[j];
1303
1304             testFloatEquals(
1305                 combustion_ptr->production_vec_kW[i] -
1306                 combustion_ptr->dispatch_vec_kW[i] -
1307                 combustion_ptr->curtailment_vec_kW[i] -
1308                 combustion_ptr->storage_vec_kW[i],
1309                 0,
1310                 __FILE__,
1311                 __LINE__
1312             );
1313
1314             net_load_kW -= combustion_ptr->production_vec_kW[i];
1315         }
1316
1317         for (size_t j = 0; j < test_model_ptr->noncombustion_ptr_vec.size(); j++) {
1318             noncombustion_ptr = test_model_ptr->noncombustion_ptr_vec[j];
1319
1320             testFloatEquals(
1321                 noncombustion_ptr->production_vec_kW[i] -
1322                 noncombustion_ptr->dispatch_vec_kW[i] -
1323                 noncombustion_ptr->curtailment_vec_kW[i] -
1324                 noncombustion_ptr->storage_vec_kW[i],
1325                 0,
1326                 __FILE__,
1327                 __LINE__
1328             );
1329
1330             net_load_kW -= noncombustion_ptr->production_vec_kW[i];
1331         }
1332
1333         for (size_t j = 0; j < test_model_ptr->renewable_ptr_vec.size(); j++) {
1334             renewable_ptr = test_model_ptr->renewable_ptr_vec[j];
1335
1336             testFloatEquals(
1337                 renewable_ptr->production_vec_kW[i] -
1338                 renewable_ptr->dispatch_vec_kW[i] -
1339                 renewable_ptr->curtailment_vec_kW[i] -
1340                 renewable_ptr->storage_vec_kW[i],
1341                 0,
1342                 __FILE__,
1343                 __LINE__
1344             );
1345
1346             net_load_kW -= renewable_ptr->production_vec_kW[i];
1347         }
1348
1349         for (size_t j = 0; j < test_model_ptr->storage_ptr_vec.size(); j++) {
1350             storage_ptr = test_model_ptr->storage_ptr_vec[j];
1351
1352             testTruth(
1353                 not (
1354                     storage_ptr->charging_power_vec_kW[i] > 0 and
1355                     storage_ptr->discharging_power_vec_kW[i] > 0
1356                 ),
1357                 __FILE__,
1358                 __LINE__
1359             );
1360
1361             net_load_kW -= storage_ptr->discharging_power_vec_kW[i];
1362         }
1363
1364         testLessThanOrEqualTo(

```

```

1365         net_load_kW,
1366         0,
1367         __FILE__,
1368         __LINE__
1369     );
1370 }
1371
1372 testFloatEquals(
1373     test_model_ptr->total_dispatch_discharge_kWh,
1374     2263351.62026685,
1375     __FILE__,
1376     __LINE__
1377 );
1378
1379 return;
1380 } /* testLoadBalance_Model() */

```

5.71.2.21 testPostConstructionAttributes_Model()

```

void testPostConstructionAttributes_Model (
    Model * test_model_ptr )

```

A function to check the values of various post-construction attributes.

Parameters

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

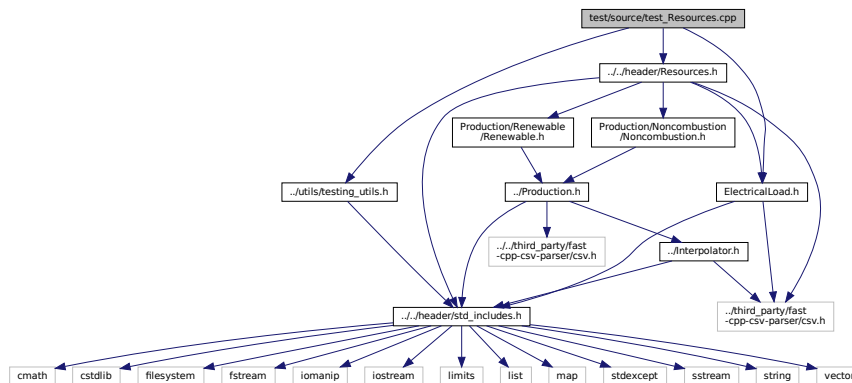
142 {
143     testFloatEquals(
144         test_model_ptr->electrical_load.n_points,
145         8760,
146         __FILE__,
147         __LINE__
148     );
149
150     testFloatEquals(
151         test_model_ptr->electrical_load.n_years,
152         0.999886,
153         __FILE__,
154         __LINE__
155     );
156
157     testFloatEquals(
158         test_model_ptr->electrical_load.min_load_kW,
159         82.1211213927802,
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals(
165         test_model_ptr->electrical_load.mean_load_kW,
166         258.373472633202,
167         __FILE__,
168         __LINE__
169     );
170
171
172     testFloatEquals(
173         test_model_ptr->electrical_load.max_load_kW,
174         500,
175         __FILE__,
176         __LINE__
177     );
178
179     return;
180 } /* testPostConstructionAttributes_Model() */

```

5.72 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

- [Resources](#) * [testConstruct_Resources](#) (void)
A function to construct a [Resources](#) object and spot check some post-construction attributes.
- void [testAddSolarResource_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)
Function to test adding a solar resource and then check the values read into the test [Resources](#) object.
- void [testBadAdd_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)
Function to test that trying to add bad resource data is being handled as expected.
- void [testAddTidalResource_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_tidal_resource_data, int tidal_resource_key)
Function to test adding a tidal resource and then check the values read into the test [Resources](#) object.
- void [testAddWaveResource_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_wave_resource_data, int wave_resource_key)
Function to test adding a wave resource and then check the values read into the test [Resources](#) object.
- void [testAddWindResource_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_wind_resource_data, int wind_resource_key)
Function to test adding a wind resource and then check the values read into the test [Resources](#) object.
- void [testAddHydroResource_Resources](#) ([Resources](#) *test_resources_ptr, [ElectricalLoad](#) *test_electrical_load_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)
Function to test adding a hydro resource and then check the values read into the test [Resources](#) object.
- int [main](#) (int argc, char **argv)

5.72.1 Detailed Description

Testing suite for [Resources](#) class.

A suite of tests for the [Resources](#) class.

5.72.2 Function Documentation

5.72.2.1 main()

```

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

    printGold("\tTesting Resources");

    srand(time(NULL));

    std::string path_2_electrical_load_time_series =
        "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

    ElectricalLoad* test_electrical_load_ptr =
        new ElectricalLoad(path_2_electrical_load_time_series);

    Resources* test_resources_ptr = testConstruct_Resources();

    try {
        int solar_resource_key = 0;
        std::string path_2_solar_resource_data =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

        testAddSolarResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_solar_resource_data,
            solar_resource_key
        );

        testBadAdd_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_solar_resource_data,
            solar_resource_key
        );

        int tidal_resource_key = 1;
        std::string path_2_tidal_resource_data =
            "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

        testAddTidalResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_tidal_resource_data,
            tidal_resource_key
        );

        int wave_resource_key = 2;
        std::string path_2_wave_resource_data =
            "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

        testAddWaveResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_wave_resource_data,
            wave_resource_key
        );

        int wind_resource_key = 3;
        std::string path_2_wind_resource_data =
            "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";

        testAddWindResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_wind_resource_data,

```



```

854         wind_resource_key
855     );
856
857
858     int hydro_resource_key = 4;
859     std::string path_2_hydro_resource_data =
860         "data/test/resources/hydro_inflow_peak-2000m3hr_1yr_dt-1hr.csv";
861
862     testAddHydroResource_Resources(
863         test_resources_ptr,
864         test_electrical_load_ptr,
865         path_2_hydro_resource_data,
866         hydro_resource_key
867     );
868 }
869
870
871 catch (...) {
872     delete test_electrical_load_ptr;
873     delete test_resources_ptr;
874
875     printGold(" ..... ");
876     printRed("FAIL");
877     std::cout << std::endl;
878     throw;
879 }
880
881
882 delete test_electrical_load_ptr;
883 delete test_resources_ptr;
884
885 printGold(" ..... ");
886 printGreen("PASS");
887 std::cout << std::endl;
888 return 0;
889 } /* main() */

```

5.72.2.2 testAddHydroResource_Resources()

```

void testAddHydroResource_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_hydro_resource_data,
    int hydro_resource_key )

```

Function to test adding a hydro resource and then check the values read into the test [Resources](#) object.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_hydro_resource_data</i>	A path (either relative or absolute) to the hydro resource data.
<i>hydro_resource_key</i>	A key used to index into the Resources component of the test Resources object.

```

705 {
706     test_resources_ptr->addResource(
707         NoncombustionType::HYDRO,
708         path_2_hydro_resource_data,
709         hydro_resource_key,
710         test_electrical_load_ptr
711     );
712
713     std::vector<double> expected_hydro_resource_vec_m3hr = {
714         2167.91531556942,
715         2046.58261560569,
716         2007.85941123153,
717         2000.11477247929,
718         1917.50527264453,
719         1963.97311577093,
720         1908.46985899809,
721         1886.5267112678,

```

```

722         1965.26388854254,
723         1953.64692935289,
724         2084.01504296306,
725         2272.46796101188,
726         2520.29645627096,
727         2715.203242423,
728         2720.36633563203,
729         3130.83228077221,
730         3289.59741021591,
731         3981.45195965772,
732         5295.45929491303,
733         7084.47124360523,
734         7709.20557708454,
735         7436.85238642936,
736         7235.49173429668,
737         6710.14695517339,
738         6015.71085806577,
739         5279.97001316337,
740         4877.24870889801,
741         4421.60569340303,
742         3919.49483690424,
743         3498.70270322341,
744         3274.10813058883,
745         3147.61233529349,
746         2904.94693324343,
747         2805.55738101,
748         2418.32535637171,
749         2398.96375630723,
750         2260.85100182222,
751         2157.58912702878,
752         2019.47637254377,
753         1913.63295220712,
754         1863.29279076589,
755         1748.41395678279,
756         1695.49224555317,
757         1599.97501375715,
758         1559.96103873397,
759         1505.74855473274,
760         1438.62833664765,
761         1384.41585476901
762     };
763
764     for (size_t i = 0; i < expected_hydro_resource_vec_m3hr.size(); i++) {
765         testFloatEquals(
766             test_resources_ptr->resource_map_1D[hydro_resource_key][i],
767             expected_hydro_resource_vec_m3hr[i],
768             __FILE__,
769             __LINE__
770         );
771     }
772
773     return;
774 } /* testAddHydroResource_Resources() */

```

5.72.2.3 testAddSolarResource_Resources()

```

void testAddSolarResource_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_solar_resource_data,
    int solar_resource_key )

```

Function to test adding a solar resource and then check the values read into the test [Resources](#) object.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_solar_resource_data</i>	A path (either relative or absolute) to the solar resource data.
<i>solar_resource_key</i>	A key used to index into the Resources component of the test Resources object.

```

132 {
133     test_resources_ptr->addResource(
134         RenewableType::SOLAR,
135         path_2_solar_resource_data,
136         solar_resource_key,
137         test_electrical_load_ptr
138     );
139
140     std::vector<double> expected_solar_resource_vec_kWm2 = {
141         0,
142         0,
143         0,
144         0,
145         0,
146         0,
147         8.51702662684015E-05,
148         0.000348341567045,
149         0.00213793728593,
150         0.004099863613322,
151         0.000997135230553,
152         0.009534527624657,
153         0.022927996790616,
154         0.0136071715294,
155         0.002535134127751,
156         0.005206897515821,
157         0.005627658648597,
158         0.000701186722215,
159         0.00017119827089,
160         0,
161         0,
162         0,
163         0,
164         0,
165         0,
166         0,
167         0,
168         0,
169         0,
170         0,
171         0,
172         0.000141055102242,
173         0.00084525014743,
174         0.024893647822702,
175         0.091245556190749,
176         0.158722176731637,
177         0.152859680515876,
178         0.149922903895116,
179         0.13049996570866,
180         0.03081254222795,
181         0.001218928911125,
182         0.000206092647423,
183         0,
184         0,
185         0,
186         0,
187         0,
188         0
189     };
190
191     for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
192         testFloatEquals(
193             test_resources_ptr->resource_map_1D[solar_resource_key][i],
194             expected_solar_resource_vec_kWm2[i],
195             __FILE__,
196             __LINE__
197         );
198     }
199
200     return;
201 } /* testAddSolarResource_Resources() */

```

5.72.2.4 testAddTidalResource_Resources()

```

void testAddTidalResource_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_tidal_resource_data,
    int tidal_resource_key )

