

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 __computeRenewableProduction()	28
4.3.3.2 __constructCombustionMap()	28
4.3.3.3 __getRenewableProduction()	30
4.3.3.4 __handleCombustionDispatch()	31
4.3.3.5 __handleNoncombustionDispatch()	34
4.3.3.6 __handleRenewableDispatch()	35
4.3.3.7 __handleStorageCharging()	36
4.3.3.8 __handleStorageDischarging()	37
4.3.3.9 applyDispatchControl()	38
4.3.3.10 clear()	41
4.3.3.11 init()	41
4.3.3.12 setControlMode()	42
4.3.3.13 setLoadOperatingReserveFactor()	42
4.3.3.14 setMaxOperatingReserveFactor()	43
4.3.4 Member Data Documentation	43
4.3.4.1 combustion_map	43
4.3.4.2 control_mode	43

4.3.4.3 control_string	44
4.3.4.4 load_operating_reserve_factor	44
4.3.4.5 max_operating_reserve_factor	44
4.3.4.6 missed_load_vec_kW	44
4.3.4.7 net_load_vec_kW	44
4.3.4.8 required_operating_reserve_kW	44
4.3.4.9 storage_discharge_bool_vec	45
4.4 Diesel Class Reference	45
4.4.1 Detailed Description	47
4.4.2 Constructor & Destructor Documentation	47
4.4.2.1 Diesel() [1/2]	47
4.4.2.2 Diesel() [2/2]	47
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()	51
4.4.3.4 __getGenericFuelSlope()	51
4.4.3.5 __getGenericOpMaintCost()	52
4.4.3.6 __handleStartStop()	52
4.4.3.7 __writeSummary()	53
4.4.3.8 __writeTimeSeries()	55
4.4.3.9 commit()	56
4.4.3.10 handleReplacement()	57
4.4.3.11 requestProductionkW()	57
4.4.4 Member Data Documentation	58
4.4.4.1 minimum_load_ratio	58
4.4.4.2 minimum_runtime_hrs	58
4.4.4.3 time_since_last_start_hrs	58
4.5 DieselInputs Struct Reference	59
4.5.1 Detailed Description	60
4.5.2 Member Data Documentation	60
4.5.2.1 capital_cost	60
4.5.2.2 CH4_emissions_intensity_kgL	60
4.5.2.3 CO2_emissions_intensity_kgL	61
4.5.2.4 CO_emissions_intensity_kgL	61
4.5.2.5 combustion_inputs	61
4.5.2.6 fuel_cost_L	61
4.5.2.7 linear_fuel_intercept_LkWh	61
4.5.2.8 linear_fuel_slope_LkWh	61
4.5.2.9 minimum_load_ratio	62
4.5.2.10 minimum_runtime_hrs	62

4.5.2.11 NOx_emissions_intensity_kgL	62
4.5.2.12 operation_maintenance_cost_kWh	62
4.5.2.13 PM_emissions_intensity_kgL	62
4.5.2.14 replace_running_hrs	62
4.5.2.15 SOx_emissions_intensity_kgL	63
4.6 ElectricalLoad Class Reference	63
4.6.1 Detailed Description	64
4.6.2 Constructor & Destructor Documentation	64
4.6.2.1 ElectricalLoad() [1/2]	64
4.6.2.2 ElectricalLoad() [2/2]	64
4.6.2.3 ~ElectricalLoad()	64
4.6.3 Member Function Documentation	64
4.6.3.1 clear()	65
4.6.3.2 readLoadData()	65
4.6.4 Member Data Documentation	66
4.6.4.1 dt_vec_hrs	66
4.6.4.2 load_vec_kW	66
4.6.4.3 max_load_kW	66
4.6.4.4 mean_load_kW	67
4.6.4.5 min_load_kW	67
4.6.4.6 n_points	67
4.6.4.7 n_years	67
4.6.4.8 path_2_electrical_load_time_series	67
4.6.4.9 time_vec_hrs	67
4.7 Emissions Struct Reference	68
4.7.1 Detailed Description	68
4.7.2 Member Data Documentation	68
4.7.2.1 CH4_kg	68
4.7.2.2 CO2_kg	68
4.7.2.3 CO_kg	69
4.7.2.4 NOx_kg	69
4.7.2.5 PM_kg	69
4.7.2.6 SOx_kg	69
4.8 Hydro Class Reference	70
4.8.1 Detailed Description	72
4.8.2 Constructor & Destructor Documentation	72
4.8.2.1 Hydro() [1/2]	72
4.8.2.2 Hydro() [2/2]	73
4.8.2.3 ~Hydro()	74
4.8.3 Member Function Documentation	74
4.8.3.1 __checkInputs()	74
4.8.3.2 __flowToPower()	75

4.8.3.3	__getAcceptableFlow()	75
4.8.3.4	__getAvailableFlow()	76
4.8.3.5	__getEfficiencyFactor()	76
4.8.3.6	__getGenericCapitalCost()	77
4.8.3.7	__getGenericOpMaintCost()	78
4.8.3.8	__getMaximumFlowm3hr()	78
4.8.3.9	__getMinimumFlowm3hr()	78
4.8.3.10	__initInterpolator()	79
4.8.3.11	__powerToFlow()	80
4.8.3.12	__updateState()	81
4.8.3.13	__writeSummary()	82
4.8.3.14	__writeTimeSeries()	84
4.8.3.15	commit()	84
4.8.3.16	handleReplacement()	85
4.8.3.17	requestProductionkW()	85
4.8.4	Member Data Documentation	86
4.8.4.1	fluid_density_kgm3	87
4.8.4.2	init_reservoir_state	87
4.8.4.3	maximum_flow_m3hr	87
4.8.4.4	minimum_flow_m3hr	87
4.8.4.5	minimum_power_kW	87
4.8.4.6	net_head_m	87
4.8.4.7	reservoir_capacity_m3	88
4.8.4.8	spill_rate_vec_m3hr	88
4.8.4.9	stored_volume_m3	88
4.8.4.10	stored_volume_vec_m3	88
4.8.4.11	turbine_flow_vec_m3hr	88
4.8.4.12	turbine_type	88
4.9	HydroInputs Struct Reference	89
4.9.1	Detailed Description	90
4.9.2	Member Data Documentation	90
4.9.2.1	capital_cost	90
4.9.2.2	fluid_density_kgm3	90
4.9.2.3	init_reservoir_state	90
4.9.2.4	net_head_m	90
4.9.2.5	noncombustion_inputs	90
4.9.2.6	operation_maintenance_cost_kWh	91
4.9.2.7	reservoir_capacity_m3	91
4.9.2.8	resource_key	91
4.9.2.9	turbine_type	91
4.10	Interpolator Class Reference	91
4.10.1	Detailed Description	93

4.10.2 Constructor & Destructor Documentation	93
4.10.2.1 Interpolator()	93
4.10.2.2 ~Interpolator()	93
4.10.3 Member Function Documentation	93
4.10.3.1 __checkBounds1D()	93
4.10.3.2 __checkBounds2D()	94
4.10.3.3 __checkDataKey1D()	95
4.10.3.4 __checkDataKey2D()	96
4.10.3.5 __getDataStringMatrix()	96
4.10.3.6 __getInterpolationIndex()	97
4.10.3.7 __isNonNumeric()	97
4.10.3.8 __readData1D()	98
4.10.3.9 __readData2D()	99
4.10.3.10 __splitCommaSeparatedString()	100
4.10.3.11 __throwReadError()	101
4.10.3.12 addData1D()	101
4.10.3.13 addData2D()	102
4.10.3.14 interp1D()	102
4.10.3.15 interp2D()	103
4.10.4 Member Data Documentation	104
4.10.4.1 interp_map_1D	104
4.10.4.2 interp_map_2D	104
4.10.4.3 path_map_1D	104
4.10.4.4 path_map_2D	104
4.11 InterpolatorStruct1D Struct Reference	105
4.11.1 Detailed Description	105
4.11.2 Member Data Documentation	105
4.11.2.1 max_x	105
4.11.2.2 min_x	105
4.11.2.3 n_points	106
4.11.2.4 x_vec	106
4.11.2.5 y_vec	106
4.12 InterpolatorStruct2D Struct Reference	106
4.12.1 Detailed Description	107
4.12.2 Member Data Documentation	107
4.12.2.1 max_x	107
4.12.2.2 max_y	107
4.12.2.3 min_x	107
4.12.2.4 min_y	107
4.12.2.5 n_cols	107
4.12.2.6 n_rows	108
4.12.2.7 x_vec	108

4.12.2.8 y_vec	108
4.12.2.9 z_matrix	108
4.13 Lilon Class Reference	109
4.13.1 Detailed Description	111
4.13.2 Constructor & Destructor Documentation	111
4.13.2.1 Lilon() [1/2]	111
4.13.2.2 Lilon() [2/2]	112
4.13.2.3 ~Lilon()	113
4.13.3 Member Function Documentation	113
4.13.3.1 __checkInputs()	113
4.13.3.2 __getBcal()	115
4.13.3.3 __getEacal()	116
4.13.3.4 __getGenericCapitalCost()	116
4.13.3.5 __getGenericOpMaintCost()	117
4.13.3.6 __handleDegradation()	117
4.13.3.7 __modelDegradation()	118
4.13.3.8 __toggleDepleted()	118
4.13.3.9 __writeSummary()	119
4.13.3.10 __writeTimeSeries()	120
4.13.3.11 commitCharge()	121
4.13.3.12 commitDischarge()	122
4.13.3.13 getAcceptablekW()	122
4.13.3.14 getAvailablekW()	123
4.13.3.15 handleReplacement()	124
4.13.4 Member Data Documentation	124
4.13.4.1 charging_efficiency	124
4.13.4.2 degradation_a_cal	125
4.13.4.3 degradation_alpha	125
4.13.4.4 degradation_B_hat_cal_0	125
4.13.4.5 degradation_beta	125
4.13.4.6 degradation_Ea_cal_0	125
4.13.4.7 degradation_r_cal	125
4.13.4.8 degradation_s_cal	126
4.13.4.9 discharging_efficiency	126
4.13.4.10 dynamic_energy_capacity_kWh	126
4.13.4.11 dynamic_power_capacity_kW	126
4.13.4.12 gas_constant_JmolK	126
4.13.4.13 hysteresis_SOC	126
4.13.4.14 init_SOC	127
4.13.4.15 max_SOC	127
4.13.4.16 min_SOC	127
4.13.4.17 power_degradation_flag	127

4.13.4.18 replace_SOH	127
4.13.4.19 SOH	127
4.13.4.20 SOH_vec	128
4.13.4.21 temperature_K	128
4.14 LilonInputs Struct Reference	128
4.14.1 Detailed Description	129
4.14.2 Member Data Documentation	130
4.14.2.1 capital_cost	130
4.14.2.2 charging_efficiency	130
4.14.2.3 degradation_a_cal	130
4.14.2.4 degradation_alpha	130
4.14.2.5 degradation_B_hat_cal_0	130
4.14.2.6 degradation_beta	131
4.14.2.7 degradation_Ea_cal_0	131
4.14.2.8 degradation_r_cal	131
4.14.2.9 degradation_s_cal	131
4.14.2.10 discharging_efficiency	131
4.14.2.11 gas_constant_JmolK	131
4.14.2.12 hysteresis_SOC	132
4.14.2.13 init_SOC	132
4.14.2.14 max_SOC	132
4.14.2.15 min_SOC	132
4.14.2.16 operation_maintenance_cost_kWh	132
4.14.2.17 power_degradation_flag	132
4.14.2.18 replace_SOH	133
4.14.2.19 storage_inputs	133
4.14.2.20 temperature_K	133
4.15 Model Class Reference	133
4.15.1 Detailed Description	135
4.15.2 Constructor & Destructor Documentation	135
4.15.2.1 Model() [1/2]	136
4.15.2.2 Model() [2/2]	136
4.15.2.3 ~Model()	136
4.15.3 Member Function Documentation	136
4.15.3.1 __checkInputs()	137
4.15.3.2 __computeEconomics()	137
4.15.3.3 __computeFuelAndEmissions()	138
4.15.3.4 __computeLevellizedCostOfEnergy()	138
4.15.3.5 __computeNetPresentCost()	139
4.15.3.6 __writeSummary()	140
4.15.3.7 __writeTimeSeries()	142
4.15.3.8 addDiesel()	143

4.15.3.9 addHydro()	144
4.15.3.10 addLilon()	144
4.15.3.11 addResource() [1/2]	145
4.15.3.12 addResource() [2/2]	145
4.15.3.13 addSolar()	146
4.15.3.14 addTidal()	146
4.15.3.15 addWave()	146
4.15.3.16 addWind()	147
4.15.3.17 clear()	147
4.15.3.18 reset()	148
4.15.3.19 run()	148
4.15.3.20 writeResults()	149
4.15.4 Member Data Documentation	150
4.15.4.1 combustion_ptr_vec	150
4.15.4.2 controller	150
4.15.4.3 electrical_load	150
4.15.4.4 levellized_cost_of_energy_kWh	151
4.15.4.5 net_present_cost	151
4.15.4.6 noncombustion_ptr_vec	151
4.15.4.7 renewable_ptr_vec	151
4.15.4.8 resources	151
4.15.4.9 storage_ptr_vec	151
4.15.4.10 total_dispatch_discharge_kWh	152
4.15.4.11 total_emissions	152
4.15.4.12 total_fuel_consumed_L	152
4.15.4.13 total_renewable_dispatch_kWh	152
4.16 ModelInputs Struct Reference	152
4.16.1 Detailed Description	153
4.16.2 Member Data Documentation	153
4.16.2.1 control_mode	153
4.16.2.2 load_operating_reserve_factor	153
4.16.2.3 max_operating_reserve_factor	153
4.16.2.4 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	firmness_factor	188
4.21.4.2	resource_key	189
4.21.4.3	type	189
4.22	RenewableInputs Struct Reference	189
4.22.1	Detailed Description	190
4.22.2	Member Data Documentation	190
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()	192
4.23.2.2	~Resources()	192
4.23.3	Member Function Documentation	192
4.23.3.1	__checkResourceKey1D() [1/2]	192
4.23.3.2	__checkResourceKey1D() [2/2]	193
4.23.3.3	__checkResourceKey2D()	194
4.23.3.4	__checkTimePoint()	194
4.23.3.5	__readHydroResource()	195
4.23.3.6	__readSolarResource()	196
4.23.3.7	__readTidalResource()	197
4.23.3.8	__readWaveResource()	198
4.23.3.9	__readWindResource()	199
4.23.3.10	__throwLengthError()	200
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	203
4.23.4.2	path_map_2D	203
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	207
4.24.2 Constructor & Destructor Documentation	207
4.24.2.1 Solar() [1/2]	207
4.24.2.2 Solar() [2/2]	207
4.24.2.3 ~Solar()	209
4.24.3 Member Function Documentation	209
4.24.3.1 __checkInputs()	209
4.24.3.2 __computeDetailedProductionkW()	211
4.24.3.3 __computeSimpleProductionkW()	211
4.24.3.4 __getAngleOfIncidenceRad()	212
4.24.3.5 __getBeamIrradiancekWm2()	213
4.24.3.6 __getDeclinationRad()	213
4.24.3.7 __getDiffuseHorizontalIrradiancekWm2()	214
4.24.3.8 __getDiffuseIrradiancekWm2()	214
4.24.3.9 __getDirectNormalIrradiancekWm2()	215
4.24.3.10 __getEclipticLongitudeRad()	215
4.24.3.11 __getGenericCapitalCost()	216
4.24.3.12 __getGenericOpMaintCost()	216
4.24.3.13 __getGreenwichMeanSiderialTimeHrs()	217
4.24.3.14 __getGroundReflectedIrradiancekWm2()	217
4.24.3.15 __getHourAngleRad()	218
4.24.3.16 __getLocalMeanSiderialTimeHrs()	218
4.24.3.17 __getMeanAnomalyRad()	219
4.24.3.18 __getMeanLongitudeDeg()	219
4.24.3.19 __getObliquityOfEclipticRad()	220
4.24.3.20 __getPlaneOfArrayIrradiancekWm2()	220
4.24.3.21 __getRightAscensionRad()	222
4.24.3.22 __getSolarAltitudeRad()	222
4.24.3.23 __getSolarAzimuthRad()	223
4.24.3.24 __getSolarZenithRad()	224
4.24.3.25 __writeSummary()	225
4.24.3.26 __writeTimeSeries()	226
4.24.3.27 commit()	227
4.24.3.28 computeProductionkW()	228
4.24.3.29 handleReplacement()	229
4.24.4 Member Data Documentation	229
4.24.4.1 albedo_ground_reflectance	229
4.24.4.2 derating	230
4.24.4.3 julian_day	230
4.24.4.4 latitude_deg	230
4.24.4.5 latitude_rad	230
4.24.4.6 longitude_deg	230

4.24.4.7 longitude_rad	230
4.24.4.8 panel_azimuth_deg	231
4.24.4.9 panel_azimuth_rad	231
4.24.4.10 panel_tilt_deg	231
4.24.4.11 panel_tilt_rad	231
4.24.4.12 power_model	231
4.24.4.13 power_model_string	231
4.25 SolarInputs Struct Reference	232
4.25.1 Detailed Description	233
4.25.2 Member Data Documentation	233
4.25.2.1 albedo_ground_reflectance	233
4.25.2.2 capital_cost	233
4.25.2.3 derating	233
4.25.2.4 firmness_factor	234
4.25.2.5 julian_day	234
4.25.2.6 latitude_deg	234
4.25.2.7 longitude_deg	234
4.25.2.8 operation_maintenance_cost_kWh	234
4.25.2.9 panel_azimuth_deg	234
4.25.2.10 panel_tilt_deg	235
4.25.2.11 power_model	235
4.25.2.12 renewable_inputs	235
4.25.2.13 resource_key	235
4.26 Storage Class Reference	236
4.26.1 Detailed Description	238
4.26.2 Constructor & Destructor Documentation	238
4.26.2.1 Storage() [1/2]	238
4.26.2.2 Storage() [2/2]	238
4.26.2.3 ~Storage()	239
4.26.3 Member Function Documentation	239
4.26.3.1 __checkInputs()	240
4.26.3.2 __computeRealDiscountAnnual()	240
4.26.3.3 __writeSummary()	241
4.26.3.4 __writeTimeSeries()	241
4.26.3.5 commitCharge()	241
4.26.3.6 commitDischarge()	242
4.26.3.7 computeEconomics()	242
4.26.3.8 getAcceptablekW()	243
4.26.3.9 getAvailablekW()	243
4.26.3.10 handleReplacement()	243
4.26.3.11 writeResults()	244
4.26.4 Member Data Documentation	244

4.26.4.1 capital_cost	244
4.26.4.2 capital_cost_vec	245
4.26.4.3 charge_kWh	245
4.26.4.4 charge_vec_kWh	245
4.26.4.5 charging_power_vec_kW	245
4.26.4.6 discharging_power_vec_kW	245
4.26.4.7 energy_capacity_kWh	246
4.26.4.8 interpolator	246
4.26.4.9 is_depleted	246
4.26.4.10 is_sunk	246
4.26.4.11 levlized_cost_of_energy_kWh	246
4.26.4.12 n_points	246
4.26.4.13 n_replacements	247
4.26.4.14 n_years	247
4.26.4.15 net_present_cost	247
4.26.4.16 nominal_discount_annual	247
4.26.4.17 nominal_inflation_annual	247
4.26.4.18 operation_maintenance_cost_kWh	247
4.26.4.19 operation_maintenance_cost_vec	248
4.26.4.20 power_capacity_kW	248
4.26.4.21 power_kW	248
4.26.4.22 print_flag	248
4.26.4.23 real_discount_annual	248
4.26.4.24 total_discharge_kWh	248
4.26.4.25 type	249
4.26.4.26 type_str	249
4.27 StorageInputs Struct Reference	249
4.27.1 Detailed Description	249
4.27.2 Member Data Documentation	250
4.27.2.1 energy_capacity_kWh	250
4.27.2.2 is_sunk	250
4.27.2.3 nominal_discount_annual	250
4.27.2.4 nominal_inflation_annual	250
4.27.2.5 power_capacity_kW	250
4.27.2.6 print_flag	251
4.28 Tidal Class Reference	251
4.28.1 Detailed Description	253
4.28.2 Constructor & Destructor Documentation	253
4.28.2.1 Tidal() [1/2]	253
4.28.2.2 Tidal() [2/2]	253
4.28.2.3 ~Tidal()	255
4.28.3 Member Function Documentation	255

4.28.3.1	__checkInputs()	255
4.28.3.2	__computeCubicProductionkW()	256
4.28.3.3	__computeExponentialProductionkW()	257
4.28.3.4	__computeLookupProductionkW()	257
4.28.3.5	__getGenericCapitalCost()	258
4.28.3.6	__getGenericOpMaintCost()	258
4.28.3.7	__writeSummary()	258
4.28.3.8	__writeTimeSeries()	260
4.28.3.9	commit()	261
4.28.3.10	computeProductionkW()	261
4.28.3.11	handleReplacement()	262
4.28.4	Member Data Documentation	263
4.28.4.1	design_speed_ms	263
4.28.4.2	power_model	263
4.28.4.3	power_model_string	263
4.29	TidallInputs Struct Reference	264
4.29.1	Detailed Description	265
4.29.2	Member Data Documentation	265
4.29.2.1	capital_cost	265
4.29.2.2	design_speed_ms	265
4.29.2.3	firmness_factor	265
4.29.2.4	operation_maintenance_cost_kWh	265
4.29.2.5	power_model	266
4.29.2.6	renewable_inputs	266
4.29.2.7	resource_key	266
4.30	Wave Class Reference	266
4.30.1	Detailed Description	268
4.30.2	Constructor & Destructor Documentation	268
4.30.2.1	Wave() [1/2]	268
4.30.2.2	Wave() [2/2]	268
4.30.2.3	~Wave()	270
4.30.3	Member Function Documentation	270
4.30.3.1	__checkInputs()	270
4.30.3.2	__computeGaussianProductionkW()	271
4.30.3.3	__computeLookupProductionkW()	272
4.30.3.4	__computeParaboloidProductionkW()	272
4.30.3.5	__getGenericCapitalCost()	274
4.30.3.6	__getGenericOpMaintCost()	275
4.30.3.7	__writeSummary()	275
4.30.3.8	__writeTimeSeries()	276
4.30.3.9	commit()	277
4.30.3.10	computeProductionkW()	278

4.30.3.11 handleReplacement()	279
4.30.4 Member Data Documentation	280
4.30.4.1 design_energy_period_s	280
4.30.4.2 design_significant_wave_height_m	280
4.30.4.3 power_model	280
4.30.4.4 power_model_string	280
4.31 WaveInputs Struct Reference	281
4.31.1 Detailed Description	282
4.31.2 Member Data Documentation	282
4.31.2.1 capital_cost	282
4.31.2.2 design_energy_period_s	282
4.31.2.3 design_significant_wave_height_m	282
4.31.2.4 firmness_factor	282
4.31.2.5 operation_maintenance_cost_kWh	283
4.31.2.6 path_2_normalized_performance_matrix	283
4.31.2.7 power_model	283
4.31.2.8 renewable_inputs	283
4.31.2.9 resource_key	283
4.32 Wind Class Reference	284
4.32.1 Detailed Description	285
4.32.2 Constructor & Destructor Documentation	286
4.32.2.1 Wind() [1/2]	286
4.32.2.2 Wind() [2/2]	286
4.32.2.3 ~Wind()	287
4.32.3 Member Function Documentation	287
4.32.3.1 __checkInputs()	288
4.32.3.2 __computeCubicProductionkW()	288
4.32.3.3 __computeExponentialProductionkW()	289
4.32.3.4 __computeLookupProductionkW()	290
4.32.3.5 __getGenericCapitalCost()	290
4.32.3.6 __getGenericOpMaintCost()	291
4.32.3.7 __writeSummary()	291
4.32.3.8 __writeTimeSeries()	292
4.32.3.9 commit()	293
4.32.3.10 computeProductionkW()	294
4.32.3.11 handleReplacement()	295
4.32.4 Member Data Documentation	296
4.32.4.1 design_speed_ms	296
4.32.4.2 power_model	296
4.32.4.3 power_model_string	296
4.33 WindInputs Struct Reference	297
4.33.1 Detailed Description	298

4.33.2 Member Data Documentation	298
4.33.2.1 capital_cost	298
4.33.2.2 design_speed_ms	298
4.33.2.3 firmness_factor	298
4.33.2.4 operation_maintenance_cost_kWh	298
4.33.2.5 power_model	299
4.33.2.6 renewable_inputs	299
4.33.2.7 resource_key	299
5 File Documentation	301
5.1 header/Controller.h File Reference	301
5.1.1 Detailed Description	302
5.1.2 Enumeration Type Documentation	302
5.1.2.1 ControlMode	302
5.2 header/doxygen_cite.h File Reference	302
5.2.1 Detailed Description	302
5.3 header/ElectricalLoad.h File Reference	303
5.3.1 Detailed Description	303
5.4 header/Interpolator.h File Reference	303
5.4.1 Detailed Description	304
5.5 header/Model.h File Reference	304
5.5.1 Detailed Description	305
5.6 header/Production/Combustion/Combustion.h File Reference	305
5.6.1 Detailed Description	306
5.6.2 Enumeration Type Documentation	306
5.6.2.1 CombustionType	306
5.6.2.2 FuelMode	306
5.7 header/Production/Combustion/Diesel.h File Reference	308
5.7.1 Detailed Description	309
5.8 header/Production/Noncombustion/Hydro.h File Reference	309
5.8.1 Detailed Description	310
5.8.2 Enumeration Type Documentation	310
5.8.2.1 HydroInterpKeys	310
5.8.2.2 HydroTurbineType	310
5.9 header/Production/Noncombustion/Noncombustion.h File Reference	311
5.9.1 Enumeration Type Documentation	312
5.9.1.1 NoncombustionType	312
5.10 header/Production/Production.h File Reference	312
5.10.1 Detailed Description	313
5.11 header/Production/Renewable/Renewable.h File Reference	313
5.11.1 Detailed Description	314
5.11.2 Enumeration Type Documentation	314

5.11.2.1 RenewableType	314
5.12 header/Production/Renewable/Solar.h File Reference	314
5.12.1 Detailed Description	315
5.12.2 Enumeration Type Documentation	315
5.12.2.1 SolarPowerProductionModel	315
5.13 header/Production/Renewable/Tidal.h File Reference	316
5.13.1 Detailed Description	317
5.13.2 Enumeration Type Documentation	317
5.13.2.1 TidalPowerProductionModel	317
5.14 header/Production/Renewable/Wave.h File Reference	317
5.14.1 Detailed Description	318
5.14.2 Enumeration Type Documentation	318
5.14.2.1 WavePowerProductionModel	318
5.15 header/Production/Renewable/Wind.h File Reference	319
5.15.1 Detailed Description	320
5.15.2 Enumeration Type Documentation	320
5.15.2.1 WindPowerProductionModel	320
5.16 header/Resources.h File Reference	320
5.16.1 Detailed Description	321
5.17 header/std_includes.h File Reference	321
5.17.1 Detailed Description	322
5.17.2 Macro Definition Documentation	322
5.17.2.1 _USE_MATH_DEFINES	322
5.18 header/Storage/Lilon.h File Reference	322
5.18.1 Detailed Description	323
5.19 header/Storage/Storage.h File Reference	323
5.19.1 Detailed Description	324
5.19.2 Enumeration Type Documentation	324
5.19.2.1 StorageType	324
5.20 projects/example.cpp File Reference	324
5.20.1 Function Documentation	325
5.20.1.1 main()	325
5.21 pybindings/PYBIND11_PGM.cpp File Reference	329
5.21.1 Detailed Description	329
5.21.2 Function Documentation	330
5.21.2.1 PYBIND11_MODULE()	330
5.22 pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp File Reference	330
5.22.1 Detailed Description	331
5.22.2 Function Documentation	331
5.22.2.1 def()	331
5.22.2.2 def_readwrite() [1/4]	331
5.22.2.3 def_readwrite() [2/4]	332

5.22.2.4 def_readwrite() [3/4]	332
5.22.2.5 def_readwrite() [4/4]	332
5.22.2.6 value() [1/2]	332
5.22.2.7 value() [2/2]	332
5.22.3 Variable Documentation	332
5.22.3.1 def_readwrite	332
5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference	333
5.23.1 Detailed Description	334
5.23.2 Function Documentation	334
5.23.2.1 def()	334
5.23.2.2 def_readwrite() [1/8]	334
5.23.2.3 def_readwrite() [2/8]	334
5.23.2.4 def_readwrite() [3/8]	334
5.23.2.5 def_readwrite() [4/8]	335
5.23.2.6 def_readwrite() [5/8]	335
5.23.2.7 def_readwrite() [6/8]	335
5.23.2.8 def_readwrite() [7/8]	335
5.23.2.9 def_readwrite() [8/8]	335
5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp File Reference	336
5.24.1 Detailed Description	337
5.24.2 Function Documentation	337
5.24.2.1 def()	337
5.24.2.2 def_readwrite() [1/9]	337
5.24.2.3 def_readwrite() [2/9]	337
5.24.2.4 def_readwrite() [3/9]	337
5.24.2.5 def_readwrite() [4/9]	338
5.24.2.6 def_readwrite() [5/9]	338
5.24.2.7 def_readwrite() [6/9]	338
5.24.2.8 def_readwrite() [7/9]	338
5.24.2.9 def_readwrite() [8/9]	338
5.24.2.10 def_readwrite() [9/9]	338
5.24.2.11 value() [1/2]	339
5.24.2.12 value() [2/2]	339
5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp File Reference	339
5.25.1 Detailed Description	339
5.25.2 Function Documentation	340
5.25.2.1 def()	340
5.25.2.2 value()	340
5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference	340
5.26.1 Detailed Description	342
5.26.2 Function Documentation	342
5.26.2.1 def()	342

5.26.2.2 def_readwrite() [1/17]	342
5.26.2.3 def_readwrite() [2/17]	342
5.26.2.4 def_readwrite() [3/17]	343
5.26.2.5 def_readwrite() [4/17]	343
5.26.2.6 def_readwrite() [5/17]	343
5.26.2.7 def_readwrite() [6/17]	343
5.26.2.8 def_readwrite() [7/17]	343
5.26.2.9 def_readwrite() [8/17]	344
5.26.2.10 def_readwrite() [9/17]	344
5.26.2.11 def_readwrite() [10/17]	344
5.26.2.12 def_readwrite() [11/17]	344
5.26.2.13 def_readwrite() [12/17]	344
5.26.2.14 def_readwrite() [13/17]	345
5.26.2.15 def_readwrite() [14/17]	345
5.26.2.16 def_readwrite() [15/17]	345
5.26.2.17 def_readwrite() [16/17]	345
5.26.2.18 def_readwrite() [17/17]	345
5.27 pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp File Reference	346
5.27.1 Detailed Description	346
5.27.2 Function Documentation	346
5.27.2.1 def()	346
5.27.2.2 def_readwrite()	347
5.27.2.3 value() [1/2]	347
5.27.2.4 value() [2/2]	347
5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference	347
5.28.1 Detailed Description	348
5.28.2 Function Documentation	348
5.28.2.1 def()	348
5.28.2.2 def_readwrite() [1/7]	348
5.28.2.3 def_readwrite() [2/7]	349
5.28.2.4 def_readwrite() [3/7]	349
5.28.2.5 def_readwrite() [4/7]	349
5.28.2.6 def_readwrite() [5/7]	349
5.28.2.7 def_readwrite() [6/7]	349
5.28.2.8 def_readwrite() [7/7]	349
5.28.2.9 value()	350
5.29 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference	350
5.29.1 Detailed Description	351
5.29.2 Function Documentation	351
5.29.2.1 def()	351
5.29.2.2 def_readwrite() [1/4]	351
5.29.2.3 def_readwrite() [2/4]	351

5.29.2.4 def_readwrite() [3/4]	351
5.29.2.5 def_readwrite() [4/4]	352
5.29.2.6 value() [1/2]	352
5.29.2.7 value() [2/2]	352
5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference	352
5.30.1 Detailed Description	353
5.30.2 Function Documentation	353
5.30.2.1 def()	353
5.30.2.2 def_readwrite() [1/5]	353
5.30.2.3 def_readwrite() [2/5]	354
5.30.2.4 def_readwrite() [3/5]	354
5.30.2.5 def_readwrite() [4/5]	354
5.30.2.6 def_readwrite() [5/5]	354
5.30.2.7 value() [1/2]	354
5.30.2.8 value() [2/2]	354
5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference	355
5.31.1 Detailed Description	355
5.31.2 Function Documentation	356
5.31.2.1 def()	356
5.31.2.2 def_readwrite() [1/4]	356
5.31.2.3 def_readwrite() [2/4]	356
5.31.2.4 def_readwrite() [3/4]	356
5.31.2.5 def_readwrite() [4/4]	356
5.31.2.6 value() [1/2]	357
5.31.2.7 value() [2/2]	357
5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference	357
5.32.1 Detailed Description	358
5.32.2 Function Documentation	358
5.32.2.1 def() [1/3]	358
5.32.2.2 def() [2/3]	358
5.32.2.3 def() [3/3]	358
5.32.2.4 def_readwrite() [1/3]	358
5.32.2.5 def_readwrite() [2/3]	359
5.32.2.6 def_readwrite() [3/3]	359
5.32.2.7 value()	359
5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference	359
5.33.1 Detailed Description	360
5.33.2 Function Documentation	360
5.33.2.1 def_readwrite() [1/4]	360
5.33.2.2 def_readwrite() [2/4]	360
5.33.2.3 def_readwrite() [3/4]	360
5.33.2.4 def_readwrite() [4/4]	361

5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference	361
5.34.1 Detailed Description	361
5.34.2 Function Documentation	362
5.34.2.1 def()	362
5.34.2.2 def_readwrite() [1/7]	362
5.34.2.3 def_readwrite() [2/7]	362
5.34.2.4 def_readwrite() [3/7]	362
5.34.2.5 def_readwrite() [4/7]	362
5.34.2.6 def_readwrite() [5/7]	363
5.34.2.7 def_readwrite() [6/7]	363
5.34.2.8 def_readwrite() [7/7]	363
5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference	363
5.35.1 Detailed Description	364
5.35.2 Function Documentation	364
5.35.2.1 def_readwrite()	364
5.35.3 Variable Documentation	364
5.35.3.1 def_readwrite	364
5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference	365
5.36.1 Detailed Description	365
5.36.2 Function Documentation	365
5.36.2.1 def_readwrite() [1/2]	365
5.36.2.2 def_readwrite() [2/2]	366
5.37 pybindings/snippets/Storage/PYBIND11_Lilon.cpp File Reference	366
5.37.1 Detailed Description	367
5.37.2 Function Documentation	367
5.37.2.1 def_readwrite() [1/9]	367
5.37.2.2 def_readwrite() [2/9]	367
5.37.2.3 def_readwrite() [3/9]	367
5.37.2.4 def_readwrite() [4/9]	368
5.37.2.5 def_readwrite() [5/9]	368
5.37.2.6 def_readwrite() [6/9]	368
5.37.2.7 def_readwrite() [7/9]	368
5.37.2.8 def_readwrite() [8/9]	368
5.37.2.9 def_readwrite() [9/9]	369
5.37.3 Variable Documentation	369
5.37.3.1 def_readwrite	369
5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference	369
5.38.1 Detailed Description	370
5.38.2 Function Documentation	370
5.38.2.1 def_readwrite() [1/2]	370
5.38.2.2 def_readwrite() [2/2]	370
5.38.2.3 value()	370

5.38.3 Variable Documentation	371
5.38.3.1 def_readwrite	371
5.39 source/Controller.cpp File Reference	371
5.39.1 Detailed Description	371
5.40 source/ElectricalLoad.cpp File Reference	372
5.40.1 Detailed Description	372
5.41 source/Interpolator.cpp File Reference	372
5.41.1 Detailed Description	372
5.42 source/Model.cpp File Reference	373
5.42.1 Detailed Description	373
5.43 source/Production/Combustion/Combustion.cpp File Reference	373
5.43.1 Detailed Description	374
5.44 source/Production/Combustion/Diesel.cpp File Reference	374
5.44.1 Detailed Description	374
5.45 source/Production/Noncombustion/Hydro.cpp File Reference	374
5.45.1 Detailed Description	375
5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference	375
5.46.1 Detailed Description	375
5.47 source/Production/Production.cpp File Reference	376
5.47.1 Detailed Description	376
5.48 source/Production/Renewable/Renewable.cpp File Reference	376
5.48.1 Detailed Description	376
5.49 source/Production/Renewable/Solar.cpp File Reference	377
5.49.1 Detailed Description	377
5.50 source/Production/Renewable/Tidal.cpp File Reference	377
5.50.1 Detailed Description	378
5.51 source/Production/Renewable/Wave.cpp File Reference	378
5.51.1 Detailed Description	378
5.52 source/Production/Renewable/Wind.cpp File Reference	378
5.52.1 Detailed Description	379
5.53 source/Resources.cpp File Reference	379
5.53.1 Detailed Description	379
5.54 source/Storage/Lilon.cpp File Reference	380
5.54.1 Detailed Description	380
5.55 source/Storage/Storage.cpp File Reference	380
5.55.1 Detailed Description	380
5.56 test/source/Production/Combustion/test_Combustion.cpp File Reference	381
5.56.1 Detailed Description	381
5.56.2 Function Documentation	381
5.56.2.1 main()	382
5.56.2.2 testConstruct_Combustion()	382
5.57 test/source/Production/Combustion/test_Diesel.cpp File Reference	383

5.57.1 Detailed Description	384
5.57.2 Function Documentation	385
5.57.2.1 main()	385
5.57.2.2 testBadConstruct_Diesel()	385
5.57.2.3 testCapacityConstraint_Diesel()	386
5.57.2.4 testCommit_Diesel()	386
5.57.2.5 testConstruct_Diesel()	388
5.57.2.6 testConstructLookup_Diesel()	389
5.57.2.7 testEconomics_Diesel()	390
5.57.2.8 testFuelConsumptionEmissions_Diesel()	390
5.57.2.9 testFuelLookup_Diesel()	392
5.57.2.10 testMinimumLoadRatioConstraint_Diesel()	393
5.57.2.11 testMinimumRuntimeConstraint_Diesel()	394
5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference	394
5.58.1 Detailed Description	395
5.58.2 Function Documentation	395
5.58.2.1 main()	395
5.58.2.2 testCommit_Hydro()	396
5.58.2.3 testConstruct_Hydro()	397
5.58.2.4 testEfficiencyInterpolation_Hydro()	398
5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference	400
5.59.1 Detailed Description	400
5.59.2 Function Documentation	400
5.59.2.1 main()	401
5.59.2.2 testConstruct_Noncombustion()	401
5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference	402
5.60.1 Detailed Description	402
5.60.2 Function Documentation	403
5.60.2.1 main()	403
5.60.2.2 testConstruct_Renewable()	403
5.61 test/source/Production/Renewable/test_Solar.cpp File Reference	404
5.61.1 Detailed Description	405
5.61.2 Function Documentation	405
5.61.2.1 main()	405
5.61.2.2 testBadConstruct_Solar()	406
5.61.2.3 testCommit_Solar()	406
5.61.2.4 testConstruct_Solar()	408
5.61.2.5 testDetailed_Solar()	409
5.61.2.6 testEconomics_Solar()	411
5.61.2.7 testProductionConstraint_Solar()	411
5.61.2.8 testProductionOverride_Solar()	412
5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference	413

5.62.1 Detailed Description	414
5.62.2 Function Documentation	414
5.62.2.1 main()	415
5.62.2.2 testBadConstruct_Tidal()	415
5.62.2.3 testCommit_Tidal()	416
5.62.2.4 testConstruct_Tidal()	417
5.62.2.5 testEconomics_Tidal()	418
5.62.2.6 testProductionConstraint_Tidal()	419
5.63 test/source/Production/Renewable/test_Wave.cpp File Reference	419
5.63.1 Detailed Description	420
5.63.2 Function Documentation	420
5.63.2.1 main()	420
5.63.2.2 testBadConstruct_Wave()	421
5.63.2.3 testCommit_Wave()	421
5.63.2.4 testConstruct_Wave()	423
5.63.2.5 testConstructLookup_Wave()	424
5.63.2.6 testEconomics_Wave()	424
5.63.2.7 testProductionConstraint_Wave()	425
5.63.2.8 testProductionLookup_Wave()	425
5.64 test/source/Production/Renewable/test_Wind.cpp File Reference	426
5.64.1 Detailed Description	427
5.64.2 Function Documentation	427
5.64.2.1 main()	428
5.64.2.2 testBadConstruct_Wind()	428
5.64.2.3 testCommit_Wind()	429
5.64.2.4 testConstruct_Wind()	430
5.64.2.5 testEconomics_Wind()	431
5.64.2.6 testProductionConstraint_Wind()	432
5.65 test/source/Production/test_Production.cpp File Reference	432
5.65.1 Detailed Description	433
5.65.2 Function Documentation	433
5.65.2.1 main()	433
5.65.2.2 testBadConstruct_Production()	434
5.65.2.3 testConstruct_Production()	434
5.66 test/source/Storage/test_Lilon.cpp File Reference	435
5.66.1 Detailed Description	436
5.66.2 Function Documentation	436
5.66.2.1 main()	437
5.66.2.2 testBadConstruct_Lilon()	437
5.66.2.3 testCommitCharge_Lilon()	438
5.66.2.4 testCommitDischarge_Lilon()	438
5.66.2.5 testConstruct_Lilon()	439

5.67 test/source/Storage/test_Storage.cpp File Reference	441
5.67.1 Detailed Description	441
5.67.2 Function Documentation	441
5.67.2.1 main()	441
5.67.2.2 testBadConstruct_Storage()	442
5.67.2.3 testConstruct_Storage()	442
5.68 test/source/test_Controller.cpp File Reference	443
5.68.1 Detailed Description	444
5.68.2 Function Documentation	444
5.68.2.1 main()	444
5.68.2.2 testConstruct_Controller()	445
5.69 test/source/test_ElectricalLoad.cpp File Reference	445
5.69.1 Detailed Description	446
5.69.2 Function Documentation	446
5.69.2.1 main()	446
5.69.2.2 testConstruct_ElectricalLoad()	447
5.69.2.3 testDataRead_ElectricalLoad()	447
5.69.2.4 testPostConstructionAttributes_ElectricalLoad()	448
5.70 test/source/test_Interpolator.cpp File Reference	449
5.70.1 Detailed Description	450
5.70.2 Function Documentation	450
5.70.2.1 main()	450
5.70.2.2 testBadIndexing1D_Interpolator()	451
5.70.2.3 testConstruct_Interpolator()	451
5.70.2.4 testDataRead1D_Interpolator()	452
5.70.2.5 testDataRead2D_Interpolator()	453
5.70.2.6 testInterpolation1D_Interpolator()	455
5.70.2.7 testInterpolation2D_Interpolator()	456
5.70.2.8 testInvalidInterpolation1D_Interpolator()	457
5.70.2.9 testInvalidInterpolation2D_Interpolator()	459
5.71 test/source/test_Model.cpp File Reference	460
5.71.1 Detailed Description	462
5.71.2 Function Documentation	462
5.71.2.1 main()	462
5.71.2.2 testAddDiesel_Model()	464
5.71.2.3 testAddHydro_Model()	465
5.71.2.4 testAddHydroResource_Model()	465
5.71.2.5 testAddLilon_Model()	466
5.71.2.6 testAddSolar_Model()	467
5.71.2.7 testAddSolar_productionOverride_Model()	467
5.71.2.8 testAddSolarResource_Model()	468
5.71.2.9 testAddTidal_Model()	469

5.71.2.10 testAddTidalResource_Model()	470
5.71.2.11 testAddWave_Model()	471
5.71.2.12 testAddWaveResource_Model()	472
5.71.2.13 testAddWind_Model()	474
5.71.2.14 testAddWindResource_Model()	474
5.71.2.15 testBadConstruct_Model()	475
5.71.2.16 testConstruct_Model()	476
5.71.2.17 testEconomics_Model()	476
5.71.2.18 testElectricalLoadData_Model()	477
5.71.2.19 testFuelConsumptionEmissions_Model()	478
5.71.2.20 testLoadBalance_Model()	479
5.71.2.21 testOperatingReserve_Model()	482
5.71.2.22 testPostConstructionAttributes_Model()	483
5.72 test/source/test_Resources.cpp File Reference	483
5.72.1 Detailed Description	484
5.72.2 Function Documentation	484
5.72.2.1 main()	485
5.72.2.2 testAddHydroResource_Resources()	486
5.72.2.3 testAddSolarResource_Resources()	487
5.72.2.4 testAddTidalResource_Resources()	488
5.72.2.5 testAddWaveResource_Resources()	490
5.72.2.6 testAddWindResource_Resources()	491
5.72.2.7 testBadAdd_Resources()	493
5.72.2.8 testConstruct_Resources()	494
5.73 test/utls/testing_utils.cpp File Reference	494
5.73.1 Detailed Description	495
5.73.2 Function Documentation	495
5.73.2.1 expectedErrorNotDetected()	495
5.73.2.2 printGold()	496
5.73.2.3 printGreen()	496
5.73.2.4 printRed()	496
5.73.2.5 testFloatEquals()	497
5.73.2.6 testGreaterThan()	497
5.73.2.7 testGreaterThanOrEqualTo()	498
5.73.2.8 testLessThan()	499
5.73.2.9 testLessThanOrEqualTo()	499
5.73.2.10 testTruth()	500
5.74 test/utls/testing_utils.h File Reference	500
5.74.1 Detailed Description	501
5.74.2 Macro Definition Documentation	502
5.74.2.1 FLOAT_TOLERANCE	502
5.74.3 Function Documentation	502

5.74.3.1 expectedErrorNotDetected()	502
5.74.3.2 printGold()	502
5.74.3.3 printGreen()	503
5.74.3.4 printRed()	503
5.74.3.5 testFloatEquals()	503
5.74.3.6 testGreaterThan()	504
5.74.3.7 testGreaterThanOrEqualTo()	505
5.74.3.8 testLessThan()	505
5.74.3.9 testLessThanOrEqualTo()	506
5.74.3.10 testTruth()	506
Bibliography	510
Index	511

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

CombustionInputs	24
Controller	26
DieselInputs	59
ElectricalLoad	63
Emissions	68
HydroInputs	89
Interpolator	91
InterpolatorStruct1D	105
InterpolatorStruct2D	106
LilonInputs	128
Model	133
ModelInputs	152
NoncombustionInputs	162
Production	163
Combustion	9
Diesel	45
Noncombustion	154
Hydro	70
Renewable	181
Solar	204
Tidal	251
Wave	266
Wind	284
ProductionInputs	179
RenewableInputs	189
Resources	190
SolarInputs	232
Storage	236
Lilon	109
StorageInputs	249
TidalInputs	264
WaveInputs	281
WindInputs	297

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	45
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 . . .	59
ElectricalLoad	A class which contains time and electrical load data. Intended to serve as a component class of Model	63
Emissions	A structure which bundles the emitted masses of various emissions chemistries	68
Hydro	A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not)	70
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	89
Interpolator	A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies	91
InterpolatorStruct1D	A struct which holds two parallel vectors for use in 1D interpolation	105
InterpolatorStruct2D	A struct which holds two parallel vectors and a matrix for use in 2D interpolation	106
Lilon	A derived class of Storage which models energy storage by way of lithium-ion batteries	109

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	128
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	133
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)	152
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	232
Storage	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	236
StorageInputs	A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input	249
Tidal	A derived class of the Renewable branch of Production which models tidal production	251
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	264
Wave	A derived class of the Renewable branch of Production which models wave production	266
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	281
Wind	A derived class of the Renewable branch of Production which models wind production	284
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	297

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	301
header/ doxygen_cite.h	
Header file which simply cites the doxygen tool	302
header/ ElectricalLoad.h	
Header file for the ElectricalLoad class	303
header/ Interpolator.h	
Header file for the Interpolator class	303
header/ Model.h	
Header file for the Model class	304
header/ Resources.h	
Header file for the Resources class	320
header/ std_includes.h	
Header file which simply batches together some standard includes	321
header/Production/ Production.h	
Header file for the Production class	312
header/Production/Combustion/ Combustion.h	
Header file for the Combustion class	305
header/Production/Combustion/ Diesel.h	
Header file for the Diesel class	308
header/Production/Noncombustion/ Hydro.h	
Header file for the Hydro class	309
header/Production/Noncombustion/ Noncombustion.h	
Header file for the Noncombustion class	311
header/Production/Renewable/ Renewable.h	
Header file for the Renewable class	313
header/Production/Renewable/ Solar.h	
Header file for the Solar class	314
header/Production/Renewable/ Tidal.h	
Header file for the Tidal class	316
header/Production/Renewable/ Wave.h	
Header file for the Wave class	317
header/Production/Renewable/ Wind.h	
Header file for the Wind class	319
header/Storage/ Lilon.h	
Header file for the Lilon class	322

header/Storage/Storage.h	
Header file for the Storage class	323
projects/example.cpp	324
pybindings/PYBIND11_PGM.cpp	
Bindings file for PGMcpp	329
pybindings/snippets/PYBIND11_Controller.cpp	
Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp	357
pybindings/snippets/PYBIND11_ElectricalLoad.cpp	
Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp	359
pybindings/snippets/PYBIND11_Interpolator.cpp	
Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp	361
pybindings/snippets/PYBIND11_Model.cpp	
Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp	363
pybindings/snippets/PYBIND11_Resources.cpp	
Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp	365
pybindings/snippets/Production/PYBIND11_Production.cpp	
Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp	340
pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp	
Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp	330
pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp	
Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp	333
pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp	
Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp	336
pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp	
Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp	339
pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp	
Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp	346
pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp	
Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp	347
pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp	
Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp	350
pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp	
Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp	352
pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp	
Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp	355
pybindings/snippets/Storage/PYBIND11_Lilon.cpp	
Bindings file for the Lilon class. Intended to be #include'd in PYBIND11_PGM.cpp	366
pybindings/snippets/Storage/PYBIND11_Storage.cpp	
Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp	369
source/Controller.cpp	
Implementation file for the Controller class	371
source/ElectricalLoad.cpp	
Implementation file for the ElectricalLoad class	372
source/Interpolator.cpp	
Implementation file for the Interpolator class	372
source/Model.cpp	
Implementation file for the Model class	373
source/Resources.cpp	
Implementation file for the Resources class	379
source/Production/Production.cpp	
Implementation file for the Production class	376
source/Production/Combustion/Combustion.cpp	
Implementation file for the Combustion class	373
source/Production/Combustion/Diesel.cpp	
Implementation file for the Diesel class	374
source/Production/Noncombustion/Hydro.cpp	
Implementation file for the Hydro class	374

source/Production/Noncombustion/ Noncombustion.cpp	
Implementation file for the Noncombustion class	375
source/Production/Renewable/ Renewable.cpp	
Implementation file for the Renewable class	376
source/Production/Renewable/ Solar.cpp	
Implementation file for the Solar class	377
source/Production/Renewable/ Tidal.cpp	
Implementation file for the Tidal class	377
source/Production/Renewable/ Wave.cpp	
Implementation file for the Wave class	378
source/Production/Renewable/ Wind.cpp	
Implementation file for the Wind class	378
source/Storage/ Lilon.cpp	
Implementation file for the Lilon class	380
source/Storage/ Storage.cpp	
Implementation file for the Storage class	380
test/source/ test_Controller.cpp	
Testing suite for Controller class	443
test/source/ test_ElectricalLoad.cpp	
Testing suite for ElectricalLoad class	445
test/source/ test_Interpolator.cpp	
Testing suite for Interpolator class	449
test/source/ test_Model.cpp	
Testing suite for Model class	460
test/source/ test_Resources.cpp	
Testing suite for Resources class	483
test/source/Production/ test_Production.cpp	
Testing suite for Production class	432
test/source/Production/Combustion/ test_Combustion.cpp	
Testing suite for Combustion class	381
test/source/Production/Combustion/ test_Diesel.cpp	
Testing suite for Diesel class	383
test/source/Production/Noncombustion/ test_Hydro.cpp	
Testing suite for Hydro class	394
test/source/Production/Noncombustion/ test_Noncombustion.cpp	
Testing suite for Noncombustion class	400
test/source/Production/Renewable/ test_Renewable.cpp	
Testing suite for Renewable class	402
test/source/Production/Renewable/ test_Solar.cpp	
Testing suite for Solar class	404
test/source/Production/Renewable/ test_Tidal.cpp	
Testing suite for Tidal class	413
test/source/Production/Renewable/ test_Wave.cpp	
Testing suite for Wave class	419
test/source/Production/Renewable/ test_Wind.cpp	
Testing suite for Wind class	426
test/source/Storage/ test_Lilon.cpp	
Testing suite for Lilon class	435
test/source/Storage/ test_Storage.cpp	
Testing suite for Storage class	441
test/utills/ testing_utils.cpp	
Implementation file for various PGMcpp testing utilities	494
test/utills/ testing_utils.h	
Header file for various PGMcpp testing utilities	500

Chapter 4

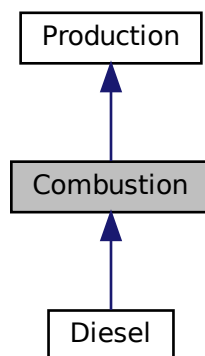
Class Documentation

4.1 Combustion Class Reference

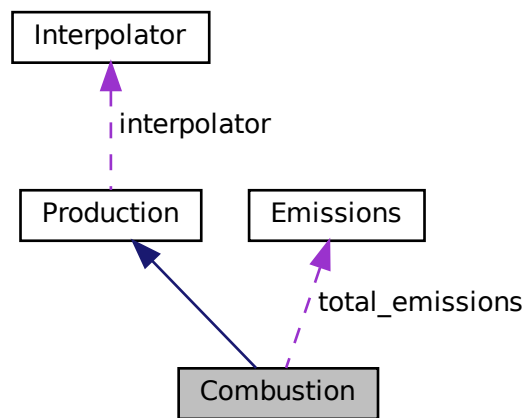
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

```
#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:

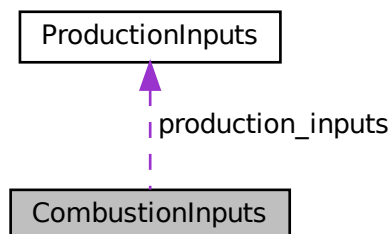
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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](#))
Method to set control mode of [Controller](#).
- void [setLoadOperatingReserveFactor](#) (double)
Method to set [Controller](#) load_operating_reserve_factor attribute.
- void [setMaxOperatingReserveFactor](#) (double)
Method to set [Controller](#) max_operating_reserve_factor attribute.
- 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.
- double load_operating_reserve_factor
An operating reserve factor [0, 1] to cover random fluctuations in load.
- double max_operating_reserve_factor
*A maximum reserve factor [0, 1] that limits the required overall operating reserve to, at most, factor * load_kW.*
- double required_operating_reserve_kW
A required operating reserve [kW], to absorb load and [Renewable](#) production fluctuations.
- std::vector< bool > storage_discharge_bool_vec
A boolean vector attribute to track which [Storage](#) assets have been discharged in each time step.
- 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 `__computeRenewableProduction` (`ElectricalLoad *`, `std::vector< Renewable * > *`, `Resources *`)
Helper method to compute and record [Renewable](#) production, net load.
- void `__constructCombustionMap` (`std::vector< Combustion * > *`)
Helper method to construct a [Combustion](#) map, for use in determining.
- double `__getRenewableProduction` (`int`, `double`, `Renewable *`, `Resources *`)
Helper method to compute the production from the given [Renewable](#) asset at the given point in time.
- double `__handleStorageDischarging` (`int`, `double`, `double`, `std::vector< Storage * > *`)
Helper method to handle the discharging of available [Storage](#) assets.
- double `__handleNoncombustionDispatch` (`int`, `double`, `double`, `std::vector< Noncombustion * > *`, `Resources *`)
Helper method to handle the dispatch of [Noncombustion](#) assets.
- double `__handleCombustionDispatch` (`int`, `double`, `double`, `double`, `double`, `double`, `std::vector< Combustion * > *`, `bool`)
Helper method to handle the dispatch of [Combustion](#) assets. Accounts for load operating reserve and [Renewable](#) production reserve (firmness), as per conversation with BC [Hydro](#) (April 2024).
- double `__handleRenewableDispatch` (`int`, `double`, `double`, `std::vector< Renewable * > *`)
Helper method to handle the dispatch of [Renewable](#) assets.
- void `__handleStorageCharging` (`int`, `double`, `std::vector< Storage * > *`, `std::vector< Combustion * > *`, `std::vector< Noncombustion * > *`, `std::vector< Renewable * > *`)
Helper method to handle the charging of available [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.