```

Function to test adding a tidal resource and then check the values read into the test [Resources](#) object.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_tidal_resource_data</i>	A path (either relative or absolute) to the tidal resource data.
<i>tidal_resource_key</i>	A key used to index into the Resources component of the test Resources object.

```

332 {
333     test_resources_ptr->addResource(
334         RenewableType::TIDAL,
335         path_2_tidal_resource_data,
336         tidal_resource_key,
337         test_electrical_load_ptr
338     );
339
340     std::vector<double> expected_tidal_resource_vec_ms = {
341         0.347439913040533,
342         0.770545522195602,
343         0.731352084836198,
344         0.293389814389542,
345         0.209959110813115,
346         0.610609623896497,
347         1.78067162013604,
348         2.53522775118089,
349         2.75966627832024,
350         2.52101111143895,
351         2.05389330201031,
352         1.3461515862445,
353         0.28909254878384,
354         0.897754086048563,
355         1.71406453837407,
356         1.85047408742869,
357         1.71507908595979,
358         1.33540349705416,
359         0.434586143463003,
360         0.500623815700637,
361         1.37172172646733,
362         1.68294125491228,
363         1.56101300975417,
364         1.04925834219412,
365         0.211395463930223,
366         1.03720048903385,
367         1.85059536356448,
368         1.85203242794517,
369         1.4091471616277,
370         0.767776539039899,
371         0.251464906990961,
372         1.47018469375652,
373         2.36260493698197,
374         2.46653750048625,
375         2.12851908739291,
376         1.62783753197988,
377         0.734594890957439,
378         0.441886297300355,
379         1.6574418350918,
380         2.0684558286637,
381         1.87717416992136,
382         1.58871262337931,
383         1.03451227609235,
384         0.193371305159817,
385         0.976400122458815,
386         1.6583227369707,
387         1.76690616570953,
388         1.54801328553115
389     };
390
391     for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
392         testFloatEquals(
393             test_resources_ptr->resource_map_1D[tidal_resource_key][i],
394             expected_tidal_resource_vec_ms[i],
395             __FILE__,
396             __LINE__
397         );
398     }
399
400     return;
401 } /* testAddTidalResource_Resources() */

```

5.72.2.5 testAddWaveResource_Resources()

```
void testAddWaveResource_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_wave_resource_data,
    int wave_resource_key )
```

Function to test adding a wave resource and then check the values read into the test [Resources](#) object.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_wave_resource_data</i>	A path (either relative or absolute) to the wave resource data.
<i>wave_resource_key</i>	A key used to index into the Resources component of the test Resources object.

```
437 {
438     test_resources_ptr->addResource(
439         RenewableType::WAVE,
440         path_2_wave_resource_data,
441         wave_resource_key,
442         test_electrical_load_ptr
443     );
444
445     std::vector<double> expected_significant_wave_height_vec_m = {
446         4.26175222125028,
447         4.25020976167872,
448         4.25656524330349,
449         4.27193854786718,
450         4.28744955711233,
451         4.29421815278154,
452         4.2839937266082,
453         4.25716982457976,
454         4.22419391611483,
455         4.19588925217606,
456         4.17338788587412,
457         4.14672746914214,
458         4.10560041173665,
459         4.05074966447193,
460         3.9953696962433,
461         3.95316976150866,
462         3.92771018142378,
463         3.91129562488595,
464         3.89558312094911,
465         3.87861093931749,
466         3.86538307240754,
467         3.86108961027929,
468         3.86459448853189,
469         3.86796474016882,
470         3.86357412779993,
471         3.85554872014731,
472         3.86044266668675,
473         3.89445961915999,
474         3.95554798115731,
475         4.02265508610476,
476         4.07419587011404,
477         4.10314247143958,
478         4.11738045085928,
479         4.12554995596708,
480         4.12923992001675,
481         4.1229292327442,
482         4.10123955307441,
483         4.06748827895363,
484         4.0336230651344,
485         4.01134236393876,
486         4.00136570034559,
487         3.99368787690411,
488         3.97820924247644,
489         3.95369335178055,
490         3.92742545608532,
491         3.90683362771686,
492         3.89331520944006,
493         3.88256045801583
494     };
495
496     std::vector<double> expected_energy_period_vec_s = {
```

```

497         10.4456008226821,
498         10.4614151137651,
499         10.4462827795433,
500         10.4127692097884,
501         10.3734397942723,
502         10.3408599227669,
503         10.32637292093,
504         10.3245412676322,
505         10.310409818185,
506         10.2589529840966,
507         10.1728100603103,
508         10.0862908658929,
509         10.03480243813,
510         10.023673635806,
511         10.0243418565116,
512         10.0063487117653,
513         9.96050302286607,
514         9.9011999635568,
515         9.84451822125472,
516         9.79726875879626,
517         9.75614594835158,
518         9.7173447961368,
519         9.68342904390577,
520         9.66380508567062,
521         9.6674009575699,
522         9.68927134575103,
523         9.70979984863046,
524         9.70967357906908,
525         9.68983025704562,
526         9.6722855524805,
527         9.67973599910003,
528         9.71977125328293,
529         9.78450442291421,
530         9.86532355233449,
531         9.96158937600019,
532         10.0807018356507,
533         10.2291022504937,
534         10.39458528356,
535         10.5464393581004,
536         10.6553277500484,
537         10.7245553190084,
538         10.7893127285064,
539         10.8846512240849,
540         11.0148158739075,
541         11.1544325654719,
542         11.2772785848343,
543         11.3744362756187,
544         11.4533643503183
545     };
546
547     for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
548         testFloatEquals(
549             test_resources_ptr->resource_map_2D[wave_resource_key][i][0],
550             expected_significant_wave_height_vec_m[i],
551             __FILE__,
552             __LINE__
553         );
554
555         testFloatEquals(
556             test_resources_ptr->resource_map_2D[wave_resource_key][i][1],
557             expected_energy_period_vec_s[i],
558             __FILE__,
559             __LINE__
560         );
561     }
562
563     return;
564 } /* testAddWaveResource_Resources() */

```

5.72.2.6 testAddWindResource_Resources()

```

void testAddWindResource_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_wind_resource_data,
    int wind_resource_key )

```

Function to test adding a wind resource and then check the values read into the test [Resources](#) object.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_wind_resource_data</i>	A path (either relative or absolute) to the wind resource data.
<i>wind_resource_key</i>	A key used to index into the Resources component of the test Resources object.

```

600 {
601     test_resources_ptr->addResource(
602         RenewableType::WIND,
603         path_2_wind_resource_data,
604         wind_resource_key,
605         test_electrical_load_ptr
606     );
607
608     std::vector<double> expected_wind_resource_vec_ms = {
609         6.8856688469997,
610         5.02177105466549,
611         3.74211715899568,
612         5.67169579985362,
613         4.90670669971858,
614         4.29586955031368,
615         7.41155377205065,
616         10.2243290476943,
617         13.1258696725555,
618         13.7016198628274,
619         16.2481482330233,
620         16.5096744355418,
621         13.4354482206162,
622         14.0129230731609,
623         14.5554549260515,
624         13.4454539065912,
625         13.3447169512094,
626         11.7372615098554,
627         12.7200070078013,
628         10.6421127908149,
629         6.09869498990661,
630         5.66355596602321,
631         4.97316966910831,
632         3.48937138360567,
633         2.15917470979169,
634         1.29061103587027,
635         3.43475751425219,
636         4.11706326260927,
637         4.28905275747408,
638         5.75850263196241,
639         8.98293663055264,
640         11.7069822941315,
641         12.4031987075858,
642         15.4096570910089,
643         16.6210843829552,
644         13.3421219142573,
645         15.2112831900548,
646         18.350864533037,
647         15.8751799822971,
648         15.3921198799796,
649         15.9729192868434,
650         12.4728950178772,
651         10.177050481096,
652         10.7342247355551,
653         8.98846695631389,
654         4.14671169124739,
655         3.17256452697149,
656         3.40036336968628
657     };
658
659     for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
660         testFloatEquals(
661             test_resources_ptr->resource_map_1D[wind_resource_key][i],
662             expected_wind_resource_vec_ms[i],
663             __FILE__,
664             __LINE__
665         );
666     }
667
668     return;
669 } /* testAddWindResource_Resources() */

```


5.72.2.7 testBadAdd_Resources()

```
void testBadAdd_Resources (
    Resources * test_resources_ptr,
    ElectricalLoad * test_electrical_load_ptr,
    std::string path_2_solar_resource_data,
    int solar_resource_key )
```

Function to test that trying to add bad resource data is being handled as expected.

Parameters

<i>test_resources_ptr</i>	A pointer to the test Resources object.
<i>test_electrical_load_ptr</i>	A pointer to the test ElectricalLoad object.
<i>path_2_solar_resource_data</i>	A path (either relative or absolute) to the given solar resource data.
<i>solar_resource_key</i>	A key for indexing into the test Resources object.

```
236 {
237     bool error_flag = true;
238
239     try {
240         test_resources_ptr->addResource(
241             RenewableType::SOLAR,
242             path_2_solar_resource_data,
243             solar_resource_key,
244             test_electrical_load_ptr
245         );
246
247         error_flag = false;
248     } catch (...) {
249         // Task failed successfully! =P
250     }
251     if (not error_flag) {
252         expectedErrorNotDetected(__FILE__, __LINE__);
253     }
254
255
256     try {
257         std::string path_2_solar_resource_data_BAD_TIMES =
258             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";
259
260         test_resources_ptr->addResource(
261             RenewableType::SOLAR,
262             path_2_solar_resource_data_BAD_TIMES,
263             -1,
264             test_electrical_load_ptr
265         );
266
267         error_flag = false;
268     } catch (...) {
269         // Task failed successfully! =P
270     }
271     if (not error_flag) {
272         expectedErrorNotDetected(__FILE__, __LINE__);
273     }
274
275
276     try {
277         std::string path_2_solar_resource_data_BAD_LENGTH =
278             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";
279
280         test_resources_ptr->addResource(
281             RenewableType::SOLAR,
282             path_2_solar_resource_data_BAD_LENGTH,
283             -2,
284             test_electrical_load_ptr
285         );
286
287         error_flag = false;
288     } catch (...) {
289         // Task failed successfully! =P
290     }
291     if (not error_flag) {
292         expectedErrorNotDetected(__FILE__, __LINE__);
293     }
294
295     return;
```

```
296 }    /* testBadAdd_Resources() */
```

5.72.2.8 testConstruct_Resources()

```
Resources * testConstruct_Resources (
    void )
```

A function to construct a [Resources](#) object and spot check some post-construction attributes.

Returns

A pointer to a test [Resources](#) object.

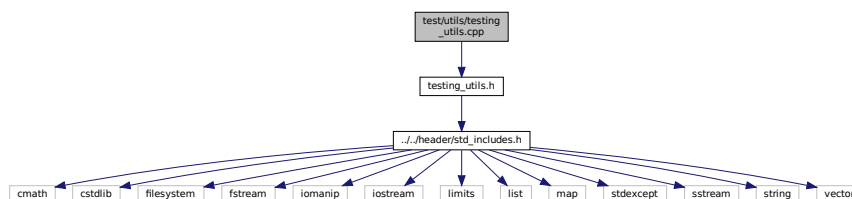
```
64 {
65     Resources* test_resources_ptr = new Resources();
66
67     testFloatEquals(
68         test_resources_ptr->resource_map_1D.size(),
69         0,
70         __FILE__,
71         __LINE__
72     );
73
74     testFloatEquals(
75         test_resources_ptr->path_map_1D.size(),
76         0,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_resources_ptr->resource_map_2D.size(),
83         0,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         test_resources_ptr->path_map_2D.size(),
90         0,
91         __FILE__,
92         __LINE__
93     );
94
95     return test_resources_ptr;
96 }    /* testConstruct_Resources() */
```

5.73 test/utlis/testing_utils.cpp File Reference

Implementation 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.

5.73.1 Detailed Description

Implementation file for various PGMcpp testing utilities.