```
940 {
941     return;
942 } /* Controller() */
```

4.3.2.2 ~Controller()

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

Destructor for the [Controller](#) class.

```
1351 {
1352     this->clear();
1353
1354     return;
1355 } /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 `__computeRenewableProduction()`

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

Helper method to compute and record [Renewable](#) production, net load.

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     double dt_hrs = 0;
84     double load_kW = 0;
85     double net_load_kW = 0;
86     double production_kW = 0;
87
88     Renewable* renewable_ptr;
89
90     for (int timestep = 0; timestep < electrical_load_ptr->n_points; timestep++) {
91         dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
92         load_kW = electrical_load_ptr->load_vec_kW[timestep];
93         net_load_kW = load_kW;
94
95         for (size_t asset = 0; asset < renewable_ptr_vec_ptr->size(); asset++) {
96             renewable_ptr = renewable_ptr_vec_ptr->at(asset);
97
98             production_kW = this->__getRenewableProduction(
99                 timestep,
100                 dt_hrs,
101                 renewable_ptr,
102                 resources_ptr
103             );
104
105             renewable_ptr->production_vec_kW[timestep] = production_kW;
106
107             net_load_kW -= production_kW;
108         }
109
110         this->net_load_vec_kW[timestep] = net_load_kW;
111     }
112
113     return;
114 } /* __computeRenewableProduction() */
```

4.3.3.2 `__constructCombustionMap()`

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

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

Parameters

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

```

136 {
137     std::string print_str = "Controller :: __constructCombustionMap() ";
138     print_str += "constructing combustion map (dispatch) ";
139
140     // 1. get state table dimensions
141     unsigned int n_cols = combustion_ptr_vec_ptr->size();
142     unsigned long int n_rows = pow(2, n_cols);
143
144     // 2. walk through all possible operating states (on/off) and populate combustion
145     // map, keeping only states with minimum number of assets running.
146     for (unsigned long int row = 0; row < n_rows; row++) {
147         std::vector<bool> state_vec(n_cols, false);
148
149         unsigned int asset_count = 0;
150         unsigned long int x = row;
151         double total_capacity_kW = 0;
152
153         for (unsigned int i = 0; i < n_cols; i++) {
154             if (x <= 0) {
155                 break;
156             }
157
158             if (x % 2 != 0) {
159                 state_vec[i] = true;
160                 total_capacity_kW += combustion_ptr_vec_ptr->at(i)->capacity_kW;
161                 asset_count++;
162             }
163
164             x /= 2;
165         }
166
167         if (this->combustion_map.count(total_capacity_kW) == 0) {
168             this->combustion_map[total_capacity_kW] = state_vec;
169         }
170
171         else {
172             unsigned int incumbent_asset_count = 0;
173
174             for (unsigned int i = 0; i < n_cols; i++) {
175                 if (this->combustion_map[total_capacity_kW][i]) {
176                     incumbent_asset_count++;
177                 }
178             }
179
180             if (asset_count < incumbent_asset_count) {
181                 this->combustion_map[total_capacity_kW] = state_vec;
182             }
183         }
184
185         if (n_cols >= 14) {
186             std::cout << print_str << row + 1 << " / " << n_rows << "\r";
187         }
188     }
189
190     if (n_cols >= 14) {
191         std::cout << print_str << n_rows << " / " << n_rows << " DONE" << std::endl;
192     }
193
194     // 3. sort combustion map by key value (ascending order)
195     /*
196     * Not necessary, since std::map is automatically sorted by key value on insertion.
197     * See https://en.cppreference.com/w/cpp/container/map, namely "std::map is a
198     * sorted associative container that contains key-value pairs with unique keys.
199     * Keys are sorted by using the comparison function Compare."
200     */
201
202     /*
203     // ==== TEST PRINT ==== //
204     std::cout << std::endl << std::endl;
205
206     std::cout << "\t\t";
207     for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
208         std::cout << combustion_ptr_vec_ptr->at(i)->capacity_kW << "\t";
209     }
210     std::cout << std::endl;
211
212     std::map<double, std::vector<bool>>::iterator iter;
213     for (
214         iter = this->combustion_map.begin();
215         iter != this->combustion_map.end();
216         iter++
217     ) {

```

```

218         std::cout << iter->first << ":\t{\t";
219
220         for (size_t i = 0; i < iter->second.size(); i++) {
221             std::cout << iter->second[i] << "\t";
222         }
223         std::cout << "}" << std::endl;
224     }
225
226
227     // ==== END TEST PRINT ==== //
228     */
229
230     return;
231 } /* __constructCombustionTable() */

```

4.3.3.3 __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.

```

267 {
268     double production_kW = 0;
269
270     switch (renewable_ptr->type) {
271         case (RenewableType :: SOLAR): {
272             double resource_value = 0;
273
274             if (not renewable_ptr->normalized_production_series_given) {
275                 resource_value =
276                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
277             }
278
279             production_kW = renewable_ptr->computeProductionkW(
280                 timestep,
281                 dt_hrs,
282                 resource_value
283             );
284
285             break;
286         }
287
288         case (RenewableType :: TIDAL): {
289             double resource_value = 0;
290
291             if (not renewable_ptr->normalized_production_series_given) {
292                 resource_value =
293                     resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
294             }
295
296             production_kW = renewable_ptr->computeProductionkW(
297                 timestep,
298                 dt_hrs,

```

```

299         resource_value
300     );
301
302     break;
303 }
304
305 case (RenewableType :: WAVE): {
306     double significant_wave_height_m = 0;
307     double energy_period_s = 0;
308
309     if (not renewable_ptr->normalized_production_series_given) {
310         significant_wave_height_m =
311             resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0];
312
313         energy_period_s =
314             resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1];
315     }
316
317     production_kW = renewable_ptr->computeProductionkW(
318         timestep,
319         dt_hrs,
320         significant_wave_height_m,
321         energy_period_s
322     );
323
324     break;
325 }
326
327 case (RenewableType :: WIND): {
328     double resource_value = 0;
329
330     if (not renewable_ptr->normalized_production_series_given) {
331         resource_value =
332             resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep];
333     }
334
335     production_kW = renewable_ptr->computeProductionkW(
336         timestep,
337         dt_hrs,
338         resource_value
339     );
340
341     break;
342 }
343
344 default: {
345     std::string error_str = "ERROR: Controller::__getRenewableProduction(): ";
346     error_str += "renewable type ";
347     error_str += std::to_string(renewable_ptr->type);
348     error_str += " not recognized";
349
350     #ifdef _WIN32
351         std::cout << error_str << std::endl;
352     #endif
353
354     throw std::runtime_error(error_str);
355
356     break;
357 }
358 }
359
360 return production_kW;
361 } /* __getRenewableProduction() */

```

4.3.3.4 __handleCombustionDispatch()

```

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

```

Helper method to handle the dispatch of [Combustion](#) assets. Accounts for load operating reserve and [Renewable](#) production reserve (firmness), as per conversation with BC [Hydro](#) (April 2024).

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>load_kW</i>	The load [kW] for this timestep.
<i>remaining_load_kW</i>	The load remaining [kW] before dispatch.
<i>total_renewable_production_kW</i>	The total production [kW] from all Renewable assets.
<i>firm_renewable_production_kW</i>	The firm production [kW] from all Renewable assets.
<i>combustion_ptr_vec_ptr</i>	A pointer to the Combustion pointer vector of the Model .
<i>is_cycle_charging</i>	A flag which indicates whether the Combustion assets are running in cycle charging mode (true) or load following mode (false).

Returns

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

```

636 {
637     // 1. set target dispatch
638     bool operating_reserve_flag = false;
639     double target_dispatch_kW = remaining_load_kW;
640
641     if (target_dispatch_kW < this->required_operating_reserve_kW) {
642         target_dispatch_kW = this->required_operating_reserve_kW;
643         operating_reserve_flag = true;
644     }
645
646     if (target_dispatch_kW < 0) {
647         target_dispatch_kW = 0;
648     }
649
650     // 2. allocate Combustion assets
651     double allocated_capacity_kW = 0;
652
653     std::map<double, std::vector<bool>::iterator> iter = this->combustion_map.begin();
654
655     while (iter != std::prev(this->combustion_map.end(), 1)) {
656         if (target_dispatch_kW <= allocated_capacity_kW) {
657             break;
658         }
659
660         iter++;
661         allocated_capacity_kW = iter->first;
662     }
663
664     // 3. dispatch Combustion assets
665     // sharing load proportionally to individual rated capacities
666     Combustion* combustion_ptr;
667
668     double production_kW = 0;
669     double request_kW = 0;
670     double target_production_kW = target_dispatch_kW;
671
672     for (
673         size_t asset = 0;
674         asset < this->combustion_map[allocated_capacity_kW].size();
675         asset++
676     ) {
677         combustion_ptr = combustion_ptr_vec_ptr->at(asset);
678
679         if (allocated_capacity_kW > 0) {
680             request_kW =
681                 int(this->combustion_map[allocated_capacity_kW][asset]) *
682                 target_production_kW *
683                 (combustion_ptr->capacity_kW / allocated_capacity_kW);
684         }
685
686         else {
687             request_kW = 0;
688         }
689
690         if (is_cycle_charging and request_kW > 0) {
691             if (request_kW < combustion_ptr->cycle_charging_setpoint * combustion_ptr->capacity_kW) {
692                 request_kW = combustion_ptr->cycle_charging_setpoint * combustion_ptr->capacity_kW;
693             }
694         }
695     }

```

```

696         production_kW = combustion_ptr->requestProductionkW(
697             timestep,
698             dt_hrs,
699             request_kW
700         );
701
702         target_dispatch_kW = combustion_ptr->commit(
703             timestep,
704             dt_hrs,
705             production_kW,
706             target_dispatch_kW
707         );
708     }
709
710     // 4. log impact of dispatch
711     if (operating_reserve_flag) {
712         remaining_load_kW -= this->required_operating_reserve_kW - target_dispatch_kW;
713         this->required_operating_reserve_kW = target_dispatch_kW;
714     }
715
716     else {
717         this->required_operating_reserve_kW -= remaining_load_kW - target_dispatch_kW;
718         remaining_load_kW = target_dispatch_kW;
719     }
720
721     return remaining_load_kW;
722 } /* __handleCombustionDispatch() */

```

4.3.3.5 __handleNoncombustionDispatch()

```

double Controller::__handleNoncombustionDispatch (
    int timestep,
    double dt_hrs,
    double remaining_load_kW,
    std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
    Resources * resources_ptr ) [private]
{
    // 1. set target dispatch
    bool operating_reserve_flag = false;
    double target_dispatch_kW = remaining_load_kW;

    if (target_dispatch_kW < this->required_operating_reserve_kW) {
        target_dispatch_kW = this->required_operating_reserve_kW;
        operating_reserve_flag = true;
    }

    if (target_dispatch_kW < 0) {
        target_dispatch_kW = 0;
    }

    // 2. dispatch Noncombustion assets
    Noncombustion* noncombustion_ptr;
    double production_kW = 0;

    for (size_t asset = 0; asset < noncombustion_ptr_vec_ptr->size(); asset++) {
        noncombustion_ptr = noncombustion_ptr_vec_ptr->at(asset);

        switch (noncombustion_ptr->type) {
            case (NoncombustionType::HYDRO): {
                double resource_value = 0;

                if (not noncombustion_ptr->normalized_production_series_given) {
                    resource_value =
                        resources_ptr->resource_map_1D[noncombustion_ptr->resource_key][timestep];
                }

                production_kW = noncombustion_ptr->requestProductionkW(
                    timestep,
                    dt_hrs,
                    target_dispatch_kW,
                    resource_value
                );

                target_dispatch_kW = noncombustion_ptr->commit(
                    timestep,
                    dt_hrs,
                    production_kW,

```

```

541             target_dispatch_kW,
542             resource_value
543         );
544
545         break;
546     }
547
548     default: {
549         production_kW = noncombustion_ptr->requestProductionkW(
550             timestep,
551             dt_hrs,
552             target_dispatch_kW
553         );
554
555         target_dispatch_kW = noncombustion_ptr->commit(
556             timestep,
557             dt_hrs,
558             production_kW,
559             target_dispatch_kW
560         );
561
562         break;
563     }
564 }
565 }
566
567 // 3. log impact of dispatch
568 if (operating_reserve_flag) {
569     remaining_load_kW -= this->required_operating_reserve_kW - target_dispatch_kW;
570     this->required_operating_reserve_kW = target_dispatch_kW;
571 }
572
573 else {
574     this->required_operating_reserve_kW -= remaining_load_kW - target_dispatch_kW;
575     remaining_load_kW = target_dispatch_kW;
576 }
577
578 return remaining_load_kW;
579 } /* __handleNoncombustionDispatch() */

```

4.3.3.6 __handleRenewableDispatch()

```

double Controller::__handleRenewableDispatch (
    int timestep,
    double dt_hrs,
    double remaining_load_kW,
    std::vector< Renewable * > * renewable_ptr_vec_ptr ) [private]
{
558 {
559     // 1. set target dispatch
560     double target_dispatch_kW = remaining_load_kW;
561
562     if (target_dispatch_kW < 0) {
563         target_dispatch_kW = 0;
564     }
565
566     // 2. dispatch Renewable assets
567     Renewable* renewable_ptr;
568     double production_kW = 0;
569
570     for (size_t asset = 0; asset < renewable_ptr_vec_ptr->size(); asset++) {
571         renewable_ptr = renewable_ptr_vec_ptr->at(asset);
572
573         production_kW = renewable_ptr->production_vec_kW[timestep];
574
575         target_dispatch_kW = renewable_ptr->commit(
576             timestep,
577             dt_hrs,
578             production_kW,
579             target_dispatch_kW
580         );
581     }
582
583     // 3. log impact of dispatch
584     remaining_load_kW = target_dispatch_kW;
585
586     return remaining_load_kW;
587 } /* __handleRenewableDispatch() */

```

4.3.3.7 __handleStorageCharging()

```
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 available [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 .

```
830 {
831     double acceptable_kW = 0;
832     double curtailment_kW = 0;
833
834     Storage* storage_ptr;
835     Combustion* combustion_ptr;
836     Noncombustion* noncombustion_ptr;
837     Renewable* renewable_ptr;
838
839     for (
840         size_t storage_asset = 0;
841         storage_asset < storage_ptr_vec_ptr->size();
842         storage_asset++
843     ) {
844         // 1. if already discharged, continue
845         if (this->storage_discharge_bool_vec[storage_asset]) {
846             continue;
847         }
848
849         // 2. get pointer to asset
850         storage_ptr = storage_ptr_vec_ptr->at(storage_asset);
851
852         // 3. attempt to charge from Combustion curtailment first
853         for (size_t asset = 0; asset < combustion_ptr_vec_ptr->size(); asset++) {
854             combustion_ptr = combustion_ptr_vec_ptr->at(asset);
855             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
856
857             if (curtailment_kW <= 0) {
858                 continue;
859             }
860
861             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
862
863             if (acceptable_kW > curtailment_kW) {
864                 acceptable_kW = curtailment_kW;
865             }
866
867             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
868             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
869             storage_ptr->power_kW += acceptable_kW;
870         }
871
872         // 4. attempt to charge from Noncombustion curtailment second
873         for (size_t asset = 0; asset < noncombustion_ptr_vec_ptr->size(); asset++) {
874             noncombustion_ptr = noncombustion_ptr_vec_ptr->at(asset);
875             curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
876
877             if (curtailment_kW <= 0) {
878                 continue;
879             }
880
881             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
882
```

```

883         if (acceptable_kW > curtailment_kW) {
884             acceptable_kW = curtailment_kW;
885         }
886
887         noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
888         noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
889         storage_ptr->power_kW += acceptable_kW;
890     }
891
892     // 5. attempt to charge from Renewable curtailment third
893     for (size_t asset = 0; asset < renewable_ptr_vec_ptr->size(); asset++) {
894         renewable_ptr = renewable_ptr_vec_ptr->at(asset);
895         curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
896
897         if (curtailment_kW <= 0) {
898             continue;
899         }
900
901         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
902
903         if (acceptable_kW > curtailment_kW) {
904             acceptable_kW = curtailment_kW;
905         }
906
907         renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
908         renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
909         storage_ptr->power_kW += acceptable_kW;
910     }
911
912     // 6. commit charge
913     storage_ptr->commitCharge(
914         timestep,
915         dt_hrs,
916         storage_ptr->power_kW
917     );
918 }
919
920 return;
921 } /* __handleStorageCharging() */

```

4.3.3.8 __handleStorageDischarging()

```

double Controller::__handleStorageDischarging (
    int timestep,
    double dt_hrs,
    double net_load_kW,
    std::vector< Storage * > * storage_ptr_vec_ptr ) [private]

```

Helper method to handle the discharging of available [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>remaining_load_kW</i>	The load remaining [kW] before discharging.
<i>storage_ptr_vec_ptr</i>	A pointer to a vector of pointers to the Storage assets.

Returns

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

```

396 {
397     // 1. set target discharge
398     bool operating_reserve_flag = false;
399     double target_discharge_kW = remaining_load_kW;
400
401     if (target_discharge_kW < this->required_operating_reserve_kW) {
402         target_discharge_kW = this->required_operating_reserve_kW;

```

```

403         operating_reserve_flag = true;
404     }
405
406     // 2. immediately return on target_discharge_kW <= 0
407     if (target_discharge_kW <= 0) {
408         return remaining_load_kW;
409     }
410
411     // 3. discharge available Storage assets
412     double discharging_kW = 0;
413
414     Storage* storage_ptr;
415
416     for (size_t asset = 0; asset < storage_ptr_vec_ptr->size(); asset++) {
417         // 3.1. break on vanishing target_discharge_kW
418         if (target_discharge_kW <= 0) {
419             break;
420         }
421
422         // 3.2. get pointer to asset
423         storage_ptr = storage_ptr_vec_ptr->at(asset);
424
425         // 3.3. continue if depleted
426         if (storage_ptr->is_depleted) {
427             continue;
428         }
429
430         // 3.4. get available discharging power
431         discharging_kW = storage_ptr->getAvailablekW(dt_hrs);
432
433         if (discharging_kW > target_discharge_kW) {
434             discharging_kW = target_discharge_kW;
435         }
436
437         // 3.5. commit discharging, log
438         target_discharge_kW = storage_ptr->commitDischarge(
439             timestep,
440             dt_hrs,
441             discharging_kW,
442             target_discharge_kW
443         );
444
445         this->storage_discharge_bool_vec[asset] = true;
446     }
447
448     // 4. log impact of discharge
449     if (operating_reserve_flag) {
450         remaining_load_kW -= this->required_operating_reserve_kW - target_discharge_kW;
451         this->required_operating_reserve_kW = target_discharge_kW;
452     }
453
454     else {
455         this->required_operating_reserve_kW -= remaining_load_kW - target_discharge_kW;
456         remaining_load_kW = target_discharge_kW;
457     }
458
459     return remaining_load_kW;
460 } /* __handleStorageDischarging() */

```

4.3.3.9 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 .
----------------------------	--

Parameters

<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 .

```

1127 {
1128     double dt_hrs = 0;
1129     double load_kW = 0;
1130     double total_renewable_production_kW = 0;
1131     double firm_renewable_production_kW = 0;
1132     double remaining_load_kW = 0;
1133     /*
1134     double required_operating_reserve_before_kW = 0;
1135     double rem_load_test_0 = 0;
1136     double rem_load_test_1 = 0;
1137     double rem_load_test_2 = 0;
1138     double rem_load_test_3 = 0;
1139     double rem_load_test_4 = 0;
1140     */
1141     this->required_operating_reserve_kW = 0;
1142     this->storage_discharge_bool_vec.clear();
1143     this->storage_discharge_bool_vec.resize(storage_ptr_vec_ptr->size(), false);
1144
1145     Renewable* renewable_ptr;
1146
1147     for (int timestep = 0; timestep < electrical_load_ptr->n_points; timestep++) {
1148         // 1. get dt_hrs and load
1149         dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
1150         load_kW = electrical_load_ptr->load_vec_kW[timestep];
1151
1152         // 2. compute firm and total Renewable productions
1153         total_renewable_production_kW = 0;
1154         firm_renewable_production_kW = 0;
1155
1156         for (size_t asset = 0; asset < renewable_ptr_vec_ptr->size(); asset++) {
1157             renewable_ptr = renewable_ptr_vec_ptr->at(asset);
1158
1159             total_renewable_production_kW += renewable_ptr->production_vec_kW[timestep];
1160
1161             firm_renewable_production_kW +=
1162                 renewable_ptr->firmness_factor * renewable_ptr->production_vec_kW[timestep];
1163         }
1164
1165         // 3. compute required operating reserve (load + Renewable), enforce max
1166         this->required_operating_reserve_kW =
1167             this->load_operating_reserve_factor * load_kW +
1168             total_renewable_production_kW - firm_renewable_production_kW;
1169
1170         if (
1171             this->required_operating_reserve_kW >
1172             this->max_operating_reserve_factor * load_kW
1173         ) {
1174             this->required_operating_reserve_kW =
1175                 this->max_operating_reserve_factor * load_kW;
1176         }
1177
1178         //required_operating_reserve_before_kW = this->required_operating_reserve_kW;
1179
1180         // 4. init remaining_load_kW
1181         remaining_load_kW = load_kW - total_renewable_production_kW;
1182
1183         //rem_load_test_0 = remaining_load_kW;
1184
1185         // 5. handle Storage discharging
1186         remaining_load_kW = this->__handleStorageDischarging(
1187             timestep,
1188             dt_hrs,
1189             remaining_load_kW,
1190             storage_ptr_vec_ptr
1191         );
1192
1193         //rem_load_test_1 = remaining_load_kW;
1194
1195         // 6. handle Noncombustion dispatch
1196         remaining_load_kW = this->__handleNoncombustionDispatch(
1197             timestep,
1198             dt_hrs,
1199             remaining_load_kW,
1200             noncombustion_ptr_vec_ptr,

```

```

1201         resources_ptr
1202     );
1203
1204     //rem_load_test_2 = remaining_load_kW;
1205
1206     // 7. handle Combustion dispatch
1207     switch(control_mode) {
1208     case (ControlMode :: LOAD_FOLLOWING): {
1209         remaining_load_kW = this->__handleCombustionDispatch(
1210             timestep,
1211             dt_hrs,
1212             load_kW,
1213             remaining_load_kW,
1214             total_renewable_production_kW,
1215             firm_renewable_production_kW,
1216             combustion_ptr_vec_ptr,
1217             false
1218         );
1219
1220         break;
1221     }
1222
1223     case (ControlMode :: CYCLE_CHARGING): {
1224         bool is_cycle_charging = false;
1225
1226         for (size_t asset = 0; asset < storage_ptr_vec_ptr->size(); asset++) {
1227             if (not this->storage_discharge_bool_vec[asset]) {
1228                 is_cycle_charging = true;
1229                 break;
1230             }
1231         }
1232
1233         remaining_load_kW = this->__handleCombustionDispatch(
1234             timestep,
1235             dt_hrs,
1236             load_kW,
1237             remaining_load_kW,
1238             total_renewable_production_kW,
1239             firm_renewable_production_kW,
1240             combustion_ptr_vec_ptr,
1241             is_cycle_charging
1242         );
1243
1244         break;
1245     }
1246
1247     default: {
1248         std::string error_str = "ERROR: Controller :: setControlMode(): ";
1249         error_str += "control mode ";
1250         error_str += std::to_string(control_mode);
1251         error_str += " not recognized";
1252
1253         #ifdef _WIN32
1254             std::cout << error_str << std::endl;
1255         #endif
1256
1257         throw std::runtime_error(error_str);
1258
1259         break;
1260     }
1261 }
1262
1263 //rem_load_test_3 = remaining_load_kW;
1264
1265 // 8. handle Renewable dispatch
1266 remaining_load_kW += total_renewable_production_kW;
1267
1268 //rem_load_test_4 = remaining_load_kW;
1269
1270 remaining_load_kW = this->__handleRenewableDispatch(
1271     timestep,
1272     dt_hrs,
1273     remaining_load_kW,
1274     renewable_ptr_vec_ptr
1275 );
1276
1277 // 9. handle Storage charging
1278 this->__handleStorageCharging(
1279     timestep,
1280     dt_hrs,
1281     storage_ptr_vec_ptr,
1282     combustion_ptr_vec_ptr,
1283     noncombustion_ptr_vec_ptr,
1284     renewable_ptr_vec_ptr
1285 );
1286
1287 // 10. log missed load, if any

```



```

1288         if (remaining_load_kW > 1e-6) {
1289             this->missed_load_vec_kW[timestep] = remaining_load_kW;
1290         }
1291
1292         // 11. reset storage_discharge_bool_vec
1293         for (size_t asset = 0; asset < storage_ptr_vec_ptr->size(); asset++) {
1294             this->storage_discharge_bool_vec[asset] = false;
1295         }
1296
1297         // 12. test print
1298         /*
1299         if (required_operating_reserve_before_kW < load_kW) {
1300             std::cout << "Timestep: " << timestep << std::endl;
1301             std::cout << "Load: " << load_kW << std::endl;
1302             std::cout << "Req Op Reserve: " << required_operating_reserve_before_kW << std::endl;
1303             std::cout << "Rem Load (before Storage): " << rem_load_test_0 << std::endl;
1304             std::cout << "Rem Load (after Storage): " << rem_load_test_1 << std::endl;
1305             std::cout << "Rem Load (after Noncombustion): " << rem_load_test_2 << std::endl;
1306             std::cout << "Rem Load (after Combustion): " << rem_load_test_3 << std::endl;
1307             std::cout << "Rem Load (before Renewable): " << rem_load_test_4 << std::endl;
1308             std::cout << "Rem Load: " << remaining_load_kW << std::endl;
1309             std::cout << std::endl;
1310         }
1311         */
1312     }
1313
1314     return;
1315 } /* applyDispatchControl() */

```

4.3.3.10 clear()

```

void Controller::clear (
    void )

```

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

```

1330 {
1331     this->net_load_vec_kW.clear();
1332     this->missed_load_vec_kW.clear();
1333     this->combustion_map.clear();
1334
1335     return;
1336 } /* clear() */

```

4.3.3.11 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 .

```

1069 {

```

```

1070 // 1. init vector attributes
1071 this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
1072 this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
1073
1074 // 2. compute Renewable production
1075 this->__computeRenewableProduction(
1076     electrical_load_ptr,
1077     renewable_ptr_vec_ptr,
1078     resources_ptr
1079 );
1080
1081 // 3. construct Combustion table
1082 this->__constructCombustionMap(combustion_ptr_vec_ptr);
1083
1084 return;
1085 } /* init() */

```

4.3.3.12 setControlMode()

```

void Controller::setControlMode (
    ControlMode control_mode )

```

Method to set control mode of [Controller](#).

Parameters

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

```

959 {
960     this->control_mode = control_mode;
961
962     switch(control_mode) {
963         case (ControlMode :: LOAD_FOLLOWING): {
964             this->control_string = "LOAD_FOLLOWING";
965
966             break;
967         }
968
969         case (ControlMode :: CYCLE_CHARGING): {
970             this->control_string = "CYCLE_CHARGING";
971
972             break;
973         }
974
975         default: {
976             std::string error_str = "ERROR: Controller :: setControlMode(): ";
977             error_str += "control mode ";
978             error_str += std::to_string(control_mode);
979             error_str += " not recognized";
980
981             #ifdef _WIN32
982                 std::cout << error_str << std::endl;
983             #endif
984
985             throw std::runtime_error(error_str);
986
987             break;
988         }
989     }
990
991     return;
992 } /* setControlMode() */

```

4.3.3.13 setLoadOperatingReserveFactor()

```

void Controller::setLoadOperatingReserveFactor (
    double load_operating_reserve_factor )

```

Method to set [Controller](#) load_operating_reserve_factor attribute.

Parameters

<i>load_operating_reserve_factor</i>	An operating reserve factor [0, 1] to cover random fluctuations in load.
--------------------------------------	--

```

1010 {
1011     this->load_operating_reserve_factor = load_operating_reserve_factor;
1012
1013     return;
1014 } /* setLoadOperatingReserveFactor() */

```

4.3.3.14 setMaxOperatingReserveFactor()

```

void Controller::setMaxOperatingReserveFactor (
    double max_operating_reserve_factor )

```

Method to set [Controller](#) max_operating_reserve_factor attribute.

Parameters

<i>max_operating_reserve_factor</i>	An operating reserve factor [0, 1] that limits the required overall operating reserve to, at most, factor * load_kW.
-------------------------------------	--

```

1032 {
1033     this->max_operating_reserve_factor = max_operating_reserve_factor;
1034
1035     return;
1036 } /* setMaxOperatingReserveFactor() */

```

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 load_operating_reserve_factor

```
double Controller::load_operating_reserve_factor
```

An operating reserve factor [0, 1] to cover random fluctuations in load.

4.3.4.5 max_operating_reserve_factor

```
double Controller::max_operating_reserve_factor
```

A maximum reserve factor [0, 1] that limits the required overall operating reserve to, at most, factor * load_kW.

4.3.4.6 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.

4.3.4.7 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.

4.3.4.8 required_operating_reserve_kW

```
double Controller::required_operating_reserve_kW
```

A required operating reserve [kW], to absorb load and [Renewable](#) production fluctuations.

4.3.4.9 storage_discharge_bool_vec

```
std::vector<bool> Controller::storage_discharge_bool_vec
```

A boolean vector attribute to track which [Storage](#) assets have been discharged in each time step.

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

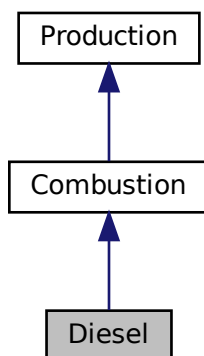
- header/[Controller.h](#)
- source/[Controller.cpp](#)

4.4 Diesel Class Reference

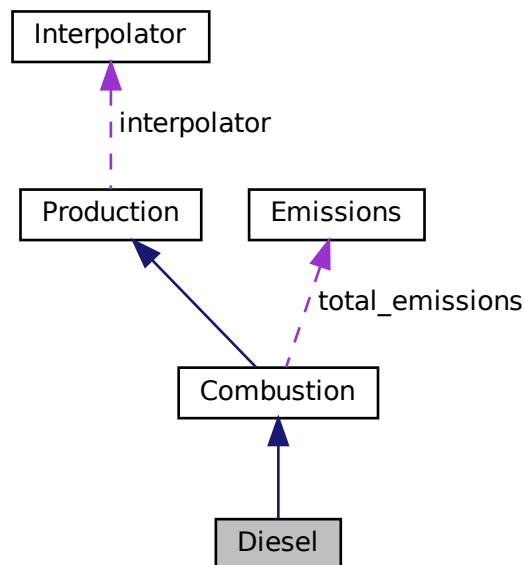
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

```
#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.

```
632 {
633     return;
634 } /* 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.

```

666 :
667 Combustion(
668     n_points,
669     n_years,
670     diesel_inputs.combustion_inputs,
671     time_vec_hrs_ptr
672 )
673 {
674     // 1. check inputs
675     this->__checkInputs(diesel_inputs);
676
677     // 2. set attributes
678     this->type = CombustionType :: DIESEL;
679     this->type_str = "DIESEL";
680
681     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
682
683     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
684
685     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
686     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
687     this->time_since_last_start_hrs = 0;
688
689     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
690     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
691     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
692     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
693     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
694     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
695
696     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
697         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
698     }
699     else {
700         this->linear_fuel_slope_LkWh = diesel_inputs.linear_fuel_slope_LkWh;
701     }
702
703     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
704         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
705     }
706     else {
707         this->linear_fuel_intercept_LkWh = diesel_inputs.linear_fuel_intercept_LkWh;
708     }
709
710     if (diesel_inputs.capital_cost < 0) {
711         this->capital_cost = this->__getGenericCapitalCost();
712     }
713     else {
714         this->capital_cost = diesel_inputs.capital_cost;
715     }
716
717     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
718         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
719     }
720     else {
721         this->operation_maintenance_cost_kWh =
722             diesel_inputs.operation_maintenance_cost_kWh;
723     }
724
725     if (not this->is_sunk) {
726         this->capital_cost_vec[0] = this->capital_cost;
727     }
728
729     // 3. construction print
730     if (this->print_flag) {
731         std::cout << "Diesel object constructed at " << this << std::endl;
732     }
733
734     return;
735 } /* Diesel() */

```


4.4.2.3 ~Diesel()

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

Destructor for the [Diesel](#) class.

```
897 {
898     // 1. destruction print
899     if (this->print_flag) {
900         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
901     }
902
903     return;
904 } /* ~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 << "Cycle Charging Setpoint: " << this->cycle_charging_setpoint << "\n";
416     ofs << "\n";
417
418     ofs << "Fuel Cost: " << this->fuel_cost_L << " per L \n";
419     ofs << "Nominal Fuel Escalation Rate (annual): "
420         << this->nominal_fuel_escalation_annual << " \n";
421     ofs << "Real Fuel Escalation Rate (annual): "
422         << this->real_fuel_escalation_annual << " \n";
423     ofs << "\n";
424
425     ofs << "Fuel Mode: " << this->fuel_mode_str << " \n";

```

```

426     switch (this->fuel_mode) {
427     case (FuelMode :: FUEL_MODE_LINEAR): {
428         ofs << "Linear Fuel Slope: " << this->linear_fuel_slope_LkWh
429             << " L/kWh \n";
430         ofs << "Linear Fuel Intercept Coefficient: "
431             << this->linear_fuel_intercept_LkWh << " L/kWh \n";
432         ofs << "\n";
433         break;
434     }
435
436     case (FuelMode :: FUEL_MODE_LOOKUP): {
437         ofs << "Fuel Consumption Data: " << this->interpolator.path_map_1D[0]
438             << " \n";
439         break;
440     }
441
442     default: {
443         // write nothing!
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
511
512

```

```

513 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
514     this->total_emissions.NOx_kg << " kg "
515     << "(Annual Average: " << this->total_emissions.NOx_kg / this->n_years
516     << " kg/yr) \n";
517
518 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
519     this->total_emissions.SOx_kg << " kg "
520     << "(Annual Average: " << this->total_emissions.SOx_kg / this->n_years
521     << " kg/yr) \n";
522
523 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
524     << "(Annual Average: " << this->total_emissions.CH4_kg / this->n_years
525     << " kg/yr) \n";
526
527 ofs << "Total Particulate Matter (PM) Emissions: " <<
528     this->total_emissions.PM_kg << " kg "
529     << "(Annual Average: " << this->total_emissions.PM_kg / this->n_years
530     << " kg/yr) \n";
531
532 ofs << "\n-----\n\n";
533
534 ofs.close();
535 return;
536 } /* __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](#).

```

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

```

```

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

```

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

```



```

876         this->operation_maintenance_cost_kWh * produced_kWh;
877         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
878     }
879 }
880
881 return load_kW;
882 } /* 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](#).

```

753 {
754     // 1. reset attributes
755     this->time_since_last_start_hrs = 0;
756
757     // 2. invoke base class method
758     Combustion::handleReplacement(timestep);
759
760     return;
761 } /* __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](#).

```

793 {

```

```

794 // 0. given production time series override
795 if (this->normalized_production_series_given) {
796     double production_kW = Production::getProductionkW(timestep);
797
798     return production_kW;
799 }
800
801 // 1. return on request of zero
802 if (request_kW <= 0) {
803     return 0;
804 }
805
806 double deliver_kW = request_kW;
807
808 // 2. enforce capacity constraint
809 if (deliver_kW > this->capacity_kW) {
810     deliver_kW = this->capacity_kW;
811 }
812
813 // 3. enforce minimum load ratio
814 if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
815     deliver_kW = this->minimum_load_ratio * this->capacity_kW;
816 }
817
818 return deliver_kW;
819 } /* 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:

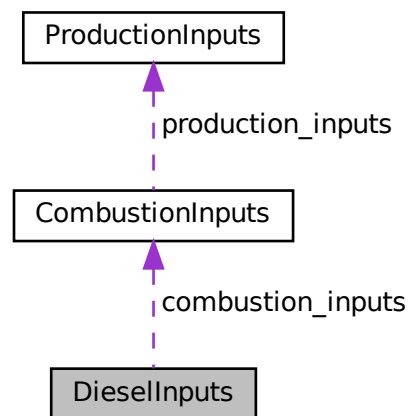
- header/Production/Combustion/[Diesel.h](#)
- source/Production/Combustion/[Diesel.cpp](#)

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:

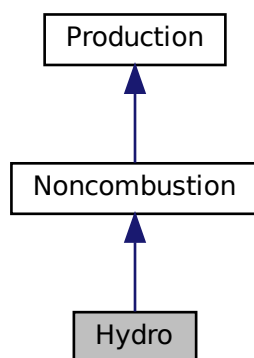
- [header/Production/Combustion/Combustion.h](#)

4.8 Hydro Class Reference

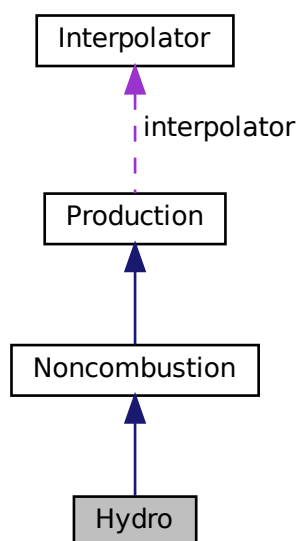
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

```
#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
182         break;
183     }
184
185     case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
186         efficiency_vec = {
187             0.000, 0.400, 0.625, 0.745, 0.810,
188             0.845, 0.880, 0.900, 0.910, 0.900,
189             0.850
190         };
191
192         break;
193     }
194
195     case (HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
196         efficiency_vec = {
197             0.000, 0.265, 0.460, 0.550, 0.650,
198             0.740, 0.805, 0.845, 0.900, 0.880,
199             0.850
200         };
201
202         break;
203     }
204 }
```

```

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:

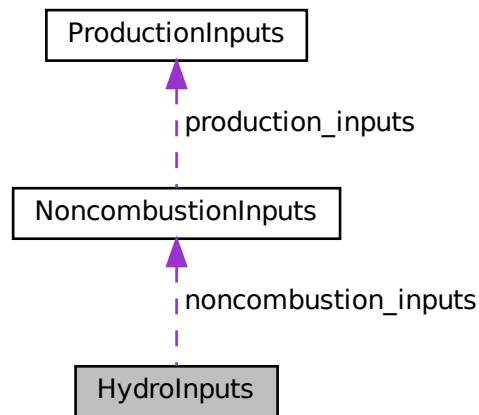
- [header/Production/Noncombustion/Hydro.h](#)
- [source/Production/Noncombustion/Hydro.cpp](#)

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.
Generated by Doxygen	

```

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

<i>data_key</i>	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:

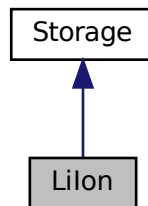
- header/[Interpolator.h](#)

4.13 Lilon Class Reference

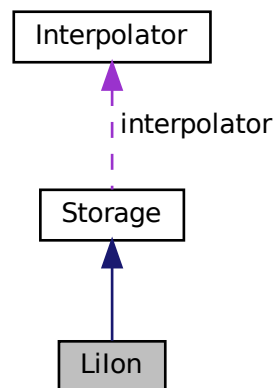
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

```
#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 [Lilon](#) 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 [Lilon](#) constructor.

Parameters

<i>liion_inputs</i>	A structure of Lilon 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 (liion_inputs.min_SOC < 0 or liion_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 (liion_inputs.hysteresis_SOC < 0 or liion_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 (liion_inputs.max_SOC < 0 or liion_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 (liion_inputs.charging_efficiency <= 0 or liion_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         liion_inputs.discharging_efficiency <= 0 or
140         liion_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 (liion_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 (liion_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 (liion_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 (lilion_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 (lilion_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 (lilion_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 (lilion_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 (lilion_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 (lilion_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 `LiIon`.

```

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 [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.
-------------------	--

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 [Liln](#).

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 [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:

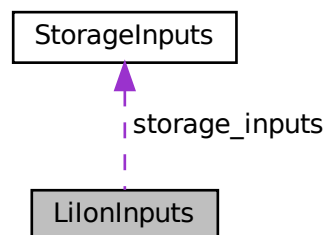
- header/Storage/[Lilon.h](#)
- source/Storage/[Lilon.cpp](#)

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:

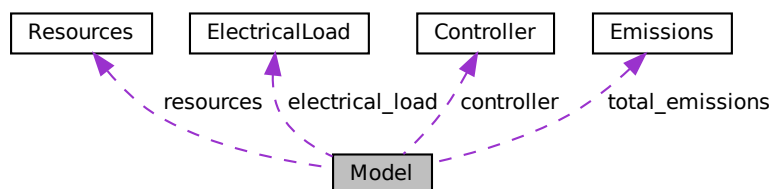
- header/Storage/[Lilon.h](#)

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.

```
632 {
633     return;
634 } /* 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.
---------------------	--

```
651 {
652     // 1. check inputs
653     this->__checkInputs(model_inputs);
654
655     // 2. read in electrical load data
656     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
657
658     // 3. set controller attributes
659     this->controller.setControlMode(model_inputs.control_mode);
660     this->controller.setLoadOperatingReserveFactor(model_inputs.load_operating_reserve_factor);
661     this->controller.setMaxOperatingReserveFactor(model_inputs.max_operating_reserve_factor);
662
663     // 4. set public attributes
664     this->total_fuel_consumed_L = 0;
665     this->net_present_cost = 0;
666     this->total_dispatch_discharge_kWh = 0;
667     this->total_renewable_dispatch_kWh = 0;
668     this->levellized_cost_of_energy_kWh = 0;
669
670     return;
671 } /* Model() */
```

4.15.2.3 ~Model()

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

Destructor for the [Model](#) class.

```
1190 {
1191     this->clear();
1192     return;
1193 } /* ~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

<code>model_inputs</code>	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(): ";
69         error_str += "ModelInputs::path_2_electrical_load_time_series 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     // 2. check load_operating_reserve_factor
79     if (
80         model_inputs.load_operating_reserve_factor < 0 or
81         model_inputs.load_operating_reserve_factor > 1
82     ) {
83         std::string error_str = "ERROR: Model(): ";
84         error_str += "ModelInputs::load_operating_reserve_factor must be in the closed interval [0, 1]";
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 max_operating_reserve_factor
94     if (
95         model_inputs.max_operating_reserve_factor < 0 or
96         model_inputs.max_operating_reserve_factor > 1
97     ) {
98         std::string error_str = "ERROR: Model(): ";
99         error_str += "ModelInputs::max_operating_reserve_factor must be in the closed interval [0, 1]";
100
101         #ifdef _WIN32
102             std::cout << error_str << std::endl;
103         #endif
104
105         throw std::invalid_argument(error_str);
106     }
107
108     return;
109 } /* __checkInputs() */
```

4.15.3.2 `__computeEconomics()`

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

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

```
295 {
296     this->__computeNetPresentCost();
297     this->__computeLevellizedCostOfEnergy();
298
299     return;
300 } /* __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.

```
125 {
126     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
127         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
128
129         this->total_fuel_consumed_L +=
130             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
131
132         this->total_emissions.CO2_kg +=
133             this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
134
135         this->total_emissions.CO_kg +=
136             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
137
138         this->total_emissions.NOx_kg +=
139             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
140
141         this->total_emissions.SOx_kg +=
142             this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
143
144         this->total_emissions.CH4_kg +=
145             this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
146
147         this->total_emissions.PM_kg +=
148             this->combustion_ptr_vec[i]->total_emissions.PM_kg;
149     }
150
151     return;
152 } /* __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.

```
242 {
243     // 1. account for Combustion economics in levellized cost of energy
244     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
245         this->levellized_cost_of_energy_kWh +=
246             (
247                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
248                 this->combustion_ptr_vec[i]->total_dispatch_kWh
249             ) / this->total_dispatch_discharge_kWh;
250     }
251
252     // 2. account for Noncombustion economics in levellized cost of energy
253     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
254         this->levellized_cost_of_energy_kWh +=
255             (
256                 this->noncombustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
257                 this->noncombustion_ptr_vec[i]->total_dispatch_kWh
258             ) / this->total_dispatch_discharge_kWh;
259     }
260
261     // 3. account for Renewable economics in levellized cost of energy
262     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
263         this->levellized_cost_of_energy_kWh +=
264             (
265                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
266                 this->renewable_ptr_vec[i]->total_dispatch_kWh
267             ) / this->total_dispatch_discharge_kWh;
268     }
269
270     // 4. account for Storage economics in levellized cost of energy
271     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
272         this->levellized_cost_of_energy_kWh +=
273             (
```

```

274         this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
275         this->storage_ptr_vec[i]->total_discharge_kWh
276     ) / this->total_dispatch_discharge_kWh;
277 }
278
279 return;
280 } /* __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.

```

169 {
170     // 1. account for Combustion economics in net present cost
171     // increment total dispatch
172     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
173         this->combustion_ptr_vec[i]->computeEconomics(
174             &(this->electrical_load.time_vec_hrs)
175         );
176
177         this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
178
179         this->total_dispatch_discharge_kWh +=
180             this->combustion_ptr_vec[i]->total_dispatch_kWh;
181     }
182
183     // 2. account for Noncombustion economics in net present cost
184     // increment total dispatch
185     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
186         this->noncombustion_ptr_vec[i]->computeEconomics(
187             &(this->electrical_load.time_vec_hrs)
188         );
189
190         this->net_present_cost += this->noncombustion_ptr_vec[i]->net_present_cost;
191
192         this->total_dispatch_discharge_kWh +=
193             this->noncombustion_ptr_vec[i]->total_dispatch_kWh;
194     }
195
196     // 3. account for Renewable economics in net present cost,
197     // increment total dispatch
198     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
199         this->renewable_ptr_vec[i]->computeEconomics(
200             &(this->electrical_load.time_vec_hrs)
201         );
202
203         this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
204
205         this->total_dispatch_discharge_kWh +=
206             this->renewable_ptr_vec[i]->total_dispatch_kWh;
207
208         this->total_renewable_dispatch_kWh +=
209             this->renewable_ptr_vec[i]->total_dispatch_kWh;
210     }
211
212     // 4. account for Storage economics in net present cost
213     // increment total dispatch
214     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
215         this->storage_ptr_vec[i]->computeEconomics(
216             &(this->electrical_load.time_vec_hrs)
217         );
218
219         this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
220
221         this->total_dispatch_discharge_kWh +=
222             this->storage_ptr_vec[i]->total_discharge_kWh;
223     }
224
225     return;
226 } /* __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.
-------------------	--

```
318 {
319     // 1. create subdirectory
320     write_path += "Model/";
321     std::filesystem::create_directory(write_path);
322
323     // 2. create filestream
324     write_path += "summary_results.md";
325     std::ofstream ofs;
326     ofs.open(write_path, std::ofstream::out);
327
328     // 3. write summary results (markdown)
329     ofs << "# Model Summary Results\n";
330     ofs << "\n-----\n\n";
331
332     // 3.1. ElectricalLoad
333     ofs << "## Electrical Load\n";
334     ofs << "\n";
335     ofs << "Path: " <<
336         this->electrical_load.path_2_electrical_load_time_series << " \n";
337     ofs << "Data Points: " << this->electrical_load.n_points << " \n";
338     ofs << "Years: " << this->electrical_load.n_years << " \n";
339     ofs << "Min: " << this->electrical_load.min_load_kW << " kW \n";
340     ofs << "Mean: " << this->electrical_load.mean_load_kW << " kW \n";
341     ofs << "Max: " << this->electrical_load.max_load_kW << " kW \n";
342     ofs << "\n-----\n\n";
343
344     // 3.2. Controller
345     ofs << "## Controller\n";
346     ofs << "\n";
347     ofs << "Control Mode: " << this->controller.control_string << " \n";
348     ofs << "Load Operating Reserve Factor: " <<
349         this->controller.load_operating_reserve_factor << " \n";
350     ofs << "Max Overall Operating Reserve Factor: " <<
351         this->controller.max_operating_reserve_factor << " \n";
352     ofs << "\n-----\n\n";
353
354     // 3.3. Resources (1D)
355     ofs << "## 1D Renewable Resources\n";
356     ofs << "\n";
357
358     std::map<int, std::string>::iterator string_map_1D_iter =
359         this->resources.string_map_1D.begin();
360     std::map<int, std::string>::iterator path_map_1D_iter =
361         this->resources.path_map_1D.begin();
362
363     while (
364         string_map_1D_iter != this->resources.string_map_1D.end() and
365         path_map_1D_iter != this->resources.path_map_1D.end()
366     ) {
367         ofs << "Resource Key: " << string_map_1D_iter->first << " \n";
368         ofs << "Type: " << string_map_1D_iter->second << " \n";
369         ofs << "Path: " << path_map_1D_iter->second << " \n";
370         ofs << "\n";
371
372         string_map_1D_iter++;
373         path_map_1D_iter++;
374     }
375
376     ofs << "\n-----\n\n";
377
378     // 3.4. Resources (2D)
379     ofs << "## 2D Renewable Resources\n";
380     ofs << "\n";
381
382     std::map<int, std::string>::iterator string_map_2D_iter =
383         this->resources.string_map_2D.begin();
384     std::map<int, std::string>::iterator path_map_2D_iter =
385         this->resources.path_map_2D.begin();
```

```

386
387 while (
388     string_map_2D_iter != this->resources.string_map_2D.end() and
389     path_map_2D_iter != this->resources.path_map_2D.end()
390 ) {
391     ofs << "Resource Key: " << string_map_2D_iter->first << " \n";
392     ofs << "Type: " << string_map_2D_iter->second << " \n";
393     ofs << "Path: " << path_map_2D_iter->second << " \n";
394     ofs << "\n";
395
396     string_map_2D_iter++;
397     path_map_2D_iter++;
398 }
399
400 ofs << "\n-----\n\n";
401
402 // 3.5. Combustion
403 ofs << "## Combustion Assets\n";
404 ofs << "\n";
405
406 for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
407     ofs << "Asset Index: " << i << " \n";
408     ofs << "Type: " << this->combustion_ptr_vec[i]->type_str << " \n";
409     ofs << "Capacity: " << this->combustion_ptr_vec[i]->capacity_kW << " kW \n";
410     ofs << "\n";
411 }
412
413 ofs << "\n-----\n\n";
414
415 // 3.6. Noncombustion
416 ofs << "## Noncombustion Assets\n";
417 ofs << "\n";
418
419 for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
420     ofs << "Asset Index: " << i << " \n";
421     ofs << "Type: " << this->noncombustion_ptr_vec[i]->type_str << " \n";
422     ofs << "Capacity: " << this->noncombustion_ptr_vec[i]->capacity_kW << " kW \n";
423
424     if (this->noncombustion_ptr_vec[i]->type == NoncombustionType :: HYDRO) {
425         ofs << "Reservoir Capacity: " <<
426             ((Hydro*) (this->noncombustion_ptr_vec[i]))->reservoir_capacity_m3 <<
427             " m3 \n";
428     }
429
430     ofs << "\n";
431 }
432
433 ofs << "\n-----\n\n";
434
435 // 3.7. Renewable
436 ofs << "## Renewable Assets\n";
437 ofs << "\n";
438
439 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
440     ofs << "Asset Index: " << i << " \n";
441     ofs << "Type: " << this->renewable_ptr_vec[i]->type_str << " \n";
442     ofs << "Capacity: " << this->renewable_ptr_vec[i]->capacity_kW << " kW \n";
443     ofs << "\n";
444 }
445
446 ofs << "\n-----\n\n";
447
448 // 3.8. Storage
449 ofs << "## Storage Assets\n";
450 ofs << "\n";
451
452 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
453     ofs << "Asset Index: " << i << " \n";
454     ofs << "Type: " << this->storage_ptr_vec[i]->type_str << " \n";
455     ofs << "Power Capacity: " << this->storage_ptr_vec[i]->power_capacity_kW
456         << " kW \n";
457     ofs << "Energy Capacity: " << this->storage_ptr_vec[i]->energy_capacity_kWh
458         << " kWh \n";
459     ofs << "\n";
460 }
461
462 ofs << "\n-----\n\n";
463
464 // 3.9. Model Results
465 ofs << "## Results\n";
466 ofs << "\n";
467
468 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
469 ofs << "\n";
470
471 ofs << "Total Dispatch + Discharge: " << this->total_dispatch_discharge_kWh
472     << " kWh \n";

```

```

473
474 ofs << "Renewable Penetration: "
475 << this->total_renewable_dispatch_kWh / this->total_dispatch_discharge_kWh
476 << " \n";
477 ofs << "\n";
478
479 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
480 << " per kWh dispatched/discharged \n";
481 ofs << "\n";
482
483 ofs << "Total Fuel Consumed: " << this->total_fuel_consumed_L << " L "
484 << "(Annual Average: " <<
485 this->total_fuel_consumed_L / this->electrical_load.n_years
486 << " L/yr) \n";
487 ofs << "\n";
488
489 ofs << "Total Carbon Dioxide (CO2) Emissions: " <<
490 this->total_emissions.CO2_kg << " kg "
491 << "(Annual Average: " <<
492 this->total_emissions.CO2_kg / this->electrical_load.n_years
493 << " kg/yr) \n";
494
495 ofs << "Total Carbon Monoxide (CO) Emissions: " <<
496 this->total_emissions.CO_kg << " kg "
497 << "(Annual Average: " <<
498 this->total_emissions.CO_kg / this->electrical_load.n_years
499 << " kg/yr) \n";
500
501 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
502 this->total_emissions.NOx_kg << " kg "
503 << "(Annual Average: " <<
504 this->total_emissions.NOx_kg / this->electrical_load.n_years
505 << " kg/yr) \n";
506
507 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
508 this->total_emissions.SOx_kg << " kg "
509 << "(Annual Average: " <<
510 this->total_emissions.SOx_kg / this->electrical_load.n_years
511 << " kg/yr) \n";
512
513 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
514 << "(Annual Average: " <<
515 this->total_emissions.CH4_kg / this->electrical_load.n_years
516 << " kg/yr) \n";
517
518 ofs << "Total Particulate Matter (PM) Emissions: " <<
519 this->total_emissions.PM_kg << " kg "
520 << "(Annual Average: " <<
521 this->total_emissions.PM_kg / this->electrical_load.n_years
522 << " kg/yr) \n";
523
524 ofs << "\n-----\n\n";
525
526 ofs.close();
527 return;
528 } /* __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.