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

5.73.2 Function Documentation

5.73.2.1 `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> ").

```
457 {
458     std::string error_str = "\n ERROR   failed to throw expected error prior to line ";
459     error_str += std::to_string(line);
```

```

460     error_str += " of ";
461     error_str += file;
462
463     #ifdef _WIN32
464         std::cout << error_str << std::endl;
465     #endif
466
467     throw std::runtime_error(error_str);
468     return;
469 } /* expectedErrorNotDetected() */

```

5.73.2.2 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.
------------------	---

```

109 {
110     std::cout << "\x1B[33m" << input_str << "\033[0m";
111     return;
112 } /* printGold() */

```

5.73.2.3 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.
------------------	---

```

89 {
90     std::cout << "\x1B[32m" << input_str << "\033[0m";
91     return;
92 } /* printGreen() */

```

5.73.2.4 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> .
------------------	---

```

129 {
130     std::cout << "\x1B[31m" << input_str << "\033[0m";
131     return;
132 } /* printRed() */

```

5.73.2.5 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> ").

```

163 {
164     if (fabs(x - y) <= FLOAT_TOLERANCE) {
165         return;
166     }
167
168     std::string error_str = "ERROR: testFloatEquals():\t in ";
169     error_str += file;
170     error_str += "\tline ";
171     error_str += std::to_string(line);
172     error_str += ":\t\n";
173     error_str += std::to_string(x);
174     error_str += " and ";
175     error_str += std::to_string(y);
176     error_str += " are not equal to within +/- ";
177     error_str += std::to_string(FLOAT_TOLERANCE);
178     error_str += "\n";
179
180     #ifdef _WIN32
181         std::cout << error_str << std::endl;
182     #endif
183
184     throw std::runtime_error(error_str);
185     return;
186 } /* testFloatEquals() */

```

5.73.2.6 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__").

```

216 {
217     if (x > y) {
218         return;
219     }
220
221     std::string error_str = "ERROR: testGreaterThan():\t in ";
222     error_str += file;
223     error_str += "\tline ";
224     error_str += std::to_string(line);
225     error_str += ":\t\n";
226     error_str += std::to_string(x);
227     error_str += " is not greater than ";
228     error_str += std::to_string(y);
229     error_str += "\n";
230
231     #ifdef _WIN32
232         std::cout << error_str << std::endl;
233     #endif
234
235     throw std::runtime_error(error_str);
236     return;
237 } /* testGreaterThan() */

```

5.73.2.7 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__").

```

267 {
268     if (x >= y) {
269         return;
270     }
271
272     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
273     error_str += file;
274     error_str += "\tline ";
275     error_str += std::to_string(line);
276     error_str += ":\t\n";
277     error_str += std::to_string(x);
278     error_str += " is not greater than or equal to ";
279     error_str += std::to_string(y);
280     error_str += "\n";
281
282     #ifdef _WIN32
283         std::cout << error_str << std::endl;
284     #endif
285
286     throw std::runtime_error(error_str);

```

```

287     return;
288 } /* testGreaterThanOrEqualTo() */

```

5.73.2.8 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__").

```

318 {
319     if (x < y) {
320         return;
321     }
322
323     std::string error_str = "ERROR: testLessThan():\t in ";
324     error_str += file;
325     error_str += "\tline ";
326     error_str += std::to_string(line);
327     error_str += ":\t\n";
328     error_str += std::to_string(x);
329     error_str += " is not less than ";
330     error_str += std::to_string(y);
331     error_str += "\n";
332
333     #ifdef _WIN32
334         std::cout << error_str << std::endl;
335     #endif
336
337     throw std::runtime_error(error_str);
338     return;
339 } /* testLessThan() */

```

5.73.2.9 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__").

```

369 {
370     if (x <= y) {
371         return;
372     }
373
374     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
375     error_str += file;
376     error_str += "\tline ";
377     error_str += std::to_string(line);
378     error_str += ":\t\n";
379     error_str += std::to_string(x);
380     error_str += " is not less than or equal to ";
381     error_str += std::to_string(y);
382     error_str += "\n";
383
384     #ifdef _WIN32
385         std::cout << error_str << std::endl;
386     #endif
387
388     throw std::runtime_error(error_str);
389     return;
390 } /* testLessThanOrEqualTo() */

```

5.73.2.10 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__").

```

417 {
418     if (statement) {
419         return;
420     }
421
422     std::string error_str = "ERROR: testTruth():\t in ";
423     error_str += file;
424     error_str += "\tline ";
425     error_str += std::to_string(line);
426     error_str += ":\t\n";
427     error_str += "Given statement is not true";
428
429     #ifdef _WIN32
430         std::cout << error_str << std::endl;
431     #endif
432
433     throw std::runtime_error(error_str);
434     return;
435 } /* testTruth() */

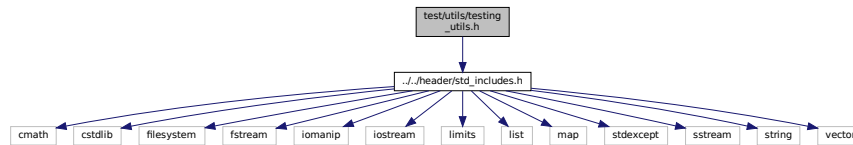
```

5.74 test/utls/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.

5.74.1 Detailed Description

Header file for various PGMcpp testing utilities.

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

5.74.2 Macro Definition Documentation

5.74.2.1 FLOAT_TOLERANCE

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

A tolerance for application to floating point equality tests.

5.74.3 Function Documentation

5.74.3.1 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__").

```
457 {
458     std::string error_str = "\n ERROR   failed to throw expected error prior to line ";
459     error_str += std::to_string(line);
460     error_str += " of ";
461     error_str += file;
462
463     #ifdef _WIN32
464         std::cout << error_str << std::endl;
465     #endif
466
467     throw std::runtime_error(error_str);
468     return;
469 } /* expectedErrorNotDetected() */
```

5.74.3.2 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.
------------------	---

```

109 {
110     std::cout << "\x1B[33m" << input_str << "\033[0m";
111     return;
112 } /* printGold() */

```

5.74.3.3 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.
------------------	---

```

89 {
90     std::cout << "\x1B[32m" << input_str << "\033[0m";
91     return;
92 } /* printGreen() */

```

5.74.3.4 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.
------------------	---

```

129 {
130     std::cout << "\x1B[31m" << input_str << "\033[0m";
131     return;
132 } /* printRed() */

```

5.74.3.5 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__").

```

163 {
164     if (fabs(x - y) <= FLOAT_TOLERANCE) {
165         return;
166     }
167
168     std::string error_str = "ERROR: testFloatEquals():\t in ";
169     error_str += file;
170     error_str += "\tline ";
171     error_str += std::to_string(line);
172     error_str += ":\t\n";
173     error_str += std::to_string(x);
174     error_str += " and ";
175     error_str += std::to_string(y);
176     error_str += " are not equal to within +/- ";
177     error_str += std::to_string(FLOAT_TOLERANCE);
178     error_str += "\n";
179
180     #ifdef _WIN32
181         std::cout << error_str << std::endl;
182     #endif
183
184     throw std::runtime_error(error_str);
185     return;
186 } /* testFloatEquals() */

```

5.74.3.6 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__").

```

216 {
217     if (x > y) {
218         return;
219     }
220
221     std::string error_str = "ERROR: testGreaterThan():\t in ";
222     error_str += file;
223     error_str += "\tline ";
224     error_str += std::to_string(line);
225     error_str += ":\t\n";
226     error_str += std::to_string(x);
227     error_str += " is not greater than ";
228     error_str += std::to_string(y);
229     error_str += "\n";
230
231     #ifdef _WIN32
232         std::cout << error_str << std::endl;
233     #endif
234

```

```

235     throw std::runtime_error(error_str);
236     return;
237 } /* testGreaterThan() */

```

5.74.3.7 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__").

```

267 {
268     if (x >= y) {
269         return;
270     }
271
272     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
273     error_str += file;
274     error_str += "\tline ";
275     error_str += std::to_string(line);
276     error_str += ":\t\n";
277     error_str += std::to_string(x);
278     error_str += " is not greater than or equal to ";
279     error_str += std::to_string(y);
280     error_str += "\n";
281
282     #ifdef _WIN32
283         std::cout << error_str << std::endl;
284     #endif
285
286     throw std::runtime_error(error_str);
287     return;
288 } /* testGreaterThanOrEqualTo() */

```

5.74.3.8 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__").

```

318 {
319     if (x < y) {
320         return;
321     }
322
323     std::string error_str = "ERROR: testLessThan():\t in ";
324     error_str += file;
325     error_str += "\tline ";
326     error_str += std::to_string(line);
327     error_str += ":\t\n";
328     error_str += std::to_string(x);
329     error_str += " is not less than ";
330     error_str += std::to_string(y);
331     error_str += "\n";
332
333     #ifdef _WIN32
334         std::cout << error_str << std::endl;
335     #endif
336
337     throw std::runtime_error(error_str);
338     return;
339 } /* testLessThan() */

```

5.74.3.9 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__").

```

369 {
370     if (x <= y) {
371         return;
372     }
373
374     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
375     error_str += file;
376     error_str += "\tline ";
377     error_str += std::to_string(line);
378     error_str += ":\t\n";
379     error_str += std::to_string(x);
380     error_str += " is not less than or equal to ";
381     error_str += std::to_string(y);
382     error_str += "\n";
383
384     #ifdef _WIN32
385         std::cout << error_str << std::endl;
386     #endif
387
388     throw std::runtime_error(error_str);
389     return;
390 } /* testLessThanOrEqualTo() */

```

5.74.3.10 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__").

```
417 {  
418     if (statement) {  
419         return;  
420     }  
421  
422     std::string error_str = "ERROR: testTruth():\t in ";  
423     error_str += file;  
424     error_str += "\tline ";  
425     error_str += std::to_string(line);  
426     error_str += ":\t\n";  
427     error_str += "Given statement is not true";  
428  
429     #ifdef _WIN32  
430         std::cout << error_str << std::endl;  
431     #endif  
432  
433     throw std::runtime_error(error_str);  
434     return;  
435 } /* testTruth() */
```


Bibliography

- G.S. Bir, M.J. Lawson, and Y. Li. Structural Design of a Horizontal-Axis Tidal Current Turbine Composite Blade. *NREL*, 2011. URL https://www.researchgate.net/publication/239886961_Structural_Design_of_a_Horizontal-Axis-Tidal-Current-Turbine-Composite-Blade. 234, 235
- 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. 235
- CIMAC. Guide to Diesel Exhaust Emissions Control of NO_x, SO_x, Particulates, Smoke, and CO₂. Technical report, Conseil International des Machines à Combustion, 2008. Included: docs/refs/diesel_emissions_ref_2.pdf. 61
- HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 172, 221
- HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 16, 159, 172, 173, 220, 221
- HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 52, 61
- HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 52, 61
- HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 52, 61
- 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. 211
- HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 172, 221
- HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 173, 220
- HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 172, 221
- W. Jakob. pybind11 — Seamless operability between C++11 and Python, 2023. URL <https://pybind11.readthedocs.io/en/stable/>. 305, 306, 309, 312, 315, 317, 322, 323, 325, 327, 329, 331, 333, 334, 336, 337, 339, 342
- M. Lewis, R.O. Murray, S. Fredriksson, J. Maskell, A. de Fockert, S.P. Neill, and P.E. Robins. A standardised tidal-stream power curve, optimised for the global resource. *Renewable Energy*, 2021. doi: 10.1016/j.renene.2021.02.032. URL https://www.researchgate.net/publication/349341552_A_standardised_tidal-stream_power_curve_optimised_for_the_global_resource. 234, 235

- 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. [237](#), [252](#)
- P. Milan, M. Wächter, S. Barth, and J. Peinke. Power curves for wind turbines. *Wind Power Generation and Wind Turbine Design*, page 595–612, 2010. doi: 10.2495/978-1-84564-205-1/18. [266](#)
- NRCan. Auto\$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. [61](#)
- 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. [251](#)
- A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023a. Included: docs/refs/battery_degradation.pdf. [117](#), [119](#), [130](#)
- A. Truelove. Hydro Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023b. Included: docs/refs/hydro.pdf. [76](#), [78](#), [79](#), [80](#), [82](#)
- A. Truelove, Dr. B. Buckham, Dr. C. Crawford, and C. Hiles. Scaling Technology Models for HOMER Pro: Wind, Tidal Stream, and Wave. Technical report, PRIMED, 2019. Included: docs/refs/wind_tidal_wave.pdf. [235](#), [250](#), [267](#)
- D. van Heesch. Doxygen: Generate documentation from source code, 2023. URL <https://www.doxygen.nl>. [278](#)
- B. Whitby and C.E. Ugalde-Loo. Performance of Pitch and Stall Regulated Tidal Stream Turbines. *IEEE Transactions on Sustainable Energy*, 5(1), 2013. doi: 10.1109/TSTE.2013.2272653. [235](#)
- U. Zafar. Literature Review of Wind Turbines. *Bauhaus Universität*, 2018. URL https://www.researchgate.net/publication/329680977_Literature_Review_of_Wind_Turbines. [266](#)