```

548 {
549     // 1. create filestream

```

```

550     write_path += "Model/time_series_results.csv";
551     std::ofstream ofs;
552     ofs.open(write_path, std::ofstream::out);
553
554     // 2. write time series results header (comma separated value)
555     ofs << "Time (since start of data) [hrs],";
556     ofs << "Electrical Load [kW],";
557     ofs << "Net Load [kW],";
558     ofs << "Missed Load [kW],";
559
560     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
561         ofs << this->renewable_ptr_vec[i]->capacity_kW << " kW "
562             << this->renewable_ptr_vec[i]->type_str << " Dispatch [kW],";
563     }
564
565     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
566         ofs << this->storage_ptr_vec[i]->power_capacity_kW << " kW "
567             << this->storage_ptr_vec[i]->energy_capacity_kWh << " kWh "
568             << this->storage_ptr_vec[i]->type_str << " Discharge [kW],";
569     }
570
571     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
572         ofs << this->noncombustion_ptr_vec[i]->capacity_kW << " kW "
573             << this->noncombustion_ptr_vec[i]->type_str << " Dispatch [kW],";
574     }
575
576     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
577         ofs << this->combustion_ptr_vec[i]->capacity_kW << " kW "
578             << this->combustion_ptr_vec[i]->type_str << " Dispatch [kW],";
579     }
580
581     ofs << "\n";
582
583     // 3. write time series results values (comma separated value)
584     for (int i = 0; i < max_lines; i++) {
585         // 3.1. load values
586         ofs << this->electrical_load.time_vec_hrs[i] << ",";
587         ofs << this->electrical_load.load_vec_kW[i] << ",";
588         ofs << this->controller.net_load_vec_kW[i] << ",";
589         ofs << this->controller.missed_load_vec_kW[i] << ",";
590
591         // 3.2. asset-wise dispatch/discharge
592         for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
593             ofs << this->renewable_ptr_vec[j]->dispatch_vec_kW[i] << ",";
594         }
595
596         for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
597             ofs << this->storage_ptr_vec[j]->discharging_power_vec_kW[i] << ",";
598         }
599
600         for (size_t j = 0; j < this->noncombustion_ptr_vec.size(); j++) {
601             ofs << this->noncombustion_ptr_vec[j]->dispatch_vec_kW[i] << ",";
602         }
603
604         for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
605             ofs << this->combustion_ptr_vec[j]->dispatch_vec_kW[i] << ",";
606         }
607
608         ofs << "\n";
609     }
610
611     ofs.close();
612     return;
613 } /* __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.
----------------------	---

```

688 {
689     Combustion* diesel_ptr = new Diesel(
690         this->electrical_load.n_points,
691         this->electrical_load.n_years,
692         diesel_inputs,
693         &(this->electrical_load.time_vec_hrs)
694     );
695
696     this->combustion_ptr_vec.push_back(diesel_ptr);
697
698     return;
699 } /* 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.
---------------------	--

```

792 {
793     Noncombustion* hydro_ptr = new Hydro(
794         this->electrical_load.n_points,
795         this->electrical_load.n_years,
796         hydro_inputs,
797         &(this->electrical_load.time_vec_hrs)
798     );
799
800     this->noncombustion_ptr_vec.push_back(hydro_ptr);
801
802     return;
803 } /* addHydro() */

```

4.15.3.10 addLiIon()

```

void Model::addLiIon (
    LiIonInputs liion_inputs )

```

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

Parameters

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

```

932 {
933     Storage* liion_ptr = new LiIon(
934         this->electrical_load.n_points,
935         this->electrical_load.n_years,
936         liion_inputs
937     );
938
939     this->storage_ptr_vec.push_back(liion_ptr);
940
941     return;
942 } /* 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.

```
728 {
729     resources.addResource(
730         noncombustion_type,
731         path_2_resource_data,
732         resource_key,
733         &(this->electrical_load)
734     );
735
736     return;
737 } /* 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.

```
766 {
767     resources.addResource(
768         renewable_type,
769         path_2_resource_data,
770         resource_key,
771         &(this->electrical_load)
772     );
773
774     return;
775 } /* 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.
---------------------	--

```
820 {
821     Renewable* solar_ptr = new Solar(
822         this->electrical_load.n_points,
823         this->electrical_load.n_years,
824         solar_inputs,
825         &(this->electrical_load.time_vec_hrs)
826     );
827
828     this->renewable_ptr_vec.push_back(solar_ptr);
829
830     return;
831 } /* 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.
---------------------	--

```
848 {
849     Renewable* tidal_ptr = new Tidal(
850         this->electrical_load.n_points,
851         this->electrical_load.n_years,
852         tidal_inputs,
853         &(this->electrical_load.time_vec_hrs)
854     );
855
856     this->renewable_ptr_vec.push_back(tidal_ptr);
857
858     return;
859 } /* 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.
--------------------	---

```

876 {
877     Renewable* wave_ptr = new Wave(
878         this->electrical_load.n_points,
879         this->electrical_load.n_years,
880         wave_inputs,
881         &(this->electrical_load.time_vec_hrs)
882     );
883
884     this->renewable_ptr_vec.push_back(wave_ptr);
885
886     return;
887 } /* addWave() */

```

4.15.3.16 addWind()

```

void Model::addWind (
    WindInputs wind_inputs )

```

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

Parameters

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

```

904 {
905     Renewable* wind_ptr = new Wind(
906         this->electrical_load.n_points,
907         this->electrical_load.n_years,
908         wind_inputs,
909         &(this->electrical_load.time_vec_hrs)
910     );
911
912     this->renewable_ptr_vec.push_back(wind_ptr);
913
914     return;
915 } /* addWind() */

```

4.15.3.17 clear()

```

void Model::clear (
    void )

```

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

```

1059 {
1060     // 1. reset
1061     this->reset();
1062
1063     // 2. clear components
1064     controller.clear();
1065     electrical_load.clear();
1066     resources.clear();
1067
1068     return;
1069 } /* 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.

```
1001 {
1002     // 1. clear combustion_ptr_vec
1003     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
1004         delete this->combustion_ptr_vec[i];
1005     }
1006     this->combustion_ptr_vec.clear();
1007
1008     // 2. clear noncombustion_ptr_vec
1009     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
1010         delete this->noncombustion_ptr_vec[i];
1011     }
1012     this->noncombustion_ptr_vec.clear();
1013
1014     // 3. clear renewable_ptr_vec
1015     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
1016         delete this->renewable_ptr_vec[i];
1017     }
1018     this->renewable_ptr_vec.clear();
1019
1020     // 4. clear storage_ptr_vec
1021     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
1022         delete this->storage_ptr_vec[i];
1023     }
1024     this->storage_ptr_vec.clear();
1025
1026     // 5. reset components and attributes
1027     this->controller.clear();
1028
1029     this->total_fuel_consumed_L = 0;
1030
1031     this->total_emissions.CO2_kg = 0;
1032     this->total_emissions.CO_kg = 0;
1033     this->total_emissions.NOx_kg = 0;
1034     this->total_emissions.SOx_kg = 0;
1035     this->total_emissions.CH4_kg = 0;
1036     this->total_emissions.PM_kg = 0;
1037
1038     this->net_present_cost = 0;
1039     this->total_dispatch_discharge_kWh = 0;
1040     this->total_renewable_dispatch_kWh = 0;
1041     this->levelized_cost_of_energy_kWh = 0;
1042
1043     return;
1044 } /* reset() */
```

4.15.3.19 run()

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

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

```
957 {
958     // 1. init Controller
959     this->controller.init(
960         &(this->electrical_load),
961         &(this->renewable_ptr_vec),
962         &(this->resources),
963         &(this->combustion_ptr_vec)
964     );
965
966     // 2. apply dispatch control
967     this->controller.applyDispatchControl(
968         &(this->electrical_load),
969         &(this->resources),
970         &(this->combustion_ptr_vec),
971         &(this->noncombustion_ptr_vec),
```

```

972         &(this->renewable_ptr_vec),
973         &(this->storage_ptr_vec)
974     );
975
976     // 3. compute total fuel consumption and emissions
977     this->__computeFuelAndEmissions();
978
979     // 4. compute key economic metrics
980     this->__computeEconomics();
981
982     return;
983 } /* 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.

```

1097 {
1098     // 1. handle sentinel
1099     if (max_lines < 0) {
1100         max_lines = this->electrical_load.n_points;
1101     }
1102
1103     // 2. check for pre-existing, warn (and remove), then create
1104     if (write_path.back() != '/') {
1105         write_path += '/';
1106     }
1107
1108     if (std::filesystem::is_directory(write_path)) {
1109         std::string warning_str = "WARNING: Model::writeResults(): ";
1110         warning_str += write_path;
1111         warning_str += " already exists, contents will be overwritten!";
1112
1113         std::cout << warning_str << std::endl;
1114
1115         std::filesystem::remove_all(write_path);
1116     }
1117
1118     std::filesystem::create_directory(write_path);
1119
1120     // 3. write summary
1121     this->__writeSummary(write_path);
1122
1123     // 4. write time series
1124     if (max_lines > this->electrical_load.n_points) {
1125         max_lines = this->electrical_load.n_points;
1126     }
1127
1128     if (max_lines > 0) {
1129         this->__writeTimeSeries(write_path, max_lines);
1130     }
1131
1132     // 5. call out to Combustion :: writeResults()
1133     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
1134         this->combustion_ptr_vec[i]->writeResults(
1135             write_path,
1136             &(this->electrical_load.time_vec_hrs),
1137             i,
1138             max_lines
1139         );
1140     }

```

```

1141
1142 // 6. call out to Noncombustion :: writeResults()
1143 for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
1144     this->noncombustion_ptr_vec[i]->writeResults(
1145         write_path,
1146         &(this->electrical_load.time_vec_hrs),
1147         i,
1148         max_lines
1149     );
1150 }
1151
1152 // 7. call out to Renewable :: writeResults()
1153 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
1154     this->renewable_ptr_vec[i]->writeResults(
1155         write_path,
1156         &(this->electrical_load.time_vec_hrs),
1157         &(this->resources.resource_map_1D),
1158         &(this->resources.resource_map_2D),
1159         i,
1160         max_lines
1161     );
1162 }
1163
1164 // 8. call out to Storage :: writeResults()
1165 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
1166     this->storage_ptr_vec[i]->writeResults(
1167         write_path,
1168         &(this->electrical_load.time_vec_hrs),
1169         i,
1170         max_lines
1171     );
1172 }
1173
1174 return;
1175 } /* 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.
- `double load_operating_reserve_factor = 0.2`
An operating reserve factor [0, 1] to cover random fluctuations in load.
- `double max_operating_reserve_factor = 1`
*A maximum reserve factor [0, 1] that limits the required overall operating reserve to, at most, factor * load_kW.*

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 load_operating_reserve_factor

```
double ModelInputs::load_operating_reserve_factor = 0.2
```

An operating reserve factor [0, 1] to cover random fluctuations in load.

4.16.2.3 max_operating_reserve_factor

```
double ModelInputs::max_operating_reserve_factor = 1
```

A maximum reserve factor [0, 1] that limits the required overall operating reserve to, at most, factor * load_kW.

4.16.2.4 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:

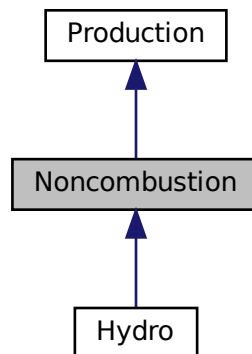
- header/[Model.h](#)

4.17 Noncombustion Class Reference

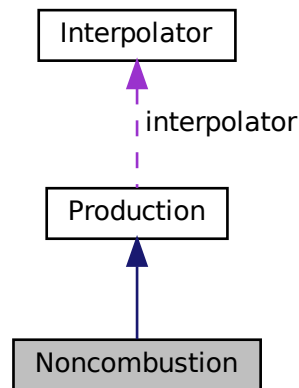
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

```
#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:

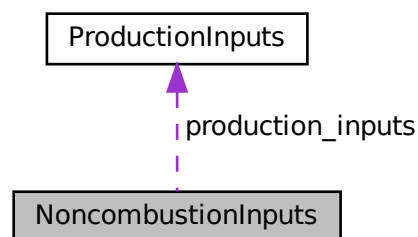
- [header/Production/Noncombustion/Noncombustion.h](#)
- [source/Production/Noncombustion/Noncombustion.cpp](#)

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:

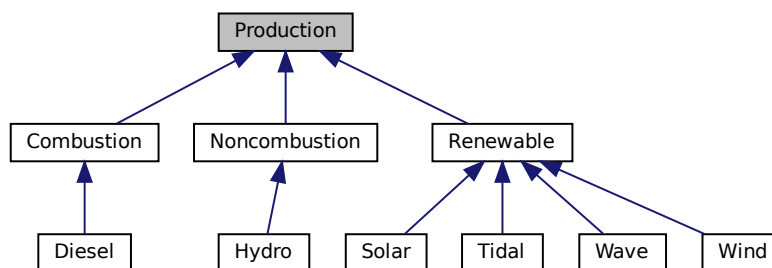
- `header/Production/Noncombustion/Noncombustion.h`

4.19 Production Class Reference

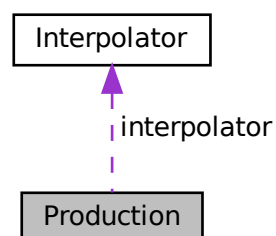
The base class of the `Production` hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

```
#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::levellized_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:

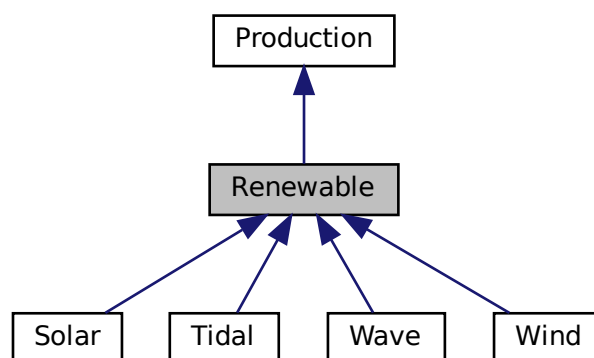
- header/Production/[Production.h](#)

4.21 Renewable Class Reference

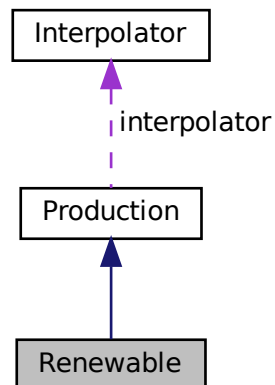
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

```
#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.
- double [firmness_factor](#)
A factor [0, 1] which defines how firm the production from this asset is.

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](#).

```
123 {return 0;}
```

4.21.3.8 computeProductionkW() [2/2]

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

Reimplemented in [Wave](#).

```
124 {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 <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>resource_map_1D_ptr</i>	A pointer to the 1D map of Resources .
<i>resource_map_2D_ptr</i>	A pointer to the 2D map of Resources .
<i>renewable_index</i>	An integer which corresponds to the index of the Renewable asset in the Model .
<i>max_lines</i>	The maximum number of lines of output to write. If <0 , then all available lines are written. If $=0$, then only summary results are written.

```

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 firmness_factor

```
double Renewable::firmness_factor
```

A factor [0, 1] which defines how firm the production from this asset is.

4.21.4.2 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.

4.21.4.3 type

```
RenewableType Renewable::type
```

The type ([RenewableType](#)) of the asset.

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

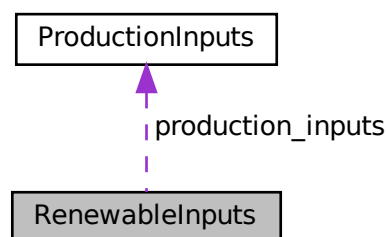
- header/Production/Renewable/[Renewable.h](#)
- source/Production/Renewable/[Renewable.cpp](#)

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 $\langle \text{int}, \text{vector}\langle \text{double} \rangle \rangle$ of given 1D renewable resource time series.
- `std::map< int, std::string >` [string_map_1D](#)
A map $\langle \text{int}, \text{string} \rangle$ of descriptors for the type of the given 1D renewable resource time series.
- `std::map< int, std::string >` [path_map_1D](#)
A map $\langle \text{int}, \text{string} \rangle$ 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 $\langle \text{int}, \text{vector}\langle \text{vector}\langle \text{double} \rangle \rangle \rangle$ of given 2D renewable resource time series.
- `std::map< int, std::string >` [string_map_2D](#)
A map $\langle \text{int}, \text{string} \rangle$ of descriptors for the type of the given 2D renewable resource time series.
- `std::map< int, std::string >` [path_map_2D](#)
A map $\langle \text{int}, \text{string} \rangle$ 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
454     // 3. read in resource data, check against time series (point-wise and length)
455     int n_points = 0;
456     double time_hrs = 0;
457     double time_expected_hrs = 0;
458     double solar_resource_kWm2 = 0;
459
460     while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
461         if (n_points > electrical_load_ptr->n_points) {
462             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
463         }
464
465         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
466         this->__checkTimePoint(
467             time_hrs,
468             time_expected_hrs,
469             path_2_resource_data,
470             electrical_load_ptr
471         );

```

```

472
473     this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
474
475     n_points++;
476 }
477
478 // 4. check data length
479 if (n_points != electrical_load_ptr->n_points) {
480     this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
481 }
482
483 return;
484 } /* __readSolarResource() */

```

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     }

```

```

557
558     n_points++;
559 }
560
561 // 4. check data length
562 if (n_points != electrical_load_ptr->n_points) {
563     this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
564 }
565
566 return;
567 } /* __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->__checkResourceKeyID(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->__checkResourceKeyID(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:

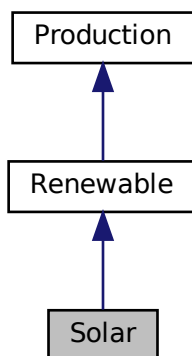
- header/[Resources.h](#)
- source/[Resources.cpp](#)

4.24 Solar Class Reference

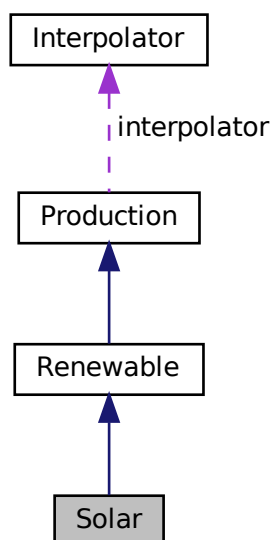
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

```
#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.).
- double [julian_day](#)
The number of days (including partial days) since 12:00 on 1 Jan 2000.
- double [latitude_deg](#)
The latitude of the solar PV array [deg].
- double [longitude_deg](#)
The longitude of the solar PV array [deg].
- double [latitude_rad](#)
The latitude of the solar PV array [rad].
- double [longitude_rad](#)
The longitude of the solar PV array [rad].
- double [panel_azimuth_deg](#)
The azimuth angle of the panels [deg], relative to north.
- double [panel_tilt_deg](#)
The tilt angle of the panels [deg], relative to ground.
- double [panel_azimuth_rad](#)
The azimuth angle of the panels [rad], relative to north.
- double [panel_tilt_rad](#)
The tilt angle of the panels [rad], relative to ground.
- double [albedo_ground_reflectance](#)
The albedo (ground reflectance) to be applied in modelling the solar PV array.
- [SolarPowerProductionModel](#) [power_model](#)
The solar power production model to be applied.
- std::string [power_model_string](#)
A string describing the active power production model.

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.
- double [__getMeanLongitudeDeg](#) (void)
Method to compute and return the mean longitude [deg], bound to the half-open interval [0, 360). From eqn (4.7) of Gilman.
- double [__getMeanAnomalyRad](#) (void)
Method to compute and return the mean anomaly [rad], bound to the half-open interval [0, 2pi). From eqn (4.8) of Gilman.
- double [__getEclipticLongitudeRad](#) (double, double)
Method to compute and return the ecliptic longitude [rad], bound to the half-open interval [0, 2pi). From eqn (4.9) of Gilman.
- double [__getObliquityOfEclipticRad](#) (void)
Method to compute and return the obliquity of the ecliptic [rad], bound to the half-open interval [0, 2pi). From eqn (4.10) of Gilman.
- double [__getGreenwichMeanSiderialTimeHrs](#) (void)
Method to compute the Greenwich mean siderial time [hrs], bound to the half-open interval [0, 24) hrs. From eqn (4.13) of Gilman.
- double [__getLocalMeanSiderialTimeHrs](#) (double)
Method to compute and return the local mean siderial time [hrs], bound to the half-open interval [0, 24) hrs. From eqn (4.14) of Gilman.
- double [__getRightAscensionRad](#) (double, double)
Method to compute and return the right ascension of the sun [rad], bound to the half-open interval [0, 2pi). From eqn (4.11) of Gilman.
- double [__getDeclinationRad](#) (double, double)
Method to compute and return the declination of the sun [rad], bound to the closed interval [-pi/2, pi/2]. From eqn (4.12) of Gilman.
- double [__getHourAngleRad](#) (double, double)
Method to compute and return the hour angle [rad] of the sun, bound to the open interval (-pi, pi). From eqn (4.15) of Gilman.
- double [__getSolarAltitudeRad](#) (double, double)
Method to compute and return the sun altitude [rad], corrected for refraction and bound to the closed interval [0, pi/2]. From eqns (4.16) and (4.17) of Gilman.
- double [__getSolarAzimuthRad](#) (double, double)
Method to compute and return the solar azimuth [rad], bound to the closed interval [-pi, pi]. From eqns (4.16) and (4.18) of Gilman.
- double [__getSolarZenithRad](#) (double, double)
Method to compute and return the solar zenith [rad], bound to the open interval (-pi/2, pi/2). From eqn (4.19) of Gilman.
- double [__getDiffuseHorizontalIrradiancekWm2](#) (double)
Method which takes in the solar resource at a particular point in time, and then returns the diffuse horizontal irradiance (DHI) [kW/m²] using a very simple, empirical model (simply DHI is proportional to GHI).
- double [__getDirectNormalIrradiancekWm2](#) (double, double, double)
Method which takes in the solar resource and DHI at a particular point in time, then the returns the direct normal irradiance (DNI) [kW/m²]. From definition of global horizontal irradiance (GHI).
- double [__getAngleOfIncidenceRad](#) (double, double)
Method to compute and return the angle of incidence [rad] between the solar beam and the panel normal. From eqn (5.1) of Gilman.

- double [__getBeamIrradiancekWm2](#) (double, double)
Method which computes and returns the beam irradiance normal to the panels [kW/m2]. From eqn (6.1) of Gilman.
- double [__getDiffuseIrradiancekWm2](#) (double)
Method which computes and returns the (isotropic) diffuse sky irradiance [kW/m2]. From eqn (6.5) of Gilman.
- double [__getGroundReflectedIrradiancekWm2](#) (double)
Method to compute and return the ground reflected irradiance [kW/m2]. From eqn (6.21) of Gilman.
- double [__getPlaneOfArrayIrradiancekWm2](#) (int, double, double)
Method which takes in the solar resource at a particular point in time, and then returns the nominal plane of array irradiance. From eqn (7.1) of Gilman.
- double [__computeSimpleProductionkW](#) (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 base on a simple, "HOMER-like" model.
- double [__computeDetailedProductionkW](#) (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 base on a detailed, "PVWatts/SAM-like" model.
- 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.

```
1404 {
1405     //...
1406
1407     return;
1408 } /* 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.

```

1440 :
1441 Renewable(
1442     n_points,
1443     n_years,
1444     solar_inputs.renewable_inputs,
1445     time_vec_hrs_ptr
1446 )
1447 {
1448     // 1. check inputs
1449     this->__checkInputs(solar_inputs);
1450
1451     // 2. set attributes
1452     this->type = RenewableType :: SOLAR;
1453     this->type_str = "SOLAR";
1454
1455     this->resource_key = solar_inputs.resource_key;
1456
1457     this->firmness_factor = solar_inputs.firmness_factor;
1458
1459     this->derating = solar_inputs.derating;
1460
1461     this->julian_day = solar_inputs.julian_day;
1462
1463     this->latitude_deg = solar_inputs.latitude_deg;
1464     this->longitude_deg = solar_inputs.longitude_deg;
1465
1466     this->latitude_rad = (M_PI / 180.0) * this->latitude_deg;
1467     this->longitude_rad = (M_PI / 180.0) * this->longitude_deg;
1468
1469     this->panel_azimuth_deg = solar_inputs.panel_azimuth_deg;
1470     this->panel_tilt_deg = solar_inputs.panel_tilt_deg;
1471
1472     this->panel_azimuth_rad = (M_PI / 180.0) * this->panel_azimuth_deg;
1473     this->panel_tilt_rad = (M_PI / 180.0) * this->panel_tilt_deg;
1474
1475     this->albedo_ground_reflectance = solar_inputs.albedo_ground_reflectance;
1476
1477     this->power_model = solar_inputs.power_model;
1478
1479     switch (this->power_model) {
1480         case (SolarPowerProductionModel :: SOLAR_POWER_SIMPLE): {
1481             this->power_model_string = "SIMPLE";
1482
1483             break;
1484         }
1485
1486         case (SolarPowerProductionModel :: SOLAR_POWER_DETAILED): {
1487             this->power_model_string = "DETAILED";
1488
1489             break;
1490         }
1491
1492         default: {
1493             std::string error_str = "ERROR: Solar(): ";
1494             error_str += "power production model ";
1495             error_str += std::to_string(this->power_model);
1496             error_str += " not recognized";
1497
1498             #ifdef _WIN32
1499                 std::cout << error_str << std::endl;
1500             #endif
1501
1502             throw std::runtime_error(error_str);
1503
1504             break;
1505         }
1506     }
1507
1508     if (solar_inputs.capital_cost < 0) {
1509         this->capital_cost = this->__getGenericCapitalCost();
1510     }
1511     else {
1512         this->capital_cost = solar_inputs.capital_cost;
1513     }
1514
1515     if (solar_inputs.operation_maintenance_cost_kWh < 0) {

```



```

1516         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
1517     }
1518     else {
1519         this->operation_maintenance_cost_kWh =
1520             solar_inputs.operation_maintenance_cost_kWh;
1521     }
1522
1523     if (not this->is_sunk) {
1524         this->capital_cost_vec[0] = this->capital_cost;
1525     }
1526
1527     // 3. construction print
1528     if (this->print_flag) {
1529         std::cout << "Solar object constructed at " << this << std::endl;
1530     }
1531
1532     return;
1533 } /* Renewable() */

```

4.24.2.3 ~Solar()

```

Solar::~~Solar (
    void )

```

Destructor for the [Solar](#) class.

```

1710 {
1711     // 1. destruction print
1712     if (this->print_flag) {
1713         std::cout << "Solar object at " << this << " destroyed" << std::endl;
1714     }
1715
1716     return;
1717 } /* ~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     // 2. check julian day
79     if (solar_inputs.julian_day < 0) {
80         std::string error_str = "ERROR: Solar(): ";
81         error_str += "SolarInputs::julian_day must be >= 0 days.";
82
83         #ifdef _WIN32
84             std::cout << error_str << std::endl;

```

```

85         #endif
86
87         throw std::invalid_argument(error_str);
88     }
89
90     // 3. check latitude
91     if (
92         solar_inputs.latitude_deg < -90 or
93         solar_inputs.latitude_deg > 90
94     ) {
95         std::string error_str = "ERROR: Solar(): ";
96         error_str += "SolarInputs::latitude_deg must be in the closed interval ";
97         error_str += "[-90, 90] degrees";
98
99         #ifdef _WIN32
100             std::cout << error_str << std::endl;
101         #endif
102
103         throw std::invalid_argument(error_str);
104     }
105
106     // 4. check longitude
107     if (
108         solar_inputs.longitude_deg < -180 or
109         solar_inputs.longitude_deg > 180
110     ) {
111         std::string error_str = "ERROR: Solar(): ";
112         error_str += "SolarInputs::longitude_deg must be in the closed interval ";
113         error_str += "[-180, 180] degrees";
114
115         #ifdef _WIN32
116             std::cout << error_str << std::endl;
117         #endif
118
119         throw std::invalid_argument(error_str);
120     }
121
122     // 5. check panel tilt angle
123     if (
124         solar_inputs.panel_tilt_deg < 0 or
125         solar_inputs.panel_tilt_deg > 90
126     ) {
127         std::string error_str = "ERROR: Solar(): ";
128         error_str += "SolarInputs::panel_tilt_deg must be in the closed interval ";
129         error_str += "[0, 90] degrees";
130
131         #ifdef _WIN32
132             std::cout << error_str << std::endl;
133         #endif
134
135         throw std::invalid_argument(error_str);
136     }
137
138     // 6. check albedo ground reflectance
139     if (
140         solar_inputs.albedo_ground_reflectance < 0 or
141         solar_inputs.albedo_ground_reflectance > 1
142     ) {
143         std::string error_str = "ERROR: Solar(): ";
144         error_str += "SolarInputs::albedo_ground_reflectance must be in the closed ";
145         error_str += "interval [0, 1]";
146
147         #ifdef _WIN32
148             std::cout << error_str << std::endl;
149         #endif
150
151         throw std::invalid_argument(error_str);
152     }
153
154     // 7. check firmness_factor
155     if (
156         solar_inputs.firmness_factor < 0 or
157         solar_inputs.firmness_factor > 1
158     ) {
159         std::string error_str = "ERROR: Solar(): ";
160         error_str += "SolarInputs::firmness_factor must be in the closed interval [0, 1]";
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 } /* __checkInputs() */

```

4.24.3.2 `__computeDetailedProductionkW()`

```
double Solar::__computeDetailedProductionkW (
    int timestep,
    double dt_hrs,
    double solar_resource_kWm2 ) [private]
```

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 base on a detailed, "PVWatts/SAM-like" model.

Ref: [Gilman et al. \[2018\]](#)

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. global horizontal irradiance) [kW/m2].

Returns

The production [kW] of the solar PV array.

```
1194 {
1195     // apply detailed production model (POA irradiance -> production)
1196     double plane_of_array_irradiance_kWm2 = this->__getPlaneOfArrayIrradiancekWm2 (
1197         timestep,
1198         dt_hrs,
1199         solar_resource_kWm2
1200     );
1201
1202     double production_kW =
1203         this->derating * plane_of_array_irradiance_kWm2 * this->capacity_kW;
1204
1205     // cap production at capacity
1206     if (production_kW > this->capacity_kW) {
1207         production_kW = this->capacity_kW;
1208     }
1209
1210     return production_kW;
1211 } /* __computeDetailedProductionkW() */
```

4.24.3.3 `__computeSimpleProductionkW()`

```
double Solar::__computeSimpleProductionkW (
    int timestep,
    double dt_hrs,
    double solar_resource_kWm2 ) [private]
```

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 base on a simple, "HOMER-like" model.

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. global horizontal irradiance) [kW/m2].

Returns

The production [kW] of the solar PV array.

```

1149 {
1150     // apply simple production model (GHI -> production)
1151     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
1152
1153     // cap production at capacity
1154     if (production_kW > this->capacity_kW) {
1155         production_kW = this->capacity_kW;
1156     }
1157
1158     return production_kW;
1159 } /* __computeSimpleProductionkW() */

```

4.24.3.4 __getAngleOfIncidenceRad()

```

double Solar::__getAngleOfIncidenceRad (
    double solar_zenith_rad,
    double solar_azimuth_rad ) [private]

```

Method to compute and return the angle of incidence [rad] between the solar beam and the panel normal. From eqn (5.1) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>solar_zenith_rad</i>	The solar zenith [rad].
<i>solar_azimuth_rad</i>	The solar azimuth [rad].

Returns

The angle of incidence [rad] between the solar beam and the panel normal.

```

884 {
885     double a =
886         sin(solar_zenith_rad) *
887         cos(solar_azimuth_rad - this->panel_azimuth_rad) *
888         sin(this->panel_tilt_rad) +
889         cos(solar_zenith_rad) *
890         cos(this->panel_tilt_rad);
891
892     double angle_of_incidence_rad = 0;
893
894     if (a < -1) {
895         angle_of_incidence_rad = M_PI;
896     }
897
898     else if (a > 1) {
899         angle_of_incidence_rad = 0;
900     }
901 }

```

```

902     else {
903         angle_of_incidence_rad = acos(a);
904     }
905
906     return angle_of_incidence_rad;
907 } /* __getAngleOfIncidenceRad() */

```

4.24.3.5 __getBeamIrradiancekWm2()

```

double Solar::__getBeamIrradiancekWm2 (
    double direct_normal_irradiance_kWm2,
    double angle_of_incidence_rad ) [private]

```

Method which computes and returns the beam irradiance normal to the panels [kW/m²]. From eqn (6.1) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>direct_normal_irradiance_kWm2</i>	The DNI [kW/m ²].
<i>angle_of_incidence_rad</i>	The angle of incidence [rad] between the solar beam and the panel normal.

Returns

The beam irradiance normal to the panels [kW/m²].

```

938 {
939     double beam_irradiance_kWm2 = direct_normal_irradiance_kWm2 *
940         cos(angle_of_incidence_rad);
941
942     return beam_irradiance_kWm2;
943 } /* __getBeamIrradiancekWm2() */

```

4.24.3.6 __getDeclinationRad()

```

double Solar::__getDeclinationRad (
    double eclong_rad,
    double obleq_rad ) [private]

```

Method to compute and return the declination of the sun [rad], bound to the closed interval $[-\pi/2, \pi/2]$. From eqn (4.12) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>eclong_rad</i>	The ecliptic longitude [rad], bound to the half-open interval $[0, 2\pi)$.
<i>obleq_rad</i>	The obliquity of the ecliptic, bound to the half-open interval $[0, 2\pi)$.

Returns

The declination of the sun [rad], bound to the closed interval $[-\pi/2, \pi/2]$.

```

483 {
484     double declination_rad = asin(sin(obleq_rad) * sin(eclong_rad));
485
486     return declination_rad;
487 } /* __getDeclinationRad() */

```

4.24.3.7 __getDiffuseHorizontalIrradiancekWm2()

```

double Solar::__getDiffuseHorizontalIrradiancekWm2 (
    double solar_resource_kWm2 ) [private]

```

Method which takes in the solar resource at a particular point in time, and then returns the diffuse horizontal irradiance (DHI) [kW/m²] using a very simple, empirical model (simply DHI is proportional to GHI).

Ref: [Safaripour and Mehrabian \[2011\]](#)

Parameters

<i>solar_resource_kWm2</i>	Solar resource (i.e. global horizontal irradiance) [kW/m ²].
----------------------------	--

Returns

The diffuse horizontal irradiance [kW/m²].

```

809 {
810     double GHI_2_DHI = 0.32;
811
812     return GHI_2_DHI * solar_resource_kWm2;
813 } /* __getDiffuseHorizontalIrradiancekWm2() */

```

4.24.3.8 __getDiffuseIrradiancekWm2()

```

double Solar::__getDiffuseIrradiancekWm2 (
    double diffuse_horizontal_irradiance_kWm2 ) [private]

```

Method which computes and returns the (isotropic) diffuse sky irradiance [kW/m²]. From eqn (6.5) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>diffuse_horizontal_irradiance_kWm2</i>	The DHI [kW/m ²]
---	------------------------------

Returns

The (isotropic) diffuse sky irradiance [kW/m²]

```

965 {
966     double diffuse_sky_irradiance_kWm2 = diffuse_horizontal_irradiance_kWm2 *
967         cos(this->panel_tilt_rad);
968
969     return diffuse_sky_irradiance_kWm2;
970 } /* __getDiffuseIrradiancekWm2() */

```

4.24.3.9 __getDirectNormalIrradiancekWm2()

```

double Solar::__getDirectNormalIrradiancekWm2 (
    double solar_resource_kWm2,
    double diffuse_horizontal_irradiance_kWm2,
    double solar_zenith_rad ) [private]

```

Method which takes in the solar resource and DHI at a particular point in time, then the returns the direct normal irradiance (DNI) [kW/m²]. From definition of global horizontal irradiance (GHI).

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>solar_resource_kWm2</i>	Solar resource (i.e. global horizontal irradiance) [kW/m ²].
<i>diffuse_horizontal_irradiance_kWm2</i>	The DHI [kW/m ²].
<i>solar_zenith_rad</i>	The solar zenith [rad].

Returns

The direct normal irradiance (DNI) [kW/m²].

```

848 {
849     double direct_normal_irradiance_kWm2 =
850         (solar_resource_kWm2 - diffuse_horizontal_irradiance_kWm2) /
851         cos(solar_zenith_rad);
852
853     return direct_normal_irradiance_kWm2;
854 } /* __getDirectNormalIrradiancekWm2() */

```

4.24.3.10 __getEclipticLongitudeRad()

```

double Solar::__getEclipticLongitudeRad (
    double mean_longitude_deg,
    double mean_anomaly_rad ) [private]

```

Method to compute and return the ecliptic longitude [rad], bound to the half-open interval [0, 2pi). From eqn (4.9) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>mean_longitude_deg</i>	The mean longitude [deg], bound to the half-open interval [0, 360) deg.
<i>mean_anomaly_rad</i>	The mean anomaly [rad], bound to the half-open interval [0, 2pi).

Returns

The ecliptic longitude [rad], bound to the half-open interval [0, 2pi).

```

321 {
322     // compute ecliptic longitude
323     double eclong_deg = mean_longitude_deg +
324         1.915 * sin(mean_anomaly_rad) +
325         0.02 * sin(2 * mean_anomaly_rad);
326
327     // bound to half-open interval [0, 360) deg
328     int eclong_deg_int = int(eclong_deg);
329     double eclong_deg_frac = eclong_deg - eclong_deg_int;
330
331     eclong_deg = eclong_deg_int % 360;
332     eclong_deg += eclong_deg_frac;
333
334     // translate to rads
335     double eclong_rad = (M_PI / 180.0) * eclong_deg;
336
337     return eclong_rad;
338 } /* __getEclipticLongitudeRad() */

```

4.24.3.11 __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].

```

192 {
193     double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;
194
195     return capital_cost_per_kW * this->capacity_kW;
196 } /* __getGenericCapitalCost() */

```

4.24.3.12 __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].

```

219 {
220     return 0.01;
221 } /* __getGenericOpMaintCost() */

```


4.24.3.13 `__getGreenwichMeanSiderialTimeHrs()`

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

Method to compute the Greenwich mean siderial time [hrs], bound to the half-open interval [0, 24) hrs. From eqn (4.13) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Returns

Greenwich mean siderial time [hrs], bound to the half-open interval [0, 24) hrs.

```
394 {
395     // compute Greenwich mean siderial time
396     double Greenwich_mean_siderial_time_hrs = 6.697375 +
397         0.0657098242 * this->julian_day -
398         (this->longitude_deg / 15);
399
400     // bound to the half-open interval [0, 24) hrs
401     int Greenwich_mean_siderial_time_hrs_int = int(Greenwich_mean_siderial_time_hrs);
402     double Greenwich_mean_siderial_time_hrs_frac = Greenwich_mean_siderial_time_hrs -
403         Greenwich_mean_siderial_time_hrs_int;
404
405     Greenwich_mean_siderial_time_hrs = Greenwich_mean_siderial_time_hrs_int % 24;
406     Greenwich_mean_siderial_time_hrs += Greenwich_mean_siderial_time_hrs_frac;
407
408     return Greenwich_mean_siderial_time_hrs;
409 } /* __getGreenwichMeanSiderialTimeHrs() */
```

4.24.3.14 `__getGroundReflectedIrradiancekWm2()`

```
double Solar::__getGroundReflectedIrradiancekWm2 (
    double solar_resource_kWm2 ) [private]
```

Method to compute and return the ground reflected irradiance [kW/m²]. From eqn (6.21) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<code>solar_resource_kWm2</code>	Solar resource (i.e. global horizontal irradiance) [kW/m ²].
----------------------------------	--

Returns

The ground reflected irradiance [kW/m²].

```
992 {
993     double ground_reflected_irradiance_kWm2 =
994         this->albedo_ground_reflectance * solar_resource_kWm2 *
995         ((1 - cos(this->panel_tilt_rad)) / 2);
996
997     return ground_reflected_irradiance_kWm2;
998 } /* __getGroundReflectedIrradiancekWm2() */
```

4.24.3.15 `__getHourAngleRad()`

```
double Solar::__getHourAngleRad (
    double local_mean_siderial_time_hrs,
    double right_ascension_rad ) [private]
```

Method to compute and return the hour angle [rad] of the sun, bound to the open interval $(-\pi, \pi)$. From eqn (4.15) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>local_mean_siderial_time_hrs</i>	The local mean siderial time [hrs], bound to the half-open interval $[0, 24)$ hrs.
<i>right_ascension_rad</i>	The right ascension of the sun [rad], bound to the half-open interval $[0, 2\pi)$.

Returns

The hour angle [rad] of the sun, bound to the open interval $(-\pi, \pi)$.

```
568 {
569     // compute hour angle
570     double b_rad = 15 * (M_PI / 180.0) * local_mean_siderial_time_hrs -
571         right_ascension_rad;
572
573     double hour_angle_rad = b_rad;
574
575     // bound to open interval  $(-\pi, \pi)$ 
576     if (b_rad < -1 * M_PI) {
577         hour_angle_rad += 2 * M_PI;
578     }
579
580     else if (b_rad > M_PI) {
581         hour_angle_rad -= 2 * M_PI;
582     }
583
584     return hour_angle_rad;
585 } /* __getHourAngleRad() */
```

4.24.3.16 `__getLocalMeanSiderialTimeHrs()`

```
double Solar::__getLocalMeanSiderialTimeHrs (
    double Greenwich_mean_siderial_time_hrs ) [private]
```

Method to compute and return the local mean siderial time [hrs], bound to the half-open interval $[0, 24)$ hrs. From eqn (4.14) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>Greenwich_mean_siderial_time_hrs</i>	The Greenwich mean siderial time [hrs], bound to the half-open interval $[0, 24)$ hrs.
---	--

Returns

The local mean siderial time [hrs], bound to the half-open interval [0, 24) hrs.

```

437 {
438     // compute local mean siderial time
439     double local_mean_siderial_time_hrs = Greenwich_mean_siderial_time_hrs +
440         (this->longitude_deg / 15);
441
442     // bound to the half-open interval [0, 24) hrs
443     int local_mean_siderial_time_hrs_int = int(local_mean_siderial_time_hrs);
444     double local_mean_siderial_time_hrs_frac = local_mean_siderial_time_hrs -
445         local_mean_siderial_time_hrs_int;
446
447     local_mean_siderial_time_hrs = local_mean_siderial_time_hrs_int % 24;
448     local_mean_siderial_time_hrs += local_mean_siderial_time_hrs_frac;
449
450     return local_mean_siderial_time_hrs;
451 } /* __getLocalMeanSiderialTimeHrs() */

```

4.24.3.17 __getMeanAnomalyRad()

```

double Solar::__getMeanAnomalyRad (
    void ) [private]

```

Method to compute and return the mean anomaly [rad], bound to the half-open interval [0, 2pi). From eqn (4.8) of Gilman.

double Solar :: [__getMeanAnomalyRad\(void\)](#)

Ref: [Gilman et al. \[2018\]](#)

Returns

The mean anomaly [rad], bound to the half-open interval [0, 2pi).

```

273 {
274     // compute mean anomaly
275     double mean_anomaly_deg = 357.528 + 0.9856003 * this->julian_day;
276
277     // bound to the half-open interval [0, 360) deg.
278     int mean_anomaly_deg_int = int(mean_anomaly_deg);
279     double mean_anomaly_deg_frac = mean_anomaly_deg - mean_anomaly_deg_int;
280
281     mean_anomaly_deg = mean_anomaly_deg_int % 360;
282     mean_anomaly_deg += mean_anomaly_deg_frac;
283
284     // translate to rads
285     double mean_anomaly_rad = (M_PI / 180.0) * mean_anomaly_deg;
286
287     return mean_anomaly_rad;
288 } /* __getMeanAnomalyRad() */

```

4.24.3.18 __getMeanLongitudeDeg()

```

double Solar::__getMeanLongitudeDeg (
    void ) [private]

```

Method to compute and return the mean longitude [deg], bound to the half-open interval [0, 360). From eqn (4.7) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Returns

The mean longitude [deg], bound to the half-open interval [0, 360).

```

241 {
242     // compute mean longitude
243     double mean_longitude_deg = 280.46 + 0.9856474 * this->julian_day;
244
245     // bound to the half-open interval [0, 360) deg
246     int mean_longitude_deg_int = int(mean_longitude_deg);
247     double mean_longitude_deg_frac = mean_longitude_deg - mean_longitude_deg_int;
248
249     mean_longitude_deg = mean_longitude_deg_int % 360;
250     mean_longitude_deg += mean_longitude_deg_frac;
251
252     return mean_longitude_deg;
253 } /* __getMeanLongitudeDeg() */

```

4.24.3.19 __getObliquityOfEclipticRad()

```

double Solar::__getObliquityOfEclipticRad (
    void ) [private]

```

Method to compute and return the obliquity of the ecliptic [rad], bound to the half-open interval [0, 2pi). From eqn (4.10) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Returns

The obliquity of the ecliptic [rad], bound to the half-open interval [0, 2pi).

```

358 {
359     // compute obliquity of ecliptic
360     double obleq_deg = 23.439 - 0.0000004 * this->julian_day;
361
362     // bound to half-open interval [0, 360) deg
363     int obleq_deg_int = int(obleq_deg);
364     double obleq_deg_frac = obleq_deg - obleq_deg_int;
365
366     obleq_deg = obleq_deg_int % 360;
367     obleq_deg += obleq_deg_frac;
368
369     // translate to rads
370     double obleq_rad = (M_PI / 180.0) * obleq_deg;
371
372     return obleq_rad;
373 } /* __getObliquityOfEclipticRad() */

```

4.24.3.20 __getPlaneOfArrayIrradiancekWm2()

```

double Solar::__getPlaneOfArrayIrradiancekWm2 (
    int timestep,
    double dt_hrs,
    double solar_resource_kWm2 ) [private]

```

Method which takes in the solar resource at a particular point in time, and then returns the nominal plane of array irradiance. From eqn (7.1) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

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. global horizontal irradiance) [kW/m2].

Returns

The nominal plane of array irradiance [kW/m2].

```

1032 {
1033     // get mean longitude and mean anomaly
1034     double mean_longitude_deg = this->__getMeanLongitudeDeg();
1035     double mean_anomaly_rad = this->__getMeanAnomalyRad();
1036
1037
1038     // get ecliptic longitude and obliquity of the ecliptic
1039     double eclong_rad = this->__getEclipticLongitudeRad(
1040         mean_longitude_deg,
1041         mean_anomaly_rad
1042     );
1043
1044     double obleq_rad = this->__getObliquityOfEclipticRad();
1045
1046
1047     // get local mean siderial time
1048     double Greenwich_mean_siderial_time_hrs = this->__getGreenwichMeanSiderialTimeHrs();
1049
1050     double local_mean_siderial_time_hrs = this->__getLocalMeanSiderialTimeHrs(
1051         Greenwich_mean_siderial_time_hrs
1052     );
1053
1054
1055     // get right ascension, declination, and hour angle
1056     double right_ascension_rad = this->__getRightAscensionRad(eclong_rad, obleq_rad);
1057     double declination_rad = this->__getDeclinationRad(eclong_rad, obleq_rad);
1058
1059     double hour_angle_rad = this->__getHourAngleRad(
1060         local_mean_siderial_time_hrs,
1061         right_ascension_rad
1062     );
1063
1064
1065     // get solar azimuth and zenith
1066     double solar_azimuth_rad = this->__getSolarAzimuthRad(
1067         declination_rad,
1068         hour_angle_rad
1069     );
1070
1071     double solar_zenith_rad = this->__getSolarZenithRad(
1072         declination_rad,
1073         hour_angle_rad
1074     );
1075
1076
1077     // get diffuse horizontal irradiance (DHI) and direct normal irradiance (DNI)
1078     double diffuse_horizontal_irradiance_kWm2 = this->__getDiffuseHorizontalIrradiancekWm2(
1079         solar_resource_kWm2
1080     );
1081
1082     double direct_normal_irradiance_kWm2 = this->__getDirectNormalIrradiancekWm2(
1083         solar_resource_kWm2,
1084         diffuse_horizontal_irradiance_kWm2,
1085         solar_zenith_rad
1086     );
1087
1088
1089     // get angle of incidence
1090     double angle_of_incidence_rad = this->__getAngleOfIncidenceRad(
1091         solar_zenith_rad,
1092         solar_azimuth_rad
1093     );
1094
1095
1096     // compute plane of array irradiance as superposition of beam, diffuse, and ground
1097     // reflected.
1098     double plane_of_array_irradiance_kWm2 = 0;
1099
1100     plane_of_array_irradiance_kWm2 += this->__getBeamIrradiancekWm2(
1101         direct_normal_irradiance_kWm2,
1102         angle_of_incidence_rad

```

```

1103     );
1104
1105     plane_of_array_irradiance_kWm2 += this->__getDiffuseIrradiancekWm2(
1106         diffuse_horizontal_irradiance_kWm2
1107     );
1108
1109     plane_of_array_irradiance_kWm2 += this->__getGroundReflectedIrradiancekWm2(
1110         solar_resource_kWm2
1111     );
1112
1113     return plane_of_array_irradiance_kWm2;
1114 } /* __getPlaneOfArrayIrradiance() */

```

4.24.3.21 __getRightAscensionRad()

```

double Solar::__getRightAscensionRad (
    double eclong_rad,
    double obleq_rad ) [private]

```

Method to compute and return the right ascension of the sun [rad], bound to the half-open interval [0, 2pi). From eqn (4.11) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>eclong_rad</i>	The ecliptic longitude [rad], bound to the half-open interval [0, 2pi).
<i>obleq_rad</i>	The obliquity of the ecliptic, bound to the half-open interval [0, 2pi).

Returns

The right ascension of the sun [rad], bound to the half-open interval [0, 2pi).