Index

- __applyCycleChargingControl_CHARGING
Controller, [28](#)
- __applyCycleChargingControl_DISCHARGING
Controller, [28](#)
- __applyLoadFollowingControl_CHARGING
Controller, [30](#)
- __applyLoadFollowingControl_DISCHARGING
Controller, [31](#)
- __checkBounds1D
Interpolator, [94](#)
- __checkBounds2D
Interpolator, [95](#)
- __checkDataKey1D
Interpolator, [96](#)
- __checkDataKey2D
Interpolator, [96](#)
- __checkInputs
Combustion, [14](#)
Diesel, [49](#)
Hydro, [75](#)
Lilon, [114](#)
Model, [137](#)
Noncombustion, [157](#)
Production, [167](#)
Renewable, [184](#)
Solar, [207](#)
Storage, [218](#)
Tidal, [234](#)
Wave, [249](#)
Wind, [265](#)
- __checkNormalizedProduction
Production, [168](#)
- __checkResourceKey1D
Resources, [192](#)
- __checkResourceKey2D
Resources, [193](#)
- __checkTimePoint
Production, [169](#)
Resources, [194](#)
- __computeCubicProductionkW
Tidal, [234](#)
Wind, [266](#)
- __computeEconomics
Model, [138](#)
- __computeExponentialProductionkW
Tidal, [235](#)
Wind, [267](#)
- __computeFuelAndEmissions
Model, [138](#)
- __computeGaussianProductionkW
Wave, [250](#)
- __computeLevellizedCostOfEnergy
Model, [139](#)
- __computeLookupProductionkW
Tidal, [236](#)
Wave, [251](#)
Wind, [267](#)
- __computeNetLoad
Controller, [32](#)
- __computeNetPresentCost
Model, [139](#)
- __computeParaboloidProductionkW
Wave, [251](#)
- __computeRealDiscountAnnual
Storage, [219](#)
- __constructCombustionMap
Controller, [33](#)
- __flowToPower
Hydro, [76](#)
- __getAcceptableFlow
Hydro, [76](#)
- __getAvailableFlow
Hydro, [77](#)
- __getBcal
Lilon, [116](#)
- __getDataStringMatrix
Interpolator, [97](#)
- __getEacal
Lilon, [117](#)
- __getEfficiencyFactor
Hydro, [77](#)
- __getGenericCapitalCost
Diesel, [51](#)
Hydro, [78](#)
Lilon, [117](#)
Solar, [207](#)
Tidal, [236](#)
Wave, [252](#)
Wind, [268](#)
- __getGenericFuelIntercept
Diesel, [51](#)
- __getGenericFuelSlope
Diesel, [52](#)
- __getGenericOpMaintCost
Diesel, [52](#)
Hydro, [78](#)
Lilon, [118](#)
Solar, [208](#)

- Tidal, [237](#)
- Wave, [252](#)
- Wind, [268](#)
- __getInterpolationIndex
 - Interpolator, [97](#)
- __getMaximumFlowm3hr
 - Hydro, [79](#)
- __getMinimumFlowm3hr
 - Hydro, [79](#)
- __getRenewableProduction
 - Controller, [34](#)
- __handleCombustionDispatch
 - Controller, [36](#)
- __handleDegradation
 - Lilon, [118](#)
- __handleNoncombustionDispatch
 - Controller, [37](#)
- __handleStartStop
 - Diesel, [53](#)
 - Noncombustion, [157](#)
 - Renewable, [184](#)
- __handleStorageCharging
 - Controller, [38](#), [39](#)
- __handleStorageDischarging
 - Controller, [41](#)
- __initInterpolator
 - Hydro, [79](#)
- __isNonNumeric
 - Interpolator, [98](#)
- __modelDegradation
 - Lilon, [119](#)
- __powerToFlow
 - Hydro, [81](#)
- __readData1D
 - Interpolator, [98](#)
- __readData2D
 - Interpolator, [99](#)
- __readHydroResource
 - Resources, [194](#)
- __readNormalizedProductionData
 - Production, [169](#)
- __readSolarResource
 - Resources, [195](#)
- __readTidalResource
 - Resources, [196](#)
- __readWaveResource
 - Resources, [197](#)
- __readWindResource
 - Resources, [198](#)
- __splitCommaSeparatedString
 - Interpolator, [101](#)
- __throwLengthError
 - Production, [170](#)
 - Resources, [199](#)
- __throwReadError
 - Interpolator, [102](#)
- __toggleDepleted
 - Lilon, [119](#)
- __updateState
 - Hydro, [82](#)
- __writeSummary
 - Combustion, [14](#)
 - Diesel, [54](#)
 - Hydro, [83](#)
 - Lilon, [120](#)
 - Model, [140](#)
 - Noncombustion, [158](#)
 - Renewable, [185](#)
 - Solar, [208](#)
 - Storage, [220](#)
 - Tidal, [237](#)
 - Wave, [252](#)
 - Wind, [269](#)
- __writeTimeSeries
 - Combustion, [15](#)
 - Diesel, [56](#)
 - Hydro, [84](#)
 - Lilon, [121](#)
 - Model, [143](#)
 - Noncombustion, [158](#)
 - Renewable, [185](#)
 - Solar, [209](#)
 - Storage, [220](#)
 - Tidal, [239](#)
 - Wave, [254](#)
 - Wind, [270](#)
- ~Combustion
 - Combustion, [13](#)
- ~Controller
 - Controller, [27](#)
- ~Diesel
 - Diesel, [49](#)
- ~ElectricalLoad
 - ElectricalLoad, [65](#)
- ~Hydro
 - Hydro, [75](#)
- ~Interpolator
 - Interpolator, [94](#)
- ~Lilon
 - Lilon, [114](#)
- ~Model
 - Model, [137](#)
- ~Noncombustion
 - Noncombustion, [157](#)
- ~Production
 - Production, [167](#)
- ~Renewable
 - Renewable, [184](#)
- ~Resources
 - Resources, [191](#)
- ~Solar
 - Solar, [207](#)
- ~Storage
 - Storage, [218](#)
- ~Tidal
 - Tidal, [234](#)

- ~Wave
 - Wave, [249](#)
- ~Wind
 - Wind, [265](#)
- addData1D
 - Interpolator, [102](#)
- addData2D
 - Interpolator, [103](#)
- addDiesel
 - Model, [144](#)
- addHydro
 - Model, [144](#)
- addLilon
 - Model, [145](#)
- addResource
 - Model, [145](#), [146](#)
 - Resources, [200](#), [201](#)
- addSolar
 - Model, [146](#)
- addTidal
 - Model, [147](#)
- addWave
 - Model, [147](#)
- addWind
 - Model, [147](#)
- applyDispatchControl
 - Controller, [41](#)
- capacity_kW
 - Production, [174](#)
 - ProductionInputs, [180](#)
- capital_cost
 - DieselInputs, [61](#)
 - HydroInputs, [91](#)
 - LilonInputs, [131](#)
 - Production, [174](#)
 - SolarInputs, [214](#)
 - Storage, [223](#)
 - TidalInputs, [244](#)
 - WaveInputs, [260](#)
 - WindInputs, [276](#)
- capital_cost_vec
 - Production, [174](#)
 - Storage, [224](#)
- CH4_emissions_intensity_kgL
 - Combustion, [20](#)
 - DieselInputs, [61](#)
- CH4_emissions_vec_kg
 - Combustion, [20](#)
- CH4_kg
 - Emissions, [69](#)
- charge_kWh
 - Storage, [224](#)
- charge_vec_kWh
 - Storage, [224](#)
- charging_efficiency
 - Lilon, [125](#)
 - LilonInputs, [131](#)
- charging_power_vec_kW
 - Storage, [224](#)
- clear
 - Controller, [43](#)
 - ElectricalLoad, [65](#)
 - Model, [148](#)
 - Resources, [202](#)
- CO2_emissions_intensity_kgL
 - Combustion, [20](#)
 - DieselInputs, [61](#)
- CO2_emissions_vec_kg
 - Combustion, [20](#)
- CO2_kg
 - Emissions, [69](#)
- CO_emissions_intensity_kgL
 - Combustion, [20](#)
 - DieselInputs, [62](#)
- CO_emissions_vec_kg
 - Combustion, [20](#)
- CO_kg
 - Emissions, [69](#)
- Combustion, [9](#)
 - __checkInputs, [14](#)
 - __writeSummary, [14](#)
 - __writeTimeSeries, [15](#)
 - ~Combustion, [13](#)
 - CH4_emissions_intensity_kgL, [20](#)
 - CH4_emissions_vec_kg, [20](#)
 - CO2_emissions_intensity_kgL, [20](#)
 - CO2_emissions_vec_kg, [20](#)
 - CO_emissions_intensity_kgL, [20](#)
 - CO_emissions_vec_kg, [20](#)
 - Combustion, [12](#)
 - commit, [15](#)
 - computeEconomics, [16](#)
 - computeFuelAndEmissions, [16](#)
 - cycle_charging_setpoint, [21](#)
 - fuel_consumption_vec_L, [21](#)
 - fuel_cost_L, [21](#)
 - fuel_cost_vec, [21](#)
 - fuel_mode, [21](#)
 - fuel_mode_str, [21](#)
 - getEmissionskg, [17](#)
 - getFuelConsumptionL, [17](#)
 - handleReplacement, [18](#)
 - linear_fuel_intercept_LkWh, [22](#)
 - linear_fuel_slope_LkWh, [22](#)
 - nominal_fuel_escalation_annual, [22](#)
 - NOx_emissions_intensity_kgL, [22](#)
 - NOx_emissions_vec_kg, [22](#)
 - PM_emissions_intensity_kgL, [22](#)
 - PM_emissions_vec_kg, [23](#)
 - real_fuel_escalation_annual, [23](#)
 - requestProductionkW, [18](#)
 - SOx_emissions_intensity_kgL, [23](#)
 - SOx_emissions_vec_kg, [23](#)
 - total_emissions, [23](#)
 - total_fuel_consumed_L, [23](#)

- type, 24
- writeResults, 19
- Combustion.h
 - CombustionType, 282
 - DIESEL, 282
 - FUEL_MODE_LINEAR, 284
 - FUEL_MODE_LOOKUP, 284
 - FuelMode, 282
 - N_COMBUSTION_TYPES, 282
 - N_FUEL_MODES, 284
- combustion_inputs
 - DieselInputs, 62
- combustion_map
 - Controller, 44
- combustion_ptr_vec
 - Model, 151
- CombustionInputs, 24
 - cycle_charging_setpoint, 25
 - fuel_mode, 25
 - nominal_fuel_escalation_annual, 25
 - path_2_fuel_interp_data, 25
 - production_inputs, 25
- CombustionType
 - Combustion.h, 282
- commit
 - Combustion, 15
 - Diesel, 57
 - Hydro, 85
 - Noncombustion, 158, 159
 - Production, 170
 - Renewable, 185
 - Solar, 210
 - Tidal, 239
 - Wave, 255
 - Wind, 271
- commitCharge
 - Lilon, 122
 - Storage, 220
- commitDischarge
 - Lilon, 122
 - Storage, 220
- computeEconomics
 - Combustion, 16
 - Noncombustion, 159
 - Production, 171
 - Renewable, 186
 - Storage, 221
- computeFuelAndEmissions
 - Combustion, 16
- computeProductionkW
 - Renewable, 186, 187
 - Solar, 211
 - Tidal, 240
 - Wave, 256
 - Wind, 272
- computeRealDiscountAnnual
 - Production, 172
- control_mode
 - Controller, 44
 - ModelInputs, 153
- control_string
 - Controller, 45
- Controller, 26
 - __applyCycleChargingControl_CHARGING, 28
 - __applyCycleChargingControl_DISCHARGING, 28
 - __applyLoadFollowingControl_CHARGING, 30
 - __applyLoadFollowingControl_DISCHARGING, 31
 - __computeNetLoad, 32
 - __constructCombustionMap, 33
 - __getRenewableProduction, 34
 - __handleCombustionDispatch, 36
 - __handleNoncombustionDispatch, 37
 - __handleStorageCharging, 38, 39
 - __handleStorageDischarging, 41
 - ~Controller, 27
 - applyDispatchControl, 41
 - clear, 43
 - combustion_map, 44
 - control_mode, 44
 - control_string, 45
 - Controller, 27
 - init, 43
 - missed_load_vec_kW, 45
 - net_load_vec_kW, 45
 - setControlMode, 44
- controller
 - Model, 151
- Controller.h
 - ControlMode, 278
 - CYCLE_CHARGING, 278
 - LOAD_FOLLOWING, 278
 - N_CONTROL_MODES, 278
- ControlMode
 - Controller.h, 278
- curtailment_vec_kW
 - Production, 174
- CYCLE_CHARGING
 - Controller.h, 278
- cycle_charging_setpoint
 - Combustion, 21
 - CombustionInputs, 25
- def
 - PYBIND11_Combustion.cpp, 307
 - PYBIND11_Controller.cpp, 331
 - PYBIND11_Diesel.cpp, 309
 - PYBIND11_Hydro.cpp, 312
 - PYBIND11_Interpolator.cpp, 334
 - PYBIND11_Noncombustion.cpp, 315
 - PYBIND11_Production.cpp, 317
 - PYBIND11_Renewable.cpp, 322
 - PYBIND11_Solar.cpp, 323
- def_readwrite
 - PYBIND11_Combustion.cpp, 307, 308
 - PYBIND11_Controller.cpp, 331
 - PYBIND11_Diesel.cpp, 309–311
 - PYBIND11_ElectricalLoad.cpp, 333

- PYBIND11_Hydro.cpp, 312–314
- PYBIND11_Interpolator.cpp, 335, 336
- PYBIND11_Lilon.cpp, 339–341
- PYBIND11_Model.cpp, 337
- PYBIND11_Production.cpp, 318–321
- PYBIND11_Resources.cpp, 338
- PYBIND11_Solar.cpp, 323, 324
- PYBIND11_Storage.cpp, 342, 343
- PYBIND11_Tidal.cpp, 325, 326
- PYBIND11_Wave.cpp, 327, 328
- PYBIND11_Wind.cpp, 329, 330
- degradation_a_cal
 - Lilon, 125
 - LilonInputs, 131
- degradation_alpha
 - Lilon, 126
 - LilonInputs, 131
- degradation_B_hat_cal_0
 - Lilon, 126
 - LilonInputs, 131
- degradation_beta
 - Lilon, 126
 - LilonInputs, 131
- degradation_Ea_cal_0
 - Lilon, 126
 - LilonInputs, 132
- degradation_r_cal
 - Lilon, 126
 - LilonInputs, 132
- degradation_s_cal
 - Lilon, 126
 - LilonInputs, 132
- derating
 - Solar, 212
 - SolarInputs, 214
- design_energy_period_s
 - Wave, 257
 - WaveInputs, 260
- design_significant_wave_height_m
 - Wave, 258
 - WaveInputs, 260
- design_speed_ms
 - Tidal, 242
 - TidalInputs, 244
 - Wind, 274
 - WindInputs, 276
- DIESEL
 - Combustion.h, 282
- Diesel, 46
 - __checkInputs, 49
 - __getGenericCapitalCost, 51
 - __getGenericFuelIntercept, 51
 - __getGenericFuelSlope, 52
 - __getGenericOpMaintCost, 52
 - __handleStartStop, 53
 - __writeSummary, 54
 - __writeTimeSeries, 56
 - ~Diesel, 49
 - commit, 57
 - Diesel, 48
 - handleReplacement, 58
 - minimum_load_ratio, 59
 - minimum_runtime_hrs, 59
 - requestProductionkW, 58
 - time_since_last_start_hrs, 59
- DieselInputs, 60
 - capital_cost, 61
 - CH4_emissions_intensity_kgL, 61
 - CO2_emissions_intensity_kgL, 61
 - CO_emissions_intensity_kgL, 62
 - combustion_inputs, 62
 - fuel_cost_L, 62
 - linear_fuel_intercept_LkWh, 62
 - linear_fuel_slope_LkWh, 62
 - minimum_load_ratio, 62
 - minimum_runtime_hrs, 63
 - NOx_emissions_intensity_kgL, 63
 - operation_maintenance_cost_kWh, 63
 - PM_emissions_intensity_kgL, 63
 - replace_running_hrs, 63
 - SOx_emissions_intensity_kgL, 63
- discharging_efficiency
 - Lilon, 127
 - LilonInputs, 132
- discharging_power_vec_kW
 - Storage, 224
- dispatch_vec_kW
 - Production, 174
- dt_vec_hrs
 - ElectricalLoad, 67
- dynamic_energy_capacity_kWh
 - Lilon, 127
- dynamic_power_capacity_kW
 - Lilon, 127
- electrical_load
 - Model, 151
- ElectricalLoad, 64
 - ~ElectricalLoad, 65
 - clear, 65
 - dt_vec_hrs, 67
 - ElectricalLoad, 65
 - load_vec_kW, 67
 - max_load_kW, 67
 - mean_load_kW, 67
 - min_load_kW, 68
 - n_points, 68
 - n_years, 68
 - path_2_electrical_load_time_series, 68
 - readLoadData, 66
 - time_vec_hrs, 68
- Emissions, 69
 - CH4_kg, 69
 - CO2_kg, 69
 - CO_kg, 69
 - NOx_kg, 70
 - PM_kg, 70

- SOx_kg, 70
- energy_capacity_kWh
 - Storage, 224
 - StorageInputs, 229
- example.cpp
 - main, 300
- expectedErrorNotDetected
 - testing_utils.cpp, 461
 - testing_utils.h, 468
- FLOAT_TOLERANCE
 - testing_utils.h, 468
- FLOW_TO_POWER_INTERP_KEY
 - Hydro.h, 286
- fluid_density_kgm3
 - Hydro, 87
 - HydroInputs, 91
- fuel_consumption_vec_L
 - Combustion, 21
- fuel_cost_L
 - Combustion, 21
 - DieselInputs, 62
- fuel_cost_vec
 - Combustion, 21
- fuel_mode
 - Combustion, 21
 - CombustionInputs, 25
- FUEL_MODE_LINEAR
 - Combustion.h, 284
- FUEL_MODE_LOOKUP
 - Combustion.h, 284
- fuel_mode_str
 - Combustion, 21
- FuelMode
 - Combustion.h, 282
- gas_constant_JmolK
 - Lilon, 127
 - LilonInputs, 132
- GENERATOR_EFFICIENCY_INTERP_KEY
 - Hydro.h, 286
- getAcceptablekW
 - Lilon, 123
 - Storage, 222
- getAvailablekW
 - Lilon, 124
 - Storage, 222
- getEmissionskg
 - Combustion, 17
- getFuelConsumptionL
 - Combustion, 17
- getProductionkW
 - Production, 173
- handleReplacement
 - Combustion, 18
 - Diesel, 58
 - Hydro, 86
 - Lilon, 125
- Noncombustion, 160
- Production, 173
- Renewable, 187
- Solar, 212
- Storage, 222
- Tidal, 241
- Wave, 257
- Wind, 273
- header/Controller.h, 277
- header/doxygen_cite.h, 278
- header/ElectricalLoad.h, 279
- header/Interpolator.h, 279
- header/Model.h, 280
- header/Production/Combustion/Combustion.h, 281
- header/Production/Combustion/Diesel.h, 284
- header/Production/Noncombustion/Hydro.h, 285
- header/Production/Noncombustion/Noncombustion.h, 287
- header/Production/Production.h, 288
- header/Production/Renewable/Renewable.h, 289
- header/Production/Renewable/Solar.h, 290
- header/Production/Renewable/Tidal.h, 291
- header/Production/Renewable/Wave.h, 293
- header/Production/Renewable/Wind.h, 294
- header/Resources.h, 296
- header/std_includes.h, 297
- header/Storage/Lilon.h, 297
- header/Storage/Storage.h, 298
- HYDRO
 - Noncombustion.h, 288
- Hydro, 71
 - __checkInputs, 75
 - __flowToPower, 76
 - __getAcceptableFlow, 76
 - __getAvailableFlow, 77
 - __getEfficiencyFactor, 77
 - __getGenericCapitalCost, 78
 - __getGenericOpMaintCost, 78
 - __getMaximumFlowm3hr, 79
 - __getMinimumFlowm3hr, 79
 - __initInterpolator, 79
 - __powerToFlow, 81
 - __updateState, 82
 - __writeSummary, 83
 - __writeTimeSeries, 84
 - ~Hydro, 75
 - commit, 85
 - fluid_density_kgm3, 87
 - handleReplacement, 86
 - Hydro, 73
 - init_reservoir_state, 88
 - maximum_flow_m3hr, 88
 - minimum_flow_m3hr, 88
 - minimum_power_kW, 88
 - net_head_m, 88
 - requestProductionkW, 86
 - reservoir_capacity_m3, 88
 - spill_rate_vec_m3hr, 89

- stored_volume_m3, [89](#)
- stored_volume_vec_m3, [89](#)
- turbine_flow_vec_m3hr, [89](#)
- turbine_type, [89](#)
- Hydro.h
 - FLOW_TO_POWER_INTERP_KEY, [286](#)
 - GENERATOR_EFFICIENCY_INTERP_KEY, [286](#)
 - HYDRO_TURBINE_FRANCIS, [286](#)
 - HYDRO_TURBINE_KAPLAN, [286](#)
 - HYDRO_TURBINE_PELTON, [286](#)
 - HydroInterpKeys, [286](#)
 - HydroTurbineType, [286](#)
 - N_HYDRO_INTERP_KEYS, [286](#)
 - N_HYDRO_TURBINES, [286](#)
 - TURBINE_EFFICIENCY_INTERP_KEY, [286](#)
- HYDRO_TURBINE_FRANCIS
 - Hydro.h, [286](#)
- HYDRO_TURBINE_KAPLAN
 - Hydro.h, [286](#)
- HYDRO_TURBINE_PELTON
 - Hydro.h, [286](#)
- HydroInputs, [90](#)
 - capital_cost, [91](#)
 - fluid_density_kgm3, [91](#)
 - init_reservoir_state, [91](#)
 - net_head_m, [91](#)
 - noncombustion_inputs, [91](#)
 - operation_maintenance_cost_kWh, [91](#)
 - reservoir_capacity_m3, [92](#)
 - resource_key, [92](#)
 - turbine_type, [92](#)
- HydroInterpKeys
 - Hydro.h, [286](#)
- HydroTurbineType
 - Hydro.h, [286](#)
- hysteresis_SOC
 - Lilon, [127](#)
 - LilonInputs, [132](#)
- init
 - Controller, [43](#)
- init_reservoir_state
 - Hydro, [88](#)
 - HydroInputs, [91](#)
- init_SOC
 - Lilon, [127](#)
 - LilonInputs, [133](#)
- interp1D
 - Interpolator, [103](#)
- interp2D
 - Interpolator, [104](#)
- interp_map_1D
 - Interpolator, [105](#)
- interp_map_2D
 - Interpolator, [105](#)
- Interpolator, [92](#)
 - __checkBounds1D, [94](#)
 - __checkBounds2D, [95](#)
 - __checkDataKey1D, [96](#)
 - __checkDataKey2D, [96](#)
 - __getDataStringMatrix, [97](#)
 - __getInterpolationIndex, [97](#)
 - __isNonNumeric, [98](#)
 - __readData1D, [98](#)
 - __readData2D, [99](#)
 - __splitCommaSeparatedString, [101](#)
 - __throwReadError, [102](#)
 - ~Interpolator, [94](#)
 - addData1D, [102](#)
 - addData2D, [103](#)
 - interp1D, [103](#)
 - interp2D, [104](#)
 - interp_map_1D, [105](#)
 - interp_map_2D, [105](#)
 - Interpolator, [94](#)
 - path_map_1D, [105](#)
 - path_map_2D, [105](#)
- interpolator
 - Production, [175](#)
 - Storage, [225](#)
- InterpolatorStruct1D, [106](#)
 - max_x, [106](#)
 - min_x, [106](#)
 - n_points, [106](#)
 - x_vec, [107](#)
 - y_vec, [107](#)
- InterpolatorStruct2D, [107](#)
 - max_x, [108](#)
 - max_y, [108](#)
 - min_x, [108](#)
 - min_y, [108](#)
 - n_cols, [108](#)
 - n_rows, [108](#)
 - x_vec, [109](#)
 - y_vec, [109](#)
 - z_matrix, [109](#)
- is_depleted
 - Storage, [225](#)
- is_running
 - Production, [175](#)
- is_running_vec
 - Production, [175](#)
- is_sunk
 - Production, [175](#)
 - ProductionInputs, [180](#)
 - Storage, [225](#)
 - StorageInputs, [229](#)
- levellized_cost_of_energy_kWh
 - Model, [151](#)
 - Production, [175](#)
 - Storage, [225](#)
- LIION
 - Storage.h, [299](#)
- Lilon, [110](#)
 - __checkInputs, [114](#)
 - __getBcal, [116](#)
 - __getEcal, [117](#)