```

520 {
521     // compute right ascension
522     double right_ascension_rad = atan(
523         (cos(obleq_rad) * sin(eclong_rad)) / cos(eclong_rad)
524     );
525
526     // bound to half-open interval [0, 2pi)
527     if (cos(eclong_rad) < 0) {
528         right_ascension_rad += M_PI;
529     }
530
531     else if (cos(obleq_rad) * sin(eclong_rad) < 0) {
532         right_ascension_rad += 2 * M_PI;
533     }
534
535     return right_ascension_rad;
536 } /* __getRightAscensionRad() */

```

4.24.3.22 __getSolarAltitudeRad()

```

double Solar::__getSolarAltitudeRad (
    double declination_rad,
    double hour_angle_rad ) [private]

```

Method to compute and return the sun altitude [rad], corrected for refraction and bound to the closed interval [0, pi/2]. From eqns (4.16) and (4.17) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>declination_rad</i>	The declination of the sun [rad], bound to the closed interval $[-\pi/2, \pi/2]$.
<i>hour_angle_rad</i>	The hour angle of the sun [rad], bound to the open interval $(-\pi, \pi)$.

Returns

The sun altitude [rad], corrected for refraction and bound to the closed interval $[0, \pi/2]$.

```

618 {
619     // compute un-corrected altitude
620     double a = sin(declination_rad) * sin(this->latitude_rad) +
621         cos(declination_rad) * cos(this->latitude_rad) * cos(hour_angle_rad);
622
623     double altitude_rad = 0;
624
625     if (a < -1) {
626         altitude_rad = -1 * M_PI_2;
627     }
628
629     else if (a > 1) {
630         altitude_rad = M_PI_2;
631     }
632
633     else {
634         altitude_rad = asin(a);
635     }
636
637     // correct for refraction
638     double altitude_deg = (180.0 / M_PI) * altitude_rad;
639
640     double refraction = 0.56;
641
642     if (altitude_deg > -0.56) {
643         refraction = 3.51567 *
644             (0.1594 + 0.0196 * altitude_deg + 0.00002 * pow(altitude_deg, 2)) *
645             pow(1 + 0.505 * altitude_deg + 0.0845 * pow(altitude_deg, 2), -1);
646     }
647
648     double altitude_corrected_rad = 0;
649
650     if (altitude_deg + refraction > 90) {
651         altitude_corrected_rad = M_PI_2;
652     }
653
654     else {
655         altitude_corrected_rad = (M_PI / 180.0) * (altitude_deg + refraction);
656     }
657
658     return altitude_corrected_rad;
659 } /* __getSolarAltitudeRad() */

```

4.24.3.23 __getSolarAzimuthRad()

```

double Solar::__getSolarAzimuthRad (
    double declination_rad,
    double hour_angle_rad ) [private]

```

Method to compute and return the solar azimuth [rad], bound to the closed interval $[-\pi, \pi]$. From eqns (4.16) and (4.18) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>declination_rad</i>	The declination of the sun [rad], bound to the closed interval $[-\pi/2, \pi/2]$.
<i>hour_angle_rad</i>	The hour angle of the sun [rad], bound to the open interval $(-\pi, \pi)$.

Returns

The solar azimuth [rad], bound to the closed interval $[-\pi, \pi]$.

```

691 {
692     // compute un-corrected altitude
693     double a = sin(declination_rad) * sin(this->latitude_rad) +
694         cos(declination_rad) * cos(this->latitude_rad) * cos(hour_angle_rad);
695
696     double altitude_rad = 0;
697
698     if (a < -1) {
699         altitude_rad = -1 * M_PI_2;
700     }
701
702     else if (a > 1) {
703         altitude_rad = M_PI_2;
704     }
705
706     else {
707         altitude_rad = asin(a);
708     }
709
710     // compute a term
711     a = (sin(altitude_rad) * sin(this->latitude_rad) - sin(declination_rad)) /
712         (cos(altitude_rad) * cos(this->latitude_rad));
713
714     // compute b term
715     double b_rad = 0;
716
717     if (cos(altitude_rad) == 0 or a < -1) {
718         b_rad = M_PI;
719     }
720
721     else if (a > 1) {
722         b_rad = 0;
723     }
724
725     else {
726         b_rad = acos(a);
727     }
728
729     // compute azimuth
730     double azimuth_rad = 0;
731
732     if (hour_angle_rad < -1 * M_PI) {
733         azimuth_rad = b_rad;
734     }
735
736     else if (
737         (hour_angle_rad >= -1 * M_PI and hour_angle_rad <= 0) or
738         hour_angle_rad > M_PI
739     ) {
740         azimuth_rad = M_PI - b_rad;
741     }
742
743     else {
744         azimuth_rad = M_PI + b_rad;
745     }
746
747     return azimuth_rad;
748 } /* __getSolarAzimuth() */

```

4.24.3.24 __getSolarZenithRad()

```

double Solar::__getSolarZenithRad (
    double declination_rad,
    double hour_angle_rad ) [private]

```

Method to compute and return the solar zenith [rad], bound to the open interval $(-\pi/2, \pi/2)$. From eqn (4.19) of Gilman.

Ref: [Gilman et al. \[2018\]](#)

Parameters

<i>declination_rad</i>	The declination of the sun [rad], bound to the closed interval $[-\pi/2, \pi/2]$.
<i>hour_angle_rad</i>	The hour angle of the sun [rad], bound to the open interval $(-\pi, \pi)$.

Returns

The solar zenith [rad], bound to the open interval $(-\pi/2, \pi/2)$.

```

779 {
780     double solar_zenith_rad = M_PI_2 - this->__getSolarAltitudeRad(
781         declination_rad,
782         hour_angle_rad
783     );
784
785     return solar_zenith_rad;
786 } /* __getSolarZenith() */

```

4.24.3.25 __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](#).

```

1229 {
1230     // 1. create filestream
1231     write_path += "summary_results.md";
1232     std::ofstream ofs;
1233     ofs.open(write_path, std::ofstream::out);
1234
1235     // 2. write summary results (markdown)
1236     ofs << "# ";
1237     ofs << std::to_string(int(ceil(this->capacity_kW)));
1238     ofs << " kW SOLAR Summary Results\n";
1239     ofs << "\n-----\n\n";
1240
1241     // 2.1. Production attributes
1242     ofs << "## Production Attributes\n";
1243     ofs << "\n";
1244
1245     ofs << "Capacity: " << this->capacity_kW << " kW \n";
1246     ofs << "\n";
1247
1248     ofs << "Production Override: (N = 0 / Y = 1): "
1249     << this->normalized_production_series_given << " \n";
1250     if (this->normalized_production_series_given) {
1251         ofs << "Path to Normalized Production Time Series: "
1252         << this->path_2_normalized_production_time_series << " \n";
1253     }
1254     ofs << "\n";
1255
1256     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
1257     ofs << "Capital Cost: " << this->capital_cost << " \n";
1258     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
1259     << " per kWh produced \n";
1260     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
1261     << " \n";
1262     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
1263     << " \n";

```

```

1264     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
1265     ofs << "\n";
1266
1267     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
1268     ofs << "\n-----\n\n";
1269
1270     // 2.2. Renewable attributes
1271     ofs << "## Renewable Attributes\n";
1272     ofs << "\n";
1273
1274     ofs << "Resource Key (1D): " << this->resource_key << " \n";
1275     ofs << "Firmness Factor: " << this->firmness_factor << " \n";
1276
1277     ofs << "\n-----\n\n";
1278
1279     // 2.3. Solar attributes
1280     ofs << "## Solar Attributes\n";
1281     ofs << "\n";
1282
1283     ofs << "Derating Factor: " << this->derating << " \n";
1284
1285     ofs << "\n-----\n\n";
1286
1287     // 2.4. Solar Results
1288     ofs << "## Results\n";
1289     ofs << "\n";
1290
1291     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
1292     ofs << "\n";
1293
1294     ofs << "Total Dispatch: " << this->total_dispatch_kWh
1295         << " kWh \n";
1296
1297     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
1298         << " per kWh dispatched \n";
1299     ofs << "\n";
1300
1301     ofs << "Running Hours: " << this->running_hours << " \n";
1302     ofs << "Replacements: " << this->n_replacements << " \n";
1303
1304     ofs << "\n-----\n\n";
1305
1306     ofs.close();
1307     return;
1308 } /* __writeSummary() */

```

4.24.3.26 __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](#).

```

1346 {
1347     // 1. create filestream
1348     write_path += "time_series_results.csv";
1349     std::ofstream ofs;
1350     ofs.open(write_path, std::ofstream::out);
1351
1352     // 2. write time series results (comma separated value)
1353     ofs << "Time (since start of data) [hrs],";
1354     ofs << "Solar Resource [kW/m2],";
1355     ofs << "Production [kW],";
1356     ofs << "Dispatch [kW],";
1357     ofs << "Storage [kW],";
1358     ofs << "Curtailment [kW],";
1359     ofs << "Capital Cost (actual),";
1360     ofs << "Operation and Maintenance Cost (actual),";
1361     ofs << "\n";
1362
1363     for (int i = 0; i < max_lines; i++) {
1364         ofs << time_vec_hrs_ptr->at(i) << ", ";
1365
1366         if (not this->normalized_production_series_given) {
1367             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ", ";
1368         }
1369
1370         else {
1371             ofs << "OVERRIDE" << ", ";
1372         }
1373
1374         ofs << this->production_vec_kW[i] << ", ";
1375         ofs << this->dispatch_vec_kW[i] << ", ";
1376         ofs << this->storage_vec_kW[i] << ", ";
1377         ofs << this->curtailment_vec_kW[i] << ", ";
1378         ofs << this->capital_cost_vec[i] << ", ";
1379         ofs << this->operation_maintenance_cost_vec[i] << ", ";
1380         ofs << "\n";
1381     }
1382
1383     ofs.close();
1384     return;
1385 } /* __writeTimeSeries() */

```

4.24.3.27 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](#).

```

1681 {

```

```

1682     // 1. invoke base class method
1683     load_kW = Renewable :: commit(
1684         timestep,
1685         dt_hrs,
1686         production_kW,
1687         load_kW
1688     );
1689
1690
1691     // 2. increment julian day
1692     this->julian_day += dt_hrs / 24;
1693
1694     return load_kW;
1695 } /* commit() */

```

4.24.3.28 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.

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. global horizontal irradiance) [kW/m2].

Returns

The production [kW] of the solar PV array.

Reimplemented from [Renewable](#).

```

1591 {
1592     // given production time series override
1593     if (this->normalized_production_series_given) {
1594         double production_kW = Production :: getProductionkW(timestep);
1595
1596         return production_kW;
1597     }
1598
1599     // check if no resource
1600     if (solar_resource_kWm2 <= 0) {
1601         return 0;
1602     }
1603
1604     // compute production
1605     double production_kW = 0;
1606
1607     switch (this->power_model) {
1608         case (SolarPowerProductionModel :: SOLAR_POWER_SIMPLE): {
1609             production_kW = this->__computeSimpleProductionkW(
1610                 timestep,
1611                 dt_hrs,
1612                 solar_resource_kWm2
1613             );
1614
1615             break;
1616         }
1617
1618         case (SolarPowerProductionModel :: SOLAR_POWER_DETAILED): {
1619             production_kW = this->__computeDetailedProductionkW(
1620                 timestep,

```

```

1621         dt_hrs,
1622         solar_resource_kWm2
1623     );
1624
1625     break;
1626 }
1627
1628 default: {
1629     std::string error_str = "ERROR: Solar::computeProductionkW(): ";
1630     error_str += "power model ";
1631     error_str += std::to_string(this->power_model);
1632     error_str += " not recognized";
1633
1634     #ifdef _WIN32
1635         std::cout << error_str << std::endl;
1636     #endif
1637
1638     throw std::runtime_error(error_str);
1639
1640     break;
1641 }
1642 }
1643
1644 return production_kW;
1645 } /* computeProductionkW() */

```

4.24.3.29 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](#).

```

1551 {
1552     // 1. reset attributes
1553     //...
1554
1555     // 2. invoke base class method
1556     Renewable :: handleReplacement(timestep);
1557
1558     return;
1559 } /* __handleReplacement() */

```

4.24.4 Member Data Documentation

4.24.4.1 albedo_ground_reflectance

```
double Solar::albedo_ground_reflectance
```

The albedo (ground reflectance) to be applied in modelling the solar PV array.

4.24.4.2 derating

```
double Solar::derating
```

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

4.24.4.3 julian_day

```
double Solar::julian_day
```

The number of days (including partial days) since 12:00 on 1 Jan 2000.

4.24.4.4 latitude_deg

```
double Solar::latitude_deg
```

The latitude of the solar PV array [deg].

4.24.4.5 latitude_rad

```
double Solar::latitude_rad
```

The latitude of the solar PV array [rad].

4.24.4.6 longitude_deg

```
double Solar::longitude_deg
```

The longitude of the solar PV array [deg].

4.24.4.7 longitude_rad

```
double Solar::longitude_rad
```

The longitude of the solar PV array [rad].

4.24.4.8 panel_azimuth_deg

```
double Solar::panel_azimuth_deg
```

The azimuth angle of the panels [deg], relative to north.

4.24.4.9 panel_azimuth_rad

```
double Solar::panel_azimuth_rad
```

The azimuth angle of the panels [rad], relative to north.

4.24.4.10 panel_tilt_deg

```
double Solar::panel_tilt_deg
```

The tilt angle of the panels [deg], relative to ground.

4.24.4.11 panel_tilt_rad

```
double Solar::panel_tilt_rad
```

The tilt angle of the panels [rad], relative to ground.

4.24.4.12 power_model

```
SolarPowerProductionModel Solar::power_model
```

The solar power production model to be applied.

4.24.4.13 power_model_string

```
std::string Solar::power_model_string
```

A string describing the active power production model.

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

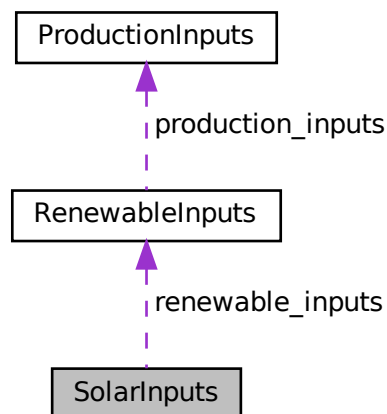
- header/Production/Renewable/[Solar.h](#)
- source/Production/Renewable/[Solar.cpp](#)

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 [firmness_factor](#) = 0.2
A factor $[0, 1]$ which defines how firm the production from this asset is.
- 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.).
- double [julian_day](#) = 0
The number of days (including partial days) since 12:00 on 1 Jan 2000.
- double [latitude_deg](#) = 0
The latitude of the solar PV array [deg].
- double [longitude_deg](#) = 0

- The longitude of the solar PV array [deg].*
 - double `panel_azimuth_deg` = 0
 - The azimuth angle of the panels [deg], relative to north.*
 - double `panel_tilt_deg` = 0
 - The tilt angle of the panels [deg], relative to ground.*
 - double `albedo_ground_reflectance` = 0.5
 - The albedo (ground reflectance) to be applied in modelling the solar PV array.*
 - `SolarPowerProductionModel power_model` = `SolarPowerProductionModel` :: `SOLAR_POWER_SIMPLE`
 - The solar power production model to be applied.*

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 `albedo_ground_reflectance`

```
double SolarInputs::albedo_ground_reflectance = 0.5
```

The albedo (ground reflectance) to be applied in modelling the solar PV array.

4.25.2.2 `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].

4.25.2.3 `derating`

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

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

4.25.2.4 firmness_factor

```
double SolarInputs::firmness_factor = 0.2
```

A factor [0, 1] which defines how firm the production from this asset is.

4.25.2.5 julian_day

```
double SolarInputs::julian_day = 0
```

The number of days (including partial days) since 12:00 on 1 Jan 2000.

4.25.2.6 latitude_deg

```
double SolarInputs::latitude_deg = 0
```

The latitude of the solar PV array [deg].

4.25.2.7 longitude_deg

```
double SolarInputs::longitude_deg = 0
```

The longitude of the solar PV array [deg].

4.25.2.8 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].

4.25.2.9 panel_azimuth_deg

```
double SolarInputs::panel_azimuth_deg = 0
```

The azimuth angle of the panels [deg], relative to north.

4.25.2.10 panel_tilt_deg

```
double SolarInputs::panel_tilt_deg = 0
```

The tilt angle of the panels [deg], relative to ground.

4.25.2.11 power_model

```
SolarPowerProductionModel SolarInputs::power_model = SolarPowerProductionModel :: SOLAR_POWER_SIMPLE
```

The solar power production model to be applied.

4.25.2.12 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.25.2.13 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:

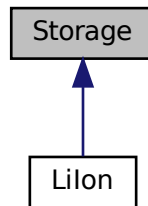
- [header/Production/Renewable/Solar.h](#)

4.26 Storage Class Reference

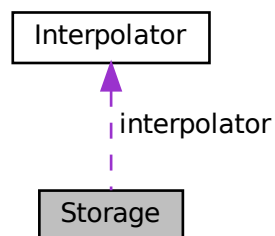
The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

```
#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

<i>n_points</i>	The number of points in the modelling time series.
<i>storage_inputs</i>	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 <code>time_vec_hrs</code> 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 time_vec_hrs 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:

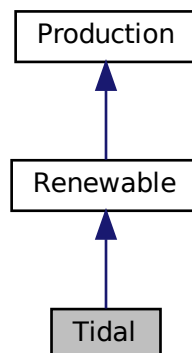
- header/Storage/[Storage.h](#)

4.28 Tidal Class Reference

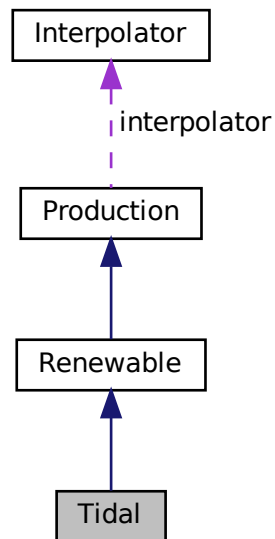
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

```
#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.

```
497 {
498     return;
499 } /* 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.

```

531 :
532 Renewable(
533     n_points,
534     n_years,
535     tidal_inputs.renewable_inputs,
536     time_vec_hrs_ptr
537 )
538 {
539     // 1. check inputs
540     this->__checkInputs(tidal_inputs);
541
542     // 2. set attributes
543     this->type = RenewableType :: TIDAL;
544     this->type_str = "TIDAL";
545
546     this->resource_key = tidal_inputs.resource_key;
547
548     this->firmness_factor = tidal_inputs.firmness_factor;
549
550     this->design_speed_ms = tidal_inputs.design_speed_ms;
551
552     this->power_model = tidal_inputs.power_model;
553
554     switch (this->power_model) {
555         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
556             this->power_model_string = "CUBIC";
557
558             break;
559         }
560
561         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
562             this->power_model_string = "EXPONENTIAL";
563
564             break;
565         }
566
567         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
568             this->power_model_string = "LOOKUP";
569
570             break;
571         }
572
573         default: {
574             std::string error_str = "ERROR: Tidal(): ";
575             error_str += "power production model ";
576             error_str += std::to_string(this->power_model);
577             error_str += " not recognized";
578
579             #ifdef _WIN32
580                 std::cout << error_str << std::endl;
581             #endif
582
583             throw std::runtime_error(error_str);
584
585             break;
586         }
587     }
588
589     if (tidal_inputs.capital_cost < 0) {
590         this->capital_cost = this->__getGenericCapitalCost();
591     }
592     else {
593         this->capital_cost = tidal_inputs.capital_cost;
594     }
595
596     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
597         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
598     }
599     else {
600         this->operation_maintenance_cost_kWh =
601             tidal_inputs.operation_maintenance_cost_kWh;
602     }
603
604     if (not this->is_sunk) {
605         this->capital_cost_vec[0] = this->capital_cost;
606     }

```

```

607
608     // 3. construction print
609     if (this->print_flag) {
610         std::cout << "Tidal object constructed at " << this << std::endl;
611     }
612
613     return;
614 } /* Renewable() */

```

4.28.2.3 ~Tidal()

```

Tidal::~Tidal (
    void )

```

Destructor for the [Tidal](#) class.

```

801 {
802     // 1. destruction print
803     if (this->print_flag) {
804         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
805     }
806
807     return;
808 } /* ~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     // 2. check firmness_factor
87     if (
88         tidal_inputs.firmness_factor < 0 or
89         tidal_inputs.firmness_factor > 1
90     ) {
91         std::string error_str = "ERROR: Tidal(): ";

```

```

92         error_str += "TidalInputs::firmness_factor must be in the closed 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     return;
102 } /* __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.

```

192 {
193     double production = 0;
194
195     if (
196         tidal_resource_ms < 0.15 * this->design_speed_ms or
197         tidal_resource_ms > 1.25 * this->design_speed_ms
198     ){
199         production = 0;
200     }
201
202     else if (
203         0.15 * this->design_speed_ms <= tidal_resource_ms and
204         tidal_resource_ms <= this->design_speed_ms
205     ) {
206         production = (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
207     }
208
209     else {
210         production = 1;
211     }
212
213     return production * this->capacity_kW;
214 } /* __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.

```
248 {
249     double production = 0;
250
251     double turbine_speed =
252         (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
253
254     if (turbine_speed < -0.71 or turbine_speed > 0.65) {
255         production = 0;
256     }
257
258     else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
259         production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
260     }
261
262     else {
263         production = 1;
264     }
265
266     return production * this->capacity_kW;
267 } /* __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.

```
299 {
300     // *** WORK IN PROGRESS *** //
301
302     return 0;
303 } /* __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].

```
124 {
125     double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;
126
127     return capital_cost_per_kW * this->capacity_kW;
128 } /* __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].

```
151 {
152     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
153
154     return operation_maintenance_cost_kWh;
155 } /* __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](#).

```

321 {
322     // 1. create filestream
323     write_path += "summary_results.md";
324     std::ofstream ofs;
325     ofs.open(write_path, std::ofstream::out);
326
327     // 2. write summary results (markdown)
328     ofs << "# ";
329     ofs << std::to_string(int(ceil(this->capacity_kW)));
330     ofs << " kW TIDAL Summary Results\n";
331     ofs << "\n-----\n\n";
332
333     // 2.1. Production attributes
334     ofs << "## Production Attributes\n";
335     ofs << "\n";
336
337     ofs << "Capacity: " << this->capacity_kW << " kW \n";
338     ofs << "\n";
339
340     ofs << "Production Override: (N = 0 / Y = 1): "
341     << this->normalized_production_series_given << " \n";
342     if (this->normalized_production_series_given) {
343         ofs << "Path to Normalized Production Time Series: "
344         << this->path_2_normalized_production_time_series << " \n";
345     }
346     ofs << "\n";
347
348     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
349     ofs << "Capital Cost: " << this->capital_cost << " \n";
350     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
351     << " per kWh produced \n";
352     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
353     << " \n";
354     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
355     << " \n";
356     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
357     ofs << "\n";
358
359     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
360     ofs << "\n-----\n\n";
361
362     // 2.2. Renewable attributes
363     ofs << "## Renewable Attributes\n";
364     ofs << "\n";
365
366     ofs << "Resource Key (1D): " << this->resource_key << " \n";
367     ofs << "Firmness Factor: " << this->firmness_factor << " \n";
368
369     ofs << "\n-----\n\n";
370
371     // 2.3. Tidal attributes
372     ofs << "## Tidal Attributes\n";
373     ofs << "\n";
374
375     ofs << "Power Production Model: " << this->power_model_string << " \n";
376     ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
377
378     ofs << "\n-----\n\n";
379
380     // 2.4. Tidal Results
381     ofs << "## Results\n";
382     ofs << "\n";
383
384     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
385     ofs << "\n";
386
387     ofs << "Total Dispatch: " << this->total_dispatch_kWh
388     << " kWh \n";
389
390     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
391     << " per kWh dispatched \n";
392     ofs << "\n";
393
394     ofs << "Running Hours: " << this->running_hours << " \n";
395     ofs << "Replacements: " << this->n_replacements << " \n";
396
397     ofs << "\n-----\n\n";

```

```

398
399     ofs.close();
400
401     return;
402 } /* __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](#).

```

440 {
441     // 1. create filestream
442     write_path += "time_series_results.csv";
443     std::ofstream ofs;
444     ofs.open(write_path, std::ofstream::out);
445
446     // 2. write time series results (comma separated value)
447     ofs << "Time (since start of data) [hrs],";
448     ofs << "Tidal Resource [m/s],";
449     ofs << "Production [kW],";
450     ofs << "Dispatch [kW],";
451     ofs << "Storage [kW],";
452     ofs << "Curtailement [kW],";
453     ofs << "Capital Cost (actual),";
454     ofs << "Operation and Maintenance Cost (actual),";
455     ofs << "\n";
456
457     for (int i = 0; i < max_lines; i++) {
458         ofs << time_vec_hrs_ptr->at(i) << ",";
459
460         if (not this->normalized_production_series_given) {
461             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ",";
462         }
463
464         else {
465             ofs << "OVERRIDE" << ",";
466         }
467
468         ofs << this->production_vec_kW[i] << ",";
469         ofs << this->dispatch_vec_kW[i] << ",";
470         ofs << this->storage_vec_kW[i] << ",";
471         ofs << this->curtailement_vec_kW[i] << ",";
472         ofs << this->capital_cost_vec[i] << ",";
473         ofs << this->operation_maintenance_cost_vec[i] << ",";
474         ofs << "\n";
475     }
476
477     return;
478 } /* __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](#).

```
773 {
774     // 1. invoke base class method
775     load_kW = Renewable::commit(
776         timestep,
777         dt_hrs,
778         production_kW,
779         load_kW
780     );
781
782
783     //...
784
785     return load_kW;
786 } /* 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](#).

```

672 {
673     // given production time series override
674     if (this->normalized_production_series_given) {
675         double production_kW = Production :: getProductionkW(timestep);
676
677         return production_kW;
678     }
679
680     // check if no resource
681     if (tidal_resource_ms <= 0) {
682         return 0;
683     }
684
685     // compute production
686     double production_kW = 0;
687
688     switch (this->power_model) {
689         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
690             production_kW = this->__computeCubicProductionkW(
691                 timestep,
692                 dt_hrs,
693                 tidal_resource_ms
694             );
695
696             break;
697         }
698
699         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
700             production_kW = this->__computeExponentialProductionkW(
701                 timestep,
702                 dt_hrs,
703                 tidal_resource_ms
704             );
705
706             break;
707         }
708
709         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
710             production_kW = this->__computeLookupProductionkW(
711                 timestep,
712                 dt_hrs,
713                 tidal_resource_ms
714             );
715
716             break;
717         }
718
719         default: {
720             std::string error_str = "ERROR: Tidal::computeProductionkW(): ";
721             error_str += "power model ";
722             error_str += std::to_string(this->power_model);
723             error_str += " not recognized";
724
725             #ifdef _WIN32
726                 std::cout << error_str << std::endl;
727             #endif
728
729             throw std::runtime_error(error_str);
730
731             break;
732         }
733     }
734
735     return production_kW;
736 }
737 /* 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](#).

```
632 {  
633     // 1. reset attributes  
634     //...  
635  
636     // 2. invoke base class method  
637     Renewable :: handleReplacement(timestep);  
638  
639     return;  
640 } /* __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:

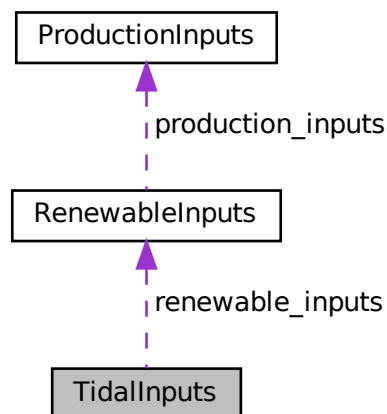
- header/Production/Renewable/[Tidal.h](#)
- source/Production/Renewable/[Tidal.cpp](#)

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` `firmness_factor` = 0.8
A factor $[0, 1]$ which defines how firm the production from this asset is.
- `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 firmness_factor

```
double TidalInputs::firmness_factor = 0.8
```

A factor [0, 1] which defines how firm the production from this asset is.

4.29.2.4 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].

4.29.2.5 power_model

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

The tidal power production model to be applied.

4.29.2.6 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.29.2.7 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:

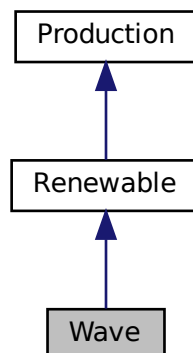
- [header/Production/Renewable/Tidal.h](#)

4.30 Wave Class Reference

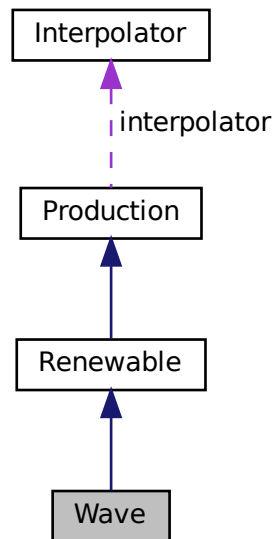
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

```
#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.

```
559 {
560     return;
561 } /* 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.