- __getGenericCapitalCost, 117
- __getGenericOpMaintCost, 118
- __handleDegradation, 118
- __modelDegradation, 119
- __toggleDepleted, 119
- __writeSummary, 120
- __writeTimeSeries, 121
- ~Lilon, 114
- charging_efficiency, 125
- commitCharge, 122
- commitDischarge, 122
- degradation_a_cal, 125
- degradation_alpha, 126
- degradation_B_hat_cal_0, 126
- degradation_beta, 126
- degradation_Ea_cal_0, 126
- degradation_r_cal, 126
- degradation_s_cal, 126
- discharging_efficiency, 127
- dynamic_energy_capacity_kWh, 127
- dynamic_power_capacity_kW, 127
- gas_constant_JmolK, 127
- getAcceptablekW, 123
- getAvailablekW, 124
- handleReplacement, 125
- hysteresis_SOC, 127
- init_SOC, 127
- Lilon, 112
- max_SOC, 128
- min_SOC, 128
- power_degradation_flag, 128
- replace_SOH, 128
- SOH, 128
- SOH_vec, 128
- temperature_K, 129
- LilonInputs, 129
 - capital_cost, 131
 - charging_efficiency, 131
 - degradation_a_cal, 131
 - degradation_alpha, 131
 - degradation_B_hat_cal_0, 131
 - degradation_beta, 131
 - degradation_Ea_cal_0, 132
 - degradation_r_cal, 132
 - degradation_s_cal, 132
 - discharging_efficiency, 132
 - gas_constant_JmolK, 132
 - hysteresis_SOC, 132
 - init_SOC, 133
 - max_SOC, 133
 - min_SOC, 133
 - operation_maintenance_cost_kWh, 133
 - power_degradation_flag, 133
 - replace_SOH, 133
 - storage_inputs, 134
 - temperature_K, 134
- linear_fuel_intercept_LkWh
 - Combustion, 22
 - DieselInputs, 62
- linear_fuel_slope_LkWh
 - Combustion, 22
 - DieselInputs, 62
- LOAD_FOLLOWING
 - Controller.h, 278
- load_vec_kW
 - ElectricalLoad, 67
- main
 - example.cpp, 300
 - test_Combustion.cpp, 354
 - test_Controller.cpp, 413
 - test_Diesel.cpp, 357
 - test_ElectricalLoad.cpp, 415
 - test_Hydro.cpp, 367
 - test_Interpolator.cpp, 418
 - test_Lilon.cpp, 405
 - test_Model.cpp, 429
 - test_Noncombustion.cpp, 372
 - test_Production.cpp, 402
 - test_Renewable.cpp, 375
 - test_Resources.cpp, 450
 - test_Solar.cpp, 377
 - test_Storage.cpp, 410
 - test_Tidal.cpp, 384
 - test_Wave.cpp, 389
 - test_Wind.cpp, 396
- max_load_kW
 - ElectricalLoad, 67
- max_SOC
 - Lilon, 128
 - LilonInputs, 133
- max_x
 - InterpolatorStruct1D, 106
 - InterpolatorStruct2D, 108
- max_y
 - InterpolatorStruct2D, 108
- maximum_flow_m3hr
 - Hydro, 88
- mean_load_kW
 - ElectricalLoad, 67
- min_load_kW
 - ElectricalLoad, 68
- min_SOC
 - Lilon, 128
 - LilonInputs, 133
- min_x
 - InterpolatorStruct1D, 106
 - InterpolatorStruct2D, 108
- min_y
 - InterpolatorStruct2D, 108
- minimum_flow_m3hr
 - Hydro, 88
- minimum_load_ratio
 - Diesel, 59
 - DieselInputs, 62
- minimum_power_kW
 - Hydro, 88

- minimum_runtime_hrs
 - Diesel, [59](#)
 - DieselInputs, [63](#)
- missed_load_vec_kW
 - Controller, [45](#)
- Model, [134](#)
 - __checkInputs, [137](#)
 - __computeEconomics, [138](#)
 - __computeFuelAndEmissions, [138](#)
 - __computeLevellizedCostOfEnergy, [139](#)
 - __computeNetPresentCost, [139](#)
 - __writeSummary, [140](#)
 - __writeTimeSeries, [143](#)
 - ~Model, [137](#)
 - addDiesel, [144](#)
 - addHydro, [144](#)
 - addLilon, [145](#)
 - addResource, [145](#), [146](#)
 - addSolar, [146](#)
 - addTidal, [147](#)
 - addWave, [147](#)
 - addWind, [147](#)
 - clear, [148](#)
 - combustion_ptr_vec, [151](#)
 - controller, [151](#)
 - electrical_load, [151](#)
 - levellized_cost_of_energy_kWh, [151](#)
 - Model, [136](#), [137](#)
 - net_present_cost, [151](#)
 - noncombustion_ptr_vec, [151](#)
 - renewable_ptr_vec, [152](#)
 - reset, [148](#)
 - resources, [152](#)
 - run, [149](#)
 - storage_ptr_vec, [152](#)
 - total_dispatch_discharge_kWh, [152](#)
 - total_emissions, [152](#)
 - total_fuel_consumed_L, [152](#)
 - total_renewable_dispatch_kWh, [153](#)
 - writeResults, [149](#)
- ModelInputs, [153](#)
 - control_mode, [153](#)
 - path_2_electrical_load_time_series, [154](#)
- n_cols
 - InterpolatorStruct2D, [108](#)
- N_COMBUSTION_TYPES
 - Combustion.h, [282](#)
- N_CONTROL_MODES
 - Controller.h, [278](#)
- N_FUEL_MODES
 - Combustion.h, [284](#)
- N_HYDRO_INTERP_KEYS
 - Hydro.h, [286](#)
- N_HYDRO_TURBINES
 - Hydro.h, [286](#)
- N_NONCOMBUSTION_TYPES
 - Noncombustion.h, [288](#)
- n_points
 - ElectricalLoad, [68](#)
 - InterpolatorStruct1D, [106](#)
 - Production, [175](#)
 - Storage, [225](#)
- N_RENEWABLE_TYPES
 - Renewable.h, [290](#)
- n_replacements
 - Production, [176](#)
 - Storage, [225](#)
- n_rows
 - InterpolatorStruct2D, [108](#)
- n_starts
 - Production, [176](#)
- N_STORAGE_TYPES
 - Storage.h, [299](#)
- N_TIDAL_POWER_PRODUCTION_MODELS
 - Tidal.h, [293](#)
- N_WAVE_POWER_PRODUCTION_MODELS
 - Wave.h, [294](#)
- N_WIND_POWER_PRODUCTION_MODELS
 - Wind.h, [295](#)
- n_years
 - ElectricalLoad, [68](#)
 - Production, [176](#)
 - Storage, [226](#)
- net_head_m
 - Hydro, [88](#)
 - HydroInputs, [91](#)
- net_load_vec_kW
 - Controller, [45](#)
- net_present_cost
 - Model, [151](#)
 - Production, [176](#)
 - Storage, [226](#)
- nominal_discount_annual
 - Production, [176](#)
 - ProductionInputs, [180](#)
 - Storage, [226](#)
 - StorageInputs, [229](#)
- nominal_fuel_escalation_annual
 - Combustion, [22](#)
 - CombustionInputs, [25](#)
- nominal_inflation_annual
 - Production, [176](#)
 - ProductionInputs, [180](#)
 - Storage, [226](#)
 - StorageInputs, [229](#)
- Noncombustion, [154](#)
 - __checkInputs, [157](#)
 - __handleStartStop, [157](#)
 - __writeSummary, [158](#)
 - __writeTimeSeries, [158](#)
 - ~Noncombustion, [157](#)
 - commit, [158](#), [159](#)
 - computeEconomics, [159](#)
 - handleReplacement, [160](#)
 - Noncombustion, [156](#)
 - requestProductionkW, [160](#)

- resource_key, 161
- type, 161
- writeResults, 160
- Noncombustion.h
 - HYDRO, 288
 - N_NONCOMBUSTION_TYPES, 288
 - NoncombustionType, 288
- noncombustion_inputs
 - HydroInputs, 91
- noncombustion_ptr_vec
 - Model, 151
- NoncombustionInputs, 162
 - production_inputs, 162
- NoncombustionType
 - Noncombustion.h, 288
- normalized_production_series_given
 - Production, 177
- normalized_production_vec
 - Production, 177
- NOx_emissions_intensity_kgL
 - Combustion, 22
 - DieselInputs, 63
- NOx_emissions_vec_kg
 - Combustion, 22
- NOx_kg
 - Emissions, 70
- operation_maintenance_cost_kWh
 - DieselInputs, 63
 - HydroInputs, 91
 - LilongInputs, 133
 - Production, 177
 - SolarInputs, 214
 - Storage, 226
 - TidalInputs, 244
 - WaveInputs, 260
 - WindInputs, 276
- operation_maintenance_cost_vec
 - Production, 177
 - Storage, 226
- path_2_electrical_load_time_series
 - ElectricalLoad, 68
 - ModelInputs, 154
- path_2_fuel_interp_data
 - CombustionInputs, 25
- path_2_normalized_performance_matrix
 - WaveInputs, 260
- path_2_normalized_production_time_series
 - Production, 177
 - ProductionInputs, 180
- path_map_1D
 - Interpolator, 105
 - Resources, 202
- path_map_2D
 - Interpolator, 105
 - Resources, 202
- PM_emissions_intensity_kgL
 - Combustion, 22
- DieselInputs, 63
- PM_emissions_vec_kg
 - Combustion, 23
- PM_kg
 - Emissions, 70
- power_capacity_kW
 - Storage, 227
 - StorageInputs, 229
- power_degradation_flag
 - Lilong, 128
 - LilongInputs, 133
- power_kW
 - Storage, 227
- power_model
 - Tidal, 242
 - TidalInputs, 244
 - Wave, 258
 - WaveInputs, 261
 - Wind, 274
 - WindInputs, 276
- power_model_string
 - Tidal, 242
 - Wave, 258
 - Wind, 274
- print_flag
 - Production, 177
 - ProductionInputs, 180
 - Storage, 227
 - StorageInputs, 229
- printGold
 - testing_utils.cpp, 462
 - testing_utils.h, 468
- printGreen
 - testing_utils.cpp, 462
 - testing_utils.h, 469
- printRed
 - testing_utils.cpp, 462
 - testing_utils.h, 469
- Production, 163
 - __checkInputs, 167
 - __checkNormalizedProduction, 168
 - __checkTimePoint, 169
 - __readNormalizedProductionData, 169
 - __throwLengthError, 170
 - ~Production, 167
 - capacity_kW, 174
 - capital_cost, 174
 - capital_cost_vec, 174
 - commit, 170
 - computeEconomics, 171
 - computeRealDiscountAnnual, 172
 - curtailment_vec_kW, 174
 - dispatch_vec_kW, 174
 - getProductionkW, 173
 - handleReplacement, 173
 - interpolator, 175
 - is_running, 175
 - is_running_vec, 175

- is_sunk, 175
- levellized_cost_of_energy_kWh, 175
- n_points, 175
- n_replacements, 176
- n_starts, 176
- n_years, 176
- net_present_cost, 176
- nominal_discount_annual, 176
- nominal_inflation_annual, 176
- normalized_production_series_given, 177
- normalized_production_vec, 177
- operation_maintenance_cost_kWh, 177
- operation_maintenance_cost_vec, 177
- path_2_normalized_production_time_series, 177
- print_flag, 177
- Production, 166
- production_vec_kW, 178
- real_discount_annual, 178
- replace_running_hrs, 178
- running_hours, 178
- storage_vec_kW, 178
- total_dispatch_kWh, 178
- type_str, 179
- production_inputs
 - CombustionInputs, 25
 - NoncombustionInputs, 162
 - RenewableInputs, 189
- production_vec_kW
 - Production, 178
- ProductionInputs, 179
 - capacity_kW, 180
 - is_sunk, 180
 - nominal_discount_annual, 180
 - nominal_inflation_annual, 180
 - path_2_normalized_production_time_series, 180
 - print_flag, 180
 - replace_running_hrs, 181
- projects/example.cpp, 300
- PYBIND11_Combustion.cpp
 - def, 307
 - def_readwrite, 307, 308
 - value, 307, 308
- PYBIND11_Controller.cpp
 - def, 331
 - def_readwrite, 331
 - value, 332
- PYBIND11_Diesel.cpp
 - def, 309
 - def_readwrite, 309–311
- PYBIND11_ElectricalLoad.cpp
 - def_readwrite, 333
- PYBIND11_Hydro.cpp
 - def, 312
 - def_readwrite, 312–314
 - value, 314
- PYBIND11_Interpolator.cpp
 - def, 334
 - def_readwrite, 335, 336
- PYBIND11_Lilon.cpp
 - def_readwrite, 339–341
- PYBIND11_Model.cpp
 - def_readwrite, 337
- PYBIND11_MODULE
 - PYBIND11_PGM.cpp, 305
- PYBIND11_Noncombustion.cpp
 - def, 315
 - value, 315
- PYBIND11_PGM.cpp
 - PYBIND11_MODULE, 305
- PYBIND11_Production.cpp
 - def, 317
 - def_readwrite, 318–321
- PYBIND11_Renewable.cpp
 - def, 322
 - value, 322
- PYBIND11_Resources.cpp
 - def_readwrite, 338
- PYBIND11_Solar.cpp
 - def, 323
 - def_readwrite, 323, 324
- PYBIND11_Storage.cpp
 - def_readwrite, 342, 343
 - value, 343
- PYBIND11_Tidal.cpp
 - def_readwrite, 325, 326
 - value, 325
- PYBIND11_Wave.cpp
 - def_readwrite, 327, 328
 - value, 327, 328
- PYBIND11_Wind.cpp
 - def_readwrite, 329, 330
 - value, 329
- pybindings/PYBIND11_PGM.cpp, 304
- pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp, 306
- pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp, 308
- pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp, 311
- pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp, 315
- pybindings/snippets/Production/PYBIND11_Production.cpp, 316
- pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp, 321
- pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp, 323
- pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp, 324
- pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp, 326
- pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp, 328
- pybindings/snippets/PYBIND11_Controller.cpp, 330
- pybindings/snippets/PYBIND11_ElectricalLoad.cpp, 332

- pybindings/snippets/PYBIND11_Interpolator.cpp, 334
- pybindings/snippets/PYBIND11_Model.cpp, 336
- pybindings/snippets/PYBIND11_Resources.cpp, 337
- pybindings/snippets/Storage/PYBIND11_Lilon.cpp, 338
- pybindings/snippets/Storage/PYBIND11_Storage.cpp, 342
- readLoadData
 - ElectricalLoad, 66
- real_discount_annual
 - Production, 178
 - Storage, 227
- real_fuel_escalation_annual
 - Combustion, 23
- Renewable, 181
 - __checkInputs, 184
 - __handleStartStop, 184
 - __writeSummary, 185
 - __writeTimeSeries, 185
 - ~Renewable, 184
 - commit, 185
 - computeEconomics, 186
 - computeProductionkW, 186, 187
 - handleReplacement, 187
 - Renewable, 183
 - resource_key, 188
 - type, 188
 - writeResults, 187
- Renewable.h
 - N_RENEWABLE_TYPES, 290
 - RenewableType, 290
 - SOLAR, 290
 - TIDAL, 290
 - WAVE, 290
 - WIND, 290
- renewable_inputs
 - SolarInputs, 214
 - TidalInputs, 244
 - WaveInputs, 261
 - WindInputs, 276
- renewable_ptr_vec
 - Model, 152
- RenewableInputs, 189
 - production_inputs, 189
- RenewableType
 - Renewable.h, 290
- replace_running_hrs
 - DieselInputs, 63
 - Production, 178
 - ProductionInputs, 181
- replace_SOH
 - Lilon, 128
 - LilonInputs, 133
- requestProductionkW
 - Combustion, 18
 - Diesel, 58
 - Hydro, 86
 - Noncombustion, 160
- reservoir_capacity_m3
 - Hydro, 88
 - HydroInputs, 92
- reset
 - Model, 148
- resource_key
 - HydroInputs, 92
 - Noncombustion, 161
 - Renewable, 188
 - SolarInputs, 214
 - TidalInputs, 244
 - WaveInputs, 261
 - WindInputs, 276
- resource_map_1D
 - Resources, 202
- resource_map_2D
 - Resources, 203
- Resources, 190
 - __checkResourceKey1D, 192
 - __checkResourceKey2D, 193
 - __checkTimePoint, 194
 - __readHydroResource, 194
 - __readSolarResource, 195
 - __readTidalResource, 196
 - __readWaveResource, 197
 - __readWindResource, 198
 - __throwLengthError, 199
 - ~Resources, 191
 - addResource, 200, 201
 - clear, 202
 - path_map_1D, 202
 - path_map_2D, 202
 - resource_map_1D, 202
 - resource_map_2D, 203
 - Resources, 191
 - string_map_1D, 203
 - string_map_2D, 203
- resources
 - Model, 152
- run
 - Model, 149
- running_hours
 - Production, 178
- setControlMode
 - Controller, 44
- SOH
 - Lilon, 128
- SOH_vec
 - Lilon, 128
- SOLAR
 - Renewable.h, 290
- Solar, 204
 - __checkInputs, 207
 - __getGenericCapitalCost, 207
 - __getGenericOpMaintCost, 208
 - __writeSummary, 208
 - __writeTimeSeries, 209
 - ~Solar, 207
 - commit, 210

- computeProductionkW, 211
- derating, 212
- handleReplacement, 212
- Solar, 205, 206
- SolarInputs, 213
 - capital_cost, 214
 - derating, 214
 - operation_maintenance_cost_kWh, 214
 - renewable_inputs, 214
 - resource_key, 214
- source/Controller.cpp, 344
- source/ElectricalLoad.cpp, 344
- source/Interpolator.cpp, 345
- source/Model.cpp, 345
- source/Production/Combustion/Combustion.cpp, 346
- source/Production/Combustion/Diesel.cpp, 346
- source/Production/Noncombustion/Hydro.cpp, 347
- source/Production/Noncombustion/Noncombustion.cpp, 347
- source/Production/Production.cpp, 348
- source/Production/Renewable/Renewable.cpp, 349
- source/Production/Renewable/Solar.cpp, 349
- source/Production/Renewable/Tidal.cpp, 350
- source/Production/Renewable/Wave.cpp, 350
- source/Production/Renewable/Wind.cpp, 351
- source/Resources.cpp, 352
- source/Storage/Lilon.cpp, 352
- source/Storage/Storage.cpp, 353
- SOx_emissions_intensity_kgL
 - Combustion, 23
 - DieselInputs, 63
- SOx_emissions_vec_kg
 - Combustion, 23
- SOx_kg
 - Emissions, 70
- spill_rate_vec_m3hr
 - Hydro, 89
- Storage, 215
 - __checkInputs, 218
 - __computeRealDiscountAnnual, 219
 - __writeSummary, 220
 - __writeTimeSeries, 220
 - ~Storage, 218
 - capital_cost, 223
 - capital_cost_vec, 224
 - charge_kWh, 224
 - charge_vec_kWh, 224
 - charging_power_vec_kW, 224
 - commitCharge, 220
 - commitDischarge, 220
 - computeEconomics, 221
 - discharging_power_vec_kW, 224
 - energy_capacity_kWh, 224
 - getAcceptablekW, 222
 - getAvailablekW, 222
 - handleReplacement, 222
 - interpolator, 225
 - is_depleted, 225
 - is_sunk, 225
 - levelized_cost_of_energy_kWh, 225
 - n_points, 225
 - n_replacements, 225
 - n_years, 226
 - net_present_cost, 226
 - nominal_discount_annual, 226
 - nominal_inflation_annual, 226
 - operation_maintenance_cost_kWh, 226
 - operation_maintenance_cost_vec, 226
 - power_capacity_kW, 227
 - power_kW, 227
 - print_flag, 227
 - real_discount_annual, 227
 - Storage, 217
 - total_discharge_kWh, 227
 - type, 227
 - type_str, 228
 - writeResults, 222
- Storage.h
 - LIION, 299
 - N_STORAGE_TYPES, 299
 - StorageType, 299
- storage_inputs
 - LilonInputs, 134
- storage_ptr_vec
 - Model, 152
- storage_vec_kW
 - Production, 178
- StorageInputs, 228
 - energy_capacity_kWh, 229
 - is_sunk, 229
 - nominal_discount_annual, 229
 - nominal_inflation_annual, 229
 - power_capacity_kW, 229
 - print_flag, 229
- StorageType
 - Storage.h, 299
- stored_volume_m3
 - Hydro, 89
- stored_volume_vec_m3
 - Hydro, 89
- string_map_1D
 - Resources, 203
- string_map_2D
 - Resources, 203
- temperature_K
 - Lilon, 129
 - LilonInputs, 134
- test/source/Production/Combustion/test_Combustion.cpp, 353
- test/source/Production/Combustion/test_Diesel.cpp, 356
- test/source/Production/Noncombustion/test_Hydro.cpp, 366
- test/source/Production/Noncombustion/test_Noncombustion.cpp, 372

test/source/Production/Renewable/test_Renewable.cpp, 374
 test/source/Production/Renewable/test_Solar.cpp, 376
 test/source/Production/Renewable/test_Tidal.cpp, 383
 test/source/Production/Renewable/test_Wave.cpp, 388
 test/source/Production/Renewable/test_Wind.cpp, 395
 test/source/Production/test_Production.cpp, 401
 test/source/Storage/test_Lilon.cpp, 404
 test/source/Storage/test_Storage.cpp, 409
 test/source/test_Controller.cpp, 412
 test/source/test_ElectricalLoad.cpp, 414
 test/source/test_Interpolator.cpp, 418
 test/source/test_Model.cpp, 428
 test/source/test_Resources.cpp, 448
 test/utls/testing_utils.cpp, 460
 test/utls/testing_utils.h, 466
 test_Combustion.cpp
 main, 354
 testConstruct_Combustion, 354
 test_Controller.cpp
 main, 413
 testConstruct_Controller, 413
 test_Diesel.cpp
 main, 357
 testBadConstruct_Diesel, 357
 testCapacityConstraint_Diesel, 358
 testCommit_Diesel, 358
 testConstruct_Diesel, 360
 testConstructLookup_Diesel, 361
 testEconomics_Diesel, 361
 testFuelConsumptionEmissions_Diesel, 362
 testFuelLookup_Diesel, 364
 testMinimumLoadRatioConstraint_Diesel, 365
 testMinimumRuntimeConstraint_Diesel, 365
 test_ElectricalLoad.cpp
 main, 415
 testConstruct_ElectricalLoad, 415
 testDataRead_ElectricalLoad, 415
 testPostConstructionAttributes_ElectricalLoad, 417
 test_Hydro.cpp
 main, 367
 testCommit_Hydro, 368
 testConstruct_Hydro, 369
 testEfficiencyInterpolation_Hydro, 370
 test_Interpolator.cpp
 main, 418
 testBadIndexing1D_Interpolator, 419
 testConstruct_Interpolator, 420
 testDataRead1D_Interpolator, 420
 testDataRead2D_Interpolator, 421
 testInterpolation1D_Interpolator, 424
 testInterpolation2D_Interpolator, 425
 testInvalidInterpolation1D_Interpolator, 426
 testInvalidInterpolation2D_Interpolator, 427
 test_Lilon.cpp
 main, 405
 testBadConstruct_Lilon, 406
 testCommitCharge_Lilon, 406
 testCommitDischarge_Lilon, 407
 testConstruct_Lilon, 408
 test_Model.cpp
 main, 429
 testAddDiesel_Model, 431
 testAddHydro_Model, 432
 testAddHydroResource_Model, 433
 testAddLilon_Model, 434
 testAddSolar_Model, 434
 testAddSolar_productionOverride_Model, 435
 testAddSolarResource_Model, 436
 testAddTidal_Model, 437
 testAddTidalResource_Model, 438
 testAddWave_Model, 439
 testAddWaveResource_Model, 439
 testAddWind_Model, 441
 testAddWindResource_Model, 442
 testBadConstruct_Model, 443
 testConstruct_Model, 443
 testEconomics_Model, 444
 testElectricalLoadData_Model, 444
 testFuelConsumptionEmissions_Model, 445
 testLoadBalance_Model, 446
 testPostConstructionAttributes_Model, 448
 test_Noncombustion.cpp
 main, 372
 testConstruct_Noncombustion, 373
 test_Production.cpp
 main, 402
 testBadConstruct_Production, 402
 testConstruct_Production, 403
 test_Renewable.cpp
 main, 375
 testConstruct_Renewable, 375
 test_Resources.cpp
 main, 450
 testAddHydroResource_Resources, 451
 testAddSolarResource_Resources, 452
 testAddTidalResource_Resources, 453
 testAddWaveResource_Resources, 455
 testAddWindResource_Resources, 457
 testBadAdd_Resources, 458
 testConstruct_Resources, 460
 test_Solar.cpp
 main, 377
 testBadConstruct_Solar, 378
 testCommit_Solar, 378
 testConstruct_Solar, 380
 testEconomics_Solar, 381
 testProductionConstraint_Solar, 381
 testProductionOverride_Solar, 382
 test_Storage.cpp
 main, 410
 testBadConstruct_Storage, 411
 testConstruct_Storage, 411
 test_Tidal.cpp
 main, 384
 testBadConstruct_Tidal, 384

- testCommit_Tidal, 385
- testConstruct_Tidal, 386
- testEconomics_Tidal, 387
- testProductionConstraint_Tidal, 387
- test_Wave.cpp
 - main, 389
 - testBadConstruct_Wave, 390
 - testCommit_Wave, 390
 - testConstruct_Wave, 392
 - testConstructLookup_Wave, 393
 - testEconomics_Wave, 393
 - testProductionConstraint_Wave, 393
 - testProductionLookup_Wave, 394
- test_Wind.cpp
 - main, 396
 - testBadConstruct_Wind, 397
 - testCommit_Wind, 398
 - testConstruct_Wind, 399
 - testEconomics_Wind, 400
 - testProductionConstraint_Wind, 400
- testAddDiesel_Model
 - test_Model.cpp, 431
- testAddHydro_Model
 - test_Model.cpp, 432
- testAddHydroResource_Model
 - test_Model.cpp, 433
- testAddHydroResource_Resources
 - test_Resources.cpp, 451
- testAddLilon_Model
 - test_Model.cpp, 434
- testAddSolar_Model
 - test_Model.cpp, 434
- testAddSolar_productionOverride_Model
 - test_Model.cpp, 435
- testAddSolarResource_Model
 - test_Model.cpp, 436
- testAddSolarResource_Resources
 - test_Resources.cpp, 452
- testAddTidal_Model
 - test_Model.cpp, 437
- testAddTidalResource_Model
 - test_Model.cpp, 438
- testAddTidalResource_Resources
 - test_Resources.cpp, 453
- testAddWave_Model
 - test_Model.cpp, 439
- testAddWaveResource_Model
 - test_Model.cpp, 439
- testAddWaveResource_Resources
 - test_Resources.cpp, 455
- testAddWind_Model
 - test_Model.cpp, 441
- testAddWindResource_Model
 - test_Model.cpp, 442
- testAddWindResource_Resources
 - test_Resources.cpp, 457
- testBadAdd_Resources
 - test_Resources.cpp, 458
- testBadConstruct_Diesel
 - test_Diesel.cpp, 357
- testBadConstruct_Lilon
 - test_Lilon.cpp, 406
- testBadConstruct_Model
 - test_Model.cpp, 443
- testBadConstruct_Production
 - test_Production.cpp, 402
- testBadConstruct_Solar
 - test_Solar.cpp, 378
- testBadConstruct_Storage
 - test_Storage.cpp, 411
- testBadConstruct_Tidal
 - test_Tidal.cpp, 384
- testBadConstruct_Wave
 - test_Wave.cpp, 390
- testBadConstruct_Wind
 - test_Wind.cpp, 397
- testBadIndexing1D_Interpolator
 - test_Interpolator.cpp, 419
- testCapacityConstraint_Diesel
 - test_Diesel.cpp, 358
- testCommit_Diesel
 - test_Diesel.cpp, 358
- testCommit_Hydro
 - test_Hydro.cpp, 368
- testCommit_Solar
 - test_Solar.cpp, 378
- testCommit_Tidal
 - test_Tidal.cpp, 385
- testCommit_Wave
 - test_Wave.cpp, 390
- testCommit_Wind
 - test_Wind.cpp, 398
- testCommitCharge_Lilon
 - test_Lilon.cpp, 406
- testCommitDischarge_Lilon
 - test_Lilon.cpp, 407
- testConstruct_Combustion
 - test_Combustion.cpp, 354
- testConstruct_Controller
 - test_Controller.cpp, 413
- testConstruct_Diesel
 - test_Diesel.cpp, 360
- testConstruct_ElectricalLoad
 - test_ElectricalLoad.cpp, 415
- testConstruct_Hydro
 - test_Hydro.cpp, 369
- testConstruct_Interpolator
 - test_Interpolator.cpp, 420
- testConstruct_Lilon
 - test_Lilon.cpp, 408
- testConstruct_Model
 - test_Model.cpp, 443
- testConstruct_Noncombustion
 - test_Noncombustion.cpp, 373
- testConstruct_Production
 - test_Production.cpp, 403

- testConstruct_Renewable
 - test_Renewable.cpp, 375
- testConstruct_Resources
 - test_Resources.cpp, 460
- testConstruct_Solar
 - test_Solar.cpp, 380
- testConstruct_Storage
 - test_Storage.cpp, 411
- testConstruct_Tidal
 - test_Tidal.cpp, 386
- testConstruct_Wave
 - test_Wave.cpp, 392
- testConstruct_Wind
 - test_Wind.cpp, 399
- testConstructLookup_Diesel
 - test_Diesel.cpp, 361
- testConstructLookup_Wave
 - test_Wave.cpp, 393
- testDataRead1D_Interpolator
 - test_Interpolator.cpp, 420
- testDataRead2D_Interpolator
 - test_Interpolator.cpp, 421
- testDataRead_ElectricalLoad
 - test_ElectricalLoad.cpp, 415
- testEconomics_Diesel
 - test_Diesel.cpp, 361
- testEconomics_Model
 - test_Model.cpp, 444
- testEconomics_Solar
 - test_Solar.cpp, 381
- testEconomics_Tidal
 - test_Tidal.cpp, 387
- testEconomics_Wave
 - test_Wave.cpp, 393
- testEconomics_Wind
 - test_Wind.cpp, 400
- testEfficiencyInterpolation_Hydro
 - test_Hydro.cpp, 370
- testElectricalLoadData_Model
 - test_Model.cpp, 444
- testFloatEquals
 - testing_utils.cpp, 463
 - testing_utils.h, 469
- testFuelConsumptionEmissions_Diesel
 - test_Diesel.cpp, 362
- testFuelConsumptionEmissions_Model
 - test_Model.cpp, 445
- testFuelLookup_Diesel
 - test_Diesel.cpp, 364
- testGreaterThan
 - testing_utils.cpp, 463
 - testing_utils.h, 470
- testGreaterThanOrEqualTo
 - testing_utils.cpp, 464
 - testing_utils.h, 471
- testing_utils.cpp
 - expectedErrorNotDetected, 461
 - printGold, 462
 - printGreen, 462
 - printRed, 462
 - testFloatEquals, 463
 - testGreaterThan, 463
 - testGreaterThanOrEqualTo, 464
 - testLessThan, 465
 - testLessThanOrEqualTo, 465
 - testTruth, 466
- testing_utils.h
 - expectedErrorNotDetected, 468
 - FLOAT_TOLERANCE, 468
 - printGold, 468
 - printGreen, 469
 - printRed, 469
 - testFloatEquals, 469
 - testGreaterThan, 470
 - testGreaterThanOrEqualTo, 471
 - testLessThan, 471
 - testLessThanOrEqualTo, 472
 - testTruth, 472
- testInterpolation1D_Interpolator
 - test_Interpolator.cpp, 424
- testInterpolation2D_Interpolator
 - test_Interpolator.cpp, 425
- testInvalidInterpolation1D_Interpolator
 - test_Interpolator.cpp, 426
- testInvalidInterpolation2D_Interpolator
 - test_Interpolator.cpp, 427
- testLessThan
 - testing_utils.cpp, 465
 - testing_utils.h, 471
- testLessThanOrEqualTo
 - testing_utils.cpp, 465
 - testing_utils.h, 472
- testLoadBalance_Model
 - test_Model.cpp, 446
- testMinimumLoadRatioConstraint_Diesel
 - test_Diesel.cpp, 365
- testMinimumRuntimeConstraint_Diesel
 - test_Diesel.cpp, 365
- testPostConstructionAttributes_ElectricalLoad
 - test_ElectricalLoad.cpp, 417
- testPostConstructionAttributes_Model
 - test_Model.cpp, 448
- testProductionConstraint_Solar
 - test_Solar.cpp, 381
- testProductionConstraint_Tidal
 - test_Tidal.cpp, 387
- testProductionConstraint_Wave
 - test_Wave.cpp, 393
- testProductionConstraint_Wind
 - test_Wind.cpp, 400
- testProductionLookup_Wave
 - test_Wave.cpp, 394
- testProductionOverride_Solar
 - test_Solar.cpp, 382
- testTruth
 - testing_utils.cpp, 466

- testing_utils.h, 472
- TIDAL
 - Renewable.h, 290
- Tidal, 230
 - __checkInputs, 234
 - __computeCubicProductionkW, 234
 - __computeExponentialProductionkW, 235
 - __computeLookupProductionkW, 236
 - __getGenericCapitalCost, 236
 - __getGenericOpMaintCost, 237
 - __writeSummary, 237
 - __writeTimeSeries, 239
 - ~Tidal, 234
 - commit, 239
 - computeProductionkW, 240
 - design_speed_ms, 242
 - handleReplacement, 241
 - power_model, 242
 - power_model_string, 242
 - Tidal, 232
- Tidal.h
 - N_TIDAL_POWER_PRODUCTION_MODELS, 293
 - TIDAL_POWER_CUBIC, 293
 - TIDAL_POWER_EXPONENTIAL, 293
 - TIDAL_POWER_LOOKUP, 293
 - TidalPowerProductionModel, 292
- TIDAL_POWER_CUBIC
 - Tidal.h, 293
- TIDAL_POWER_EXPONENTIAL
 - Tidal.h, 293
- TIDAL_POWER_LOOKUP
 - Tidal.h, 293
- TidalInputs, 243
 - capital_cost, 244
 - design_speed_ms, 244
 - operation_maintenance_cost_kWh, 244
 - power_model, 244
 - renewable_inputs, 244
 - resource_key, 244
- TidalPowerProductionModel
 - Tidal.h, 292
- time_since_last_start_hrs
 - Diesel, 59
- time_vec_hrs
 - ElectricalLoad, 68
- total_discharge_kWh
 - Storage, 227
- total_dispatch_discharge_kWh
 - Model, 152
- total_dispatch_kWh
 - Production, 178
- total_emissions
 - Combustion, 23
 - Model, 152
- total_fuel_consumed_L
 - Combustion, 23
 - Model, 152
- total_renewable_dispatch_kWh
 - Model, 153
- TURBINE_EFFICIENCY_INTERP_KEY
 - Hydro.h, 286
- turbine_flow_vec_m3hr
 - Hydro, 89
- turbine_type
 - Hydro, 89
 - HydroInputs, 92
- type
 - Combustion, 24
 - Noncombustion, 161
 - Renewable, 188
 - Storage, 227
- type_str
 - Production, 179
 - Storage, 228
- value
 - PYBIND11_Combustion.cpp, 307, 308
 - PYBIND11_Controller.cpp, 332
 - PYBIND11_Hydro.cpp, 314
 - PYBIND11_Noncombustion.cpp, 315
 - PYBIND11_Renewable.cpp, 322
 - PYBIND11_Storage.cpp, 343
 - PYBIND11_Tidal.cpp, 325
 - PYBIND11_Wave.cpp, 327, 328
 - PYBIND11_Wind.cpp, 329
- WAVE
 - Renewable.h, 290
- Wave, 245
 - __checkInputs, 249
 - __computeGaussianProductionkW, 250
 - __computeLookupProductionkW, 251
 - __computeParaboloidProductionkW, 251
 - __getGenericCapitalCost, 252
 - __getGenericOpMaintCost, 252
 - __writeSummary, 252
 - __writeTimeSeries, 254
 - ~Wave, 249
 - commit, 255
 - computeProductionkW, 256
 - design_energy_period_s, 257
 - design_significant_wave_height_m, 258
 - handleReplacement, 257
 - power_model, 258
 - power_model_string, 258
 - Wave, 247
- Wave.h
 - N_WAVE_POWER_PRODUCTION_MODELS, 294
 - WAVE_POWER_GAUSSIAN, 294
 - WAVE_POWER_LOOKUP, 294
 - WAVE_POWER_PARABOLOID, 294
 - WavePowerProductionModel, 294
- WAVE_POWER_GAUSSIAN
 - Wave.h, 294
- WAVE_POWER_LOOKUP

- Wave.h, [294](#)
- WAVE_POWER_PARABOLOID
 - Wave.h, [294](#)
- WaveInputs, [259](#)
 - capital_cost, [260](#)
 - design_energy_period_s, [260](#)
 - design_significant_wave_height_m, [260](#)
 - operation_maintenance_cost_kWh, [260](#)
 - path_2_normalized_performance_matrix, [260](#)
 - power_model, [261](#)
 - renewable_inputs, [261](#)
 - resource_key, [261](#)
- WavePowerProductionModel
 - Wave.h, [294](#)
- WIND
 - Renewable.h, [290](#)
- Wind, [262](#)
 - __checkInputs, [265](#)
 - __computeCubicProductionkW, [266](#)
 - __computeExponentialProductionkW, [267](#)
 - __computeLookupProductionkW, [267](#)
 - __getGenericCapitalCost, [268](#)
 - __getGenericOpMaintCost, [268](#)
 - __writeSummary, [269](#)
 - __writeTimeSeries, [270](#)
 - ~Wind, [265](#)
 - commit, [271](#)
 - computeProductionkW, [272](#)
 - design_speed_ms, [274](#)
 - handleReplacement, [273](#)
 - power_model, [274](#)
 - power_model_string, [274](#)
 - Wind, [264](#)
- Wind.h
 - N_WIND_POWER_PRODUCTION_MODELS, [295](#)
 - WIND_POWER_CUBIC, [295](#)
 - WIND_POWER_EXPONENTIAL, [295](#)
 - WIND_POWER_LOOKUP, [295](#)
 - WindPowerProductionModel, [295](#)
- WIND_POWER_CUBIC
 - Wind.h, [295](#)
- WIND_POWER_EXPONENTIAL
 - Wind.h, [295](#)
- WIND_POWER_LOOKUP
 - Wind.h, [295](#)
- WindInputs, [275](#)
 - capital_cost, [276](#)
 - design_speed_ms, [276](#)
 - operation_maintenance_cost_kWh, [276](#)
 - power_model, [276](#)
 - renewable_inputs, [276](#)
 - resource_key, [276](#)
- WindPowerProductionModel
 - Wind.h, [295](#)
- writeResults
 - Combustion, [19](#)
 - Model, [149](#)
 - Noncombustion, [160](#)
- Renewable, [187](#)
- Storage, [222](#)
- x_vec
 - InterpolatorStruct1D, [107](#)
 - InterpolatorStruct2D, [109](#)
- y_vec
 - InterpolatorStruct1D, [107](#)
 - InterpolatorStruct2D, [109](#)
- z_matrix
 - InterpolatorStruct2D, [109](#)