```

593 :
594 Renewable(
595     n_points,
596     n_years,
597     wave_inputs.renewable_inputs,
598     time_vec_hrs_ptr
599 )
600 {
601     // 1. check inputs
602     this->__checkInputs(wave_inputs);
603
604     // 2. set attributes
605     this->type = RenewableType :: WAVE;
606     this->type_str = "WAVE";
607
608     this->resource_key = wave_inputs.resource_key;
609
610     this->firmness_factor = wave_inputs.firmness_factor;
611
612     this->design_significant_wave_height_m =
613         wave_inputs.design_significant_wave_height_m;
614     this->design_energy_period_s = wave_inputs.design_energy_period_s;
615
616     this->power_model = wave_inputs.power_model;
617
618     switch (this->power_model) {
619         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
620             this->power_model_string = "GAUSSIAN";
621
622             break;
623         }
624
625         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
626             this->power_model_string = "PARABOLOID";
627
628             break;
629         }
630
631         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
632             this->power_model_string = "LOOKUP";
633
634             this->interpolator.addData2D(
635                 0,
636                 wave_inputs.path_2_normalized_performance_matrix
637             );
638
639             break;
640         }
641
642         default: {
643             std::string error_str = "ERROR: Wave(): ";
644             error_str += "power production model ";
645             error_str += std::to_string(this->power_model);
646             error_str += " not recognized";
647
648             #ifdef _WIN32
649                 std::cout << error_str << std::endl;
650             #endif
651
652             throw std::runtime_error(error_str);
653
654             break;
655         }
656     }
657
658     if (wave_inputs.capital_cost < 0) {
659         this->capital_cost = this->__getGenericCapitalCost();
660     }
661     else {
662         this->capital_cost = wave_inputs.capital_cost;
663     }
664
665     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
666         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
667     }
668     else {

```

```

669         this->operation_maintenance_cost_kWh =
670             wave_inputs.operation_maintenance_cost_kWh;
671     }
672
673     if (not this->is_sunk) {
674         this->capital_cost_vec[0] = this->capital_cost;
675     }
676
677     // 3. construction print
678     if (this->print_flag) {
679         std::cout << "Wave object constructed at " << this << std::endl;
680     }
681
682     return;
683 } /* Renewable() */

```

4.30.2.3 ~Wave()

```

Wave::~~Wave (
    void )

```

Destructor for the [Wave](#) class.

```

876 {
877     // 1. destruction print
878     if (this->print_flag) {
879         std::cout << "Wave object at " << this << " destroyed" << std::endl;
880     }
881
882     return;
883 } /* ~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 // 4. check firmness_factor
106 if (
107     wave_inputs.firmness_factor < 0 or
108     wave_inputs.firmness_factor > 1
109 ) {
110     std::string error_str = "ERROR: Wave(): ";
111     error_str += "WaveInputs::firmness_factor must be in the closed interval [0, 1]";
112
113     #ifdef _WIN32
114         std::cout << error_str << std::endl;
115     #endif
116
117     throw std::invalid_argument(error_str);
118 }
119
120 return;
121 } /* __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.

```

216 {
217     double H_s_nondim =
218         (significant_wave_height_m - this->design_significant_wave_height_m) /
219         this->design_significant_wave_height_m;
220
221     double T_e_nondim =
222         (energy_period_s - this->design_energy_period_s) /
223         this->design_energy_period_s;
224
225     double production = exp(
226         -2.25119 * pow(T_e_nondim, 2) +
227         3.44570 * T_e_nondim * H_s_nondim -
228         4.01508 * pow(H_s_nondim, 2)
229     );
230
231     return production * this->capacity_kW;
232 } /* __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.

```

333 {
334     double prod = this->interpolator.interp2D(
335         0,
336         significant_wave_height_m,
337         energy_period_s
338     );
339
340     return prod * this->capacity_kW;
341 } /* __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.

```

273 {
274     // first, check for idealized wave breaking (deep water)
275     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
276         return 0;
277     }
278
279     // otherwise, apply generic quadratic performance model
280     // (with outputs bounded to [0, 1])
281     double production =
282         0.289 * significant_wave_height_m -
283         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
284         0.0169 * energy_period_s;
285
286     if (production < 0) {
287         production = 0;
288     }
289
290     else if (production > 1) {
291         production = 1;
292     }
293
294     return production * this->capacity_kW;
295 } /* __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].

```

143 {
144     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
145
146     return capital_cost_per_kW * this->capacity_kW;
147 } /* __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].

```
171 {
172     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
173
174     return operation_maintenance_cost_kWh;
175 } /* __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](#).

```
359 {
360     // 1. create filestream
361     write_path += "summary_results.md";
362     std::ofstream ofs;
363     ofs.open(write_path, std::ofstream::out);
364
365     // 2. write summary results (markdown)
366     ofs << "# ";
367     ofs << std::to_string(int(ceil(this->capacity_kW)));
368     ofs << " kW WAVE Summary Results\n";
369     ofs << "\n-----\n\n";
370
371     // 2.1. Production attributes
372     ofs << "## Production Attributes\n";
373     ofs << "\n";
374
375     ofs << "Capacity: " << this->capacity_kW << " kW \n";
376     ofs << "\n";
377
378     ofs << "Production Override: (N = 0 / Y = 1): "
379         << this->normalized_production_series_given << " \n";
380     if (this->normalized_production_series_given) {
381         ofs << "Path to Normalized Production Time Series: "
382             << this->path_2_normalized_production_time_series << " \n";
383     }
384     ofs << "\n";
```

```

385
386 ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « " \n";
387 ofs « "Capital Cost: " « this->capital_cost « " \n";
388 ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
389 « " per kWh produced \n";
390 ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
391 « " \n";
392 ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
393 « " \n";
394 ofs « "Real Discount Rate (annual): " « this->real_discount_annual « " \n";
395 ofs « "\n";
396
397 ofs « "Replacement Running Hours: " « this->replace_running_hrs « " \n";
398 ofs « "\n-----\n\n";
399
400 // 2.2. Renewable attributes
401 ofs « "## Renewable Attributes\n";
402 ofs « "\n";
403
404 ofs « "Resource Key (2D): " « this->resource_key « " \n";
405 ofs « "Firmness Factor: " « this->firmness_factor « " \n";
406
407 ofs « "\n-----\n\n";
408
409 // 2.3. Wave attributes
410 ofs « "## Wave Attributes\n";
411 ofs « "\n";
412
413 ofs « "Power Production Model: " « this->power_model_string « " \n";
414 switch (this->power_model) {
415     case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
416         ofs « "Design Significant Wave Height: "
417         « this->design_significant_wave_height_m « " m \n";
418
419         ofs « "Design Energy Period: " « this->design_energy_period_s « " s \n";
420
421         break;
422     }
423
424     case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
425         ofs « "Normalized Performance Matrix: "
426         « this->interpolator.path_map_2D[0] « " \n";
427
428         break;
429     }
430
431     default: {
432         // write nothing!
433
434         break;
435     }
436 }
437
438 ofs « "\n-----\n\n";
439
440 // 2.4. Wave Results
441 ofs « "## Results\n";
442 ofs « "\n";
443
444 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
445 ofs « "\n";
446
447 ofs « "Total Dispatch: " « this->total_dispatch_kWh
448 « " kWh \n";
449
450 ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
451 « " per kWh dispatched \n";
452 ofs « "\n";
453
454 ofs « "Running Hours: " « this->running_hours « " \n";
455 ofs « "Replacements: " « this->n_replacements « " \n";
456
457 ofs « "\n-----\n\n";
458
459 ofs.close();
460
461 return;
462 } /* __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](#).

```

500 {
501     // 1. create filestream
502     write_path += "time_series_results.csv";
503     std::ofstream ofs;
504     ofs.open(write_path, std::ofstream::out);
505
506     // 2. write time series results (comma separated value)
507     ofs << "Time (since start of data) [hrs],";
508     ofs << "Significant Wave Height [m],";
509     ofs << "Energy Period [s],";
510     ofs << "Production [kW],";
511     ofs << "Dispatch [kW],";
512     ofs << "Storage [kW],";
513     ofs << "Curtailment [kW],";
514     ofs << "Capital Cost (actual),";
515     ofs << "Operation and Maintenance Cost (actual),";
516     ofs << "\n";
517
518     for (int i = 0; i < max_lines; i++) {
519         ofs << time_vec_hrs_ptr->at(i) << ", ";
520
521         if (not this->normalized_production_series_given) {
522             ofs << resource_map_2D_ptr->at(this->resource_key)[i][0] << ", ";
523             ofs << resource_map_2D_ptr->at(this->resource_key)[i][1] << ", ";
524         }
525
526         else {
527             ofs << "OVERRIDE" << ", ";
528             ofs << "OVERRIDE" << ", ";
529         }
530
531         ofs << this->production_vec_kW[i] << ", ";
532         ofs << this->dispatch_vec_kW[i] << ", ";
533         ofs << this->storage_vec_kW[i] << ", ";
534         ofs << this->curtailment_vec_kW[i] << ", ";
535         ofs << this->capital_cost_vec[i] << ", ";
536         ofs << this->operation_maintenance_cost_vec[i] << ", ";
537         ofs << "\n";
538     }
539
540     return;
541 } /* __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](#).

```
848 {
849     // 1. invoke base class method
850     load_kW = Renewable::commit(
851         timestep,
852         dt_hrs,
853         production_kW,
854         load_kW
855     );
856
857
858     //...
859
860     return load_kW;
861 } /* 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](#).

```

745 {
746     // given production time series override
747     if (this->normalized_production_series_given) {
748         double production_kW = Production :: getProductionkW(timestep);
749
750         return production_kW;
751     }
752
753     // check if no resource
754     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
755         return 0;
756     }
757
758     // compute production
759     double production_kW = 0;
760
761     switch (this->power_model) {
762         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
763             production_kW = this->__computeParaboloidProductionkW(
764                 timestep,
765                 dt_hrs,
766                 significant_wave_height_m,
767                 energy_period_s
768             );
769
770             break;
771         }
772
773         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
774             production_kW = this->__computeGaussianProductionkW(
775                 timestep,
776                 dt_hrs,
777                 significant_wave_height_m,
778                 energy_period_s
779             );
780
781             break;
782         }
783
784         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
785             production_kW = this->__computeLookupProductionkW(
786                 timestep,
787                 dt_hrs,
788                 significant_wave_height_m,
789                 energy_period_s
790             );
791
792             break;
793         }
794
795         default: {
796             std::string error_str = "ERROR: Wave::computeProductionkW(): ";
797             error_str += "power model ";
798             error_str += std::to_string(this->power_model);
799             error_str += " not recognized";
800
801             #ifdef _WIN32
802                 std::cout << error_str << std::endl;
803             #endif
804
805             throw std::runtime_error(error_str);
806
807             break;
808         }
809     }
810
811     return production_kW;
812 } /* 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](#).

```

701 {
702     // 1. reset attributes
703     //...
704
705     // 2. invoke base class method
706     Renewable::handleReplacement(timestep);
707
708     return;
709 } /* __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:

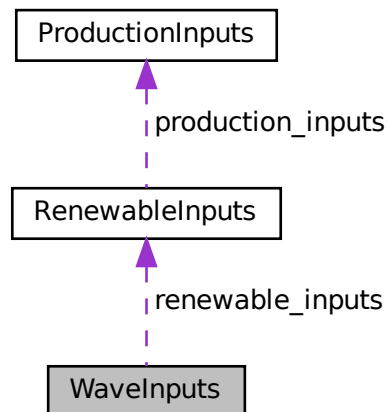
- header/Production/Renewable/[Wave.h](#)
- source/Production/Renewable/[Wave.cpp](#)

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 `firmness_factor` = 0.8
A factor [0, 1] which defines how firm the production from this asset is.
- 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 firmness_factor

```
double WaveInputs::firmness_factor = 0.8
```

A factor [0, 1] which defines how firm the production from this asset is.

4.31.2.5 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].

4.31.2.6 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.

4.31.2.7 power_model

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

The wave power production model to be applied.

4.31.2.8 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.31.2.9 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:

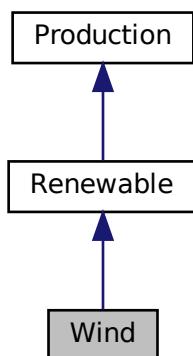
- [header/Production/Renewable/Wave.h](#)

4.32 Wind Class Reference

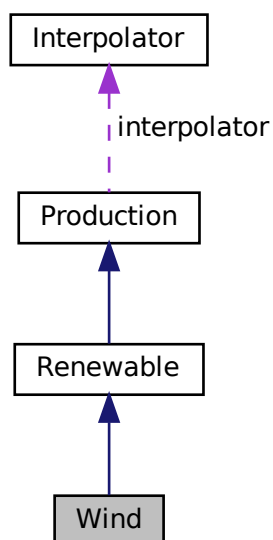
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

```
#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.

```
517 {
518     return;
519 } /* 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.

```
551 :
552 Renewable(
553     n_points,
554     n_years,
555     wind_inputs.renewable_inputs,
556     time_vec_hrs_ptr
557 )
558 {
559     // 1. check inputs
560     this->__checkInputs(wind_inputs);
561
562     // 2. set attributes
563     this->type = RenewableType :: WIND;
564     this->type_str = "WIND";
565
566     this->resource_key = wind_inputs.resource_key;
567
568     this->firmness_factor = wind_inputs.firmness_factor;
569
570     this->design_speed_ms = wind_inputs.design_speed_ms;
571
572     this->power_model = wind_inputs.power_model;
573
574     switch (this->power_model) {
575         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
576             this->power_model_string = "CUBIC";
577
578             break;
579         }
580
581         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
```

```

582         this->power_model_string = "EXPONENTIAL";
583
584         break;
585     }
586
587     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
588         this->power_model_string = "LOOKUP";
589
590         break;
591     }
592
593     default: {
594         std::string error_str = "ERROR: Wind(): ";
595         error_str += "power production model ";
596         error_str += std::to_string(this->power_model);
597         error_str += " not recognized";
598
599         #ifdef _WIN32
600             std::cout << error_str << std::endl;
601         #endif
602
603         throw std::runtime_error(error_str);
604
605         break;
606     }
607 }
608
609 if (wind_inputs.capital_cost < 0) {
610     this->capital_cost = this->__getGenericCapitalCost();
611 }
612 else {
613     this->capital_cost = wind_inputs.capital_cost;
614 }
615
616 if (wind_inputs.operation_maintenance_cost_kWh < 0) {
617     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
618 }
619 else {
620     this->operation_maintenance_cost_kWh =
621         wind_inputs.operation_maintenance_cost_kWh;
622 }
623
624 if (not this->is_sunk) {
625     this->capital_cost_vec[0] = this->capital_cost;
626 }
627
628 // 3. construction print
629 if (this->print_flag) {
630     std::cout << "Wind object constructed at " << this << std::endl;
631 }
632
633 return;
634 } /* Renewable() */

```

4.32.2.3 ~Wind()

```

Wind::~~Wind (
    void )

```

Destructor for the `Wind` class.

```

820 {
821     // 1. destruction print
822     if (this->print_flag) {
823         std::cout << "Wind object at " << this << " destroyed" << std::endl;
824     }
825
826     return;
827 } /* ~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

<code>wind_inputs</code>	A structure of <code>Wind</code> 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     // 2. check firmness_factor
88     if (
89         wind_inputs.firmness_factor < 0 or
90         wind_inputs.firmness_factor > 1
91     ) {
92         std::string error_str = "ERROR: Wind(): ";
93         error_str += "WindInputs::firmness_factor must be in the closed interval [0, 1]";
94
95         #ifdef WIN32
96             std::cout << error_str << std::endl;
97         #endif
98
99         throw std::invalid_argument(error_str);
100     }
101
102     return;
103 } /* __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.

```

191 {
192     double production = 0;
193
194     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
195         this->design_speed_ms;
196
197     if (turbine_speed < -0.7857 or turbine_speed > 0.7857) {
198         production = 0;
199     }
200
201     else if (turbine_speed >= -0.7857 and turbine_speed <= 0) {
202         production = (1 / pow(this->design_speed_ms, 3)) * pow(wind_resource_ms, 3);
203     }
204
205     else {
206         production = 1;
207     }
208
209     return production * this->capacity_kW;
210 } /* __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.

```

244 {
245     double production = 0;
246
247     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
248         this->design_speed_ms;
249
250     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
251         production = 0;

```

```

252     }
253
254     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
255         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
256     }
257
258     else {
259         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
260     }
261
262     return production * this->capacity_kW;
263 } /* __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.

```

295 {
296     // *** WORK IN PROGRESS *** //
297
298     return 0;
299 } /* __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].

```

125 {
126     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
127
128     return capital_cost_per_kW * this->capacity_kW;
129 } /* __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].

```
152 {
153     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
154
155     return operation_maintenance_cost_kWh;
156 } /* __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](#).

```
317 {
318     // 1. create filestream
319     write_path += "summary_results.md";
320     std::ofstream ofs;
321     ofs.open(write_path, std::ofstream::out);
322
323     // 2. write summary results (markdown)
324     ofs << "# ";
325     ofs << std::to_string(int(ceil(this->capacity_kW)));
326     ofs << " kW WIND Summary Results\n";
327     ofs << "\n-----\n\n";
328
329
330     // 2.1. Production attributes
331     ofs << "## Production Attributes\n";
332     ofs << "\n";
333
334     ofs << "Capacity: " << this->capacity_kW << " kW \n";
335     ofs << "\n";
336
337     ofs << "Production Override: (N = 0 / Y = 1): "
338         << this->normalized_production_series_given << " \n";
339     if (this->normalized_production_series_given) {
340         ofs << "Path to Normalized Production Time Series: "
341             << this->path_2_normalized_production_time_series << " \n";
342     }
343     ofs << "\n";
344
345     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
346     ofs << "Capital Cost: " << this->capital_cost << " \n";
```

```

347 ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
348     << " per kWh produced \n";
349 ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
350     << " \n";
351 ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
352     << " \n";
353 ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
354 ofs << "\n";
355
356 ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
357 ofs << "\n-----\n\n";
358
359 // 2.2. Renewable attributes
360 ofs << "## Renewable Attributes\n";
361 ofs << "\n";
362
363 ofs << "Resource Key (ID): " << this->resource_key << " \n";
364 ofs << "Firmness Factor: " << this->firmness_factor << " \n";
365
366 ofs << "\n-----\n\n";
367
368 // 2.3. Wind attributes
369 ofs << "## Wind Attributes\n";
370 ofs << "\n";
371
372 ofs << "Power Production Model: " << this->power_model_string << " \n";
373 switch (this->power_model) {
374     case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
375         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
376
377         break;
378     }
379
380     case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
381         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
382
383         break;
384     }
385
386     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
387         //...
388
389         break;
390     }
391
392     default: {
393         // write nothing!
394
395         break;
396     }
397 }
398 ofs << "\n-----\n\n";
399
400 // 2.4. Wind Results
401 ofs << "## Results\n";
402 ofs << "\n";
403
404 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
405 ofs << "\n";
406
407 ofs << "Total Dispatch: " << this->total_dispatch_kWh
408     << " kWh \n";
409
410 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
411     << " per kWh dispatched \n";
412 ofs << "\n";
413
414 ofs << "Running Hours: " << this->running_hours << " \n";
415 ofs << "Replacements: " << this->n_replacements << " \n";
416
417 ofs << "\n-----\n\n";
418
419 ofs.close();
420
421 return;
422 }
423 } /* __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](#).

```
461 {
462     // 1. create filestream
463     write_path += "time_series_results.csv";
464     std::ofstream ofs;
465     ofs.open(write_path, std::ofstream::out);
466
467     // 2. write time series results (comma separated value)
468     ofs << "Time (since start of data) [hrs],";
469     ofs << "Wind Resource [m/s],";
470     ofs << "Production [kW],";
471     ofs << "Dispatch [kW],";
472     ofs << "Storage [kW],";
473     ofs << "Curtailement [kW],";
474     ofs << "Capital Cost (actual),";
475     ofs << "Operation and Maintenance Cost (actual),";
476     ofs << "\n";
477
478     for (int i = 0; i < max_lines; i++) {
479         ofs << time_vec_hrs_ptr->at(i) << ", ";
480
481         if (not this->normalized_production_series_given) {
482             ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ", ";
483         }
484
485         else {
486             ofs << "OVERRIDE" << ", ";
487         }
488
489         ofs << this->production_vec_kW[i] << ", ";
490         ofs << this->dispatch_vec_kW[i] << ", ";
491         ofs << this->storage_vec_kW[i] << ", ";
492         ofs << this->curtailement_vec_kW[i] << ", ";
493         ofs << this->capital_cost_vec[i] << ", ";
494         ofs << this->operation_maintenance_cost_vec[i] << ", ";
495         ofs << "\n";
496     }
497
498     return;
499 } /* __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](#).

```

792 {
793     // 1. invoke base class method
794     load_kW = Renewable::commit(
795         timestep,
796         dt_hrs,
797         production_kW,
798         load_kW
799     );
800
801
802     //...
803
804     return load_kW;
805 } /* 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](#).

```

692 {
693     // given production time series override
694     if (this->normalized_production_series_given) {
695         double production_kW = Production::getProductionkW(timestep);
696
697         return production_kW;
698     }
699

```

```

700     // check if no resource
701     if (wind_resource_ms <= 0) {
702         return 0;
703     }
704
705     // compute production
706     double production_kW = 0;
707
708     switch (this->power_model) {
709         case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
710             production_kW = this->__computeCubicProductionkW(
711                 timestep,
712                 dt_hrs,
713                 wind_resource_ms
714             );
715
716             break;
717         }
718
719         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
720             production_kW = this->__computeExponentialProductionkW(
721                 timestep,
722                 dt_hrs,
723                 wind_resource_ms
724             );
725
726             break;
727         }
728
729         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
730             production_kW = this->__computeLookupProductionkW(
731                 timestep,
732                 dt_hrs,
733                 wind_resource_ms
734             );
735
736             break;
737         }
738
739         default: {
740             std::string error_str = "ERROR: Wind::computeProductionkW(): ";
741             error_str += "power model ";
742             error_str += std::to_string(this->power_model);
743             error_str += " not recognized";
744
745             #ifdef _WIN32
746                 std::cout << error_str << std::endl;
747             #endif
748
749             throw std::runtime_error(error_str);
750
751             break;
752         }
753     }
754
755     return production_kW;
756 } /* computeProductionkW() */

```

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](#).

```

652 {
653     // 1. reset attributes
654     //...

```

```
655
656     // 2. invoke base class method
657     Renewable :: handleReplacement(timestep);
658
659     return;
660 } /* __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:

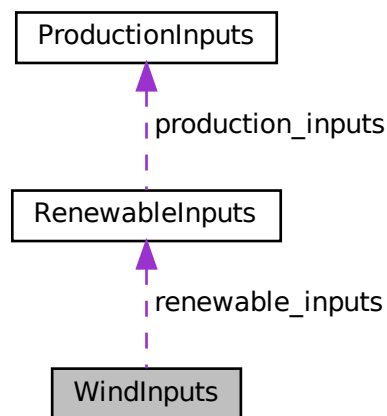
- header/Production/Renewable/[Wind.h](#)
- source/Production/Renewable/[Wind.cpp](#)

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 [firmness_factor](#) = 0.5
A factor $[0, 1]$ which defines how firm the production from this asset is.
- 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 firmness_factor

```
double WindInputs::firmness_factor = 0.5
```

A factor [0, 1] which defines how firm the production from this asset is.

4.33.2.4 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].

4.33.2.5 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel :: WIND_POWER_CUBIC
```

The wind power production model to be applied.

4.33.2.6 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.33.2.7 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

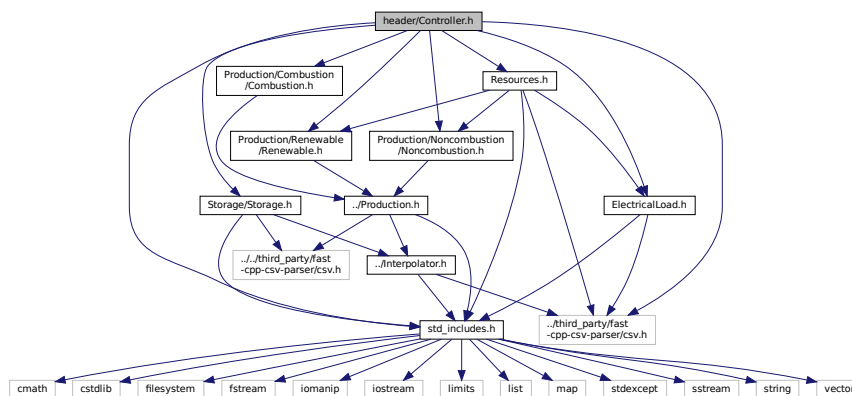
File Documentation

5.1 header/Controller.h File Reference

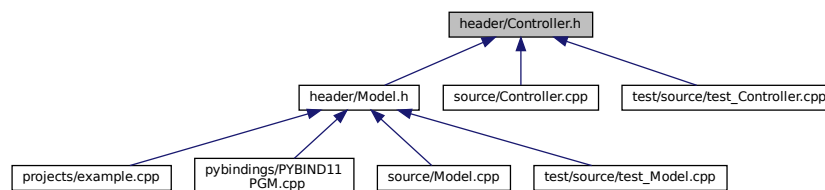
Header file for the [Controller](#) class.

```
#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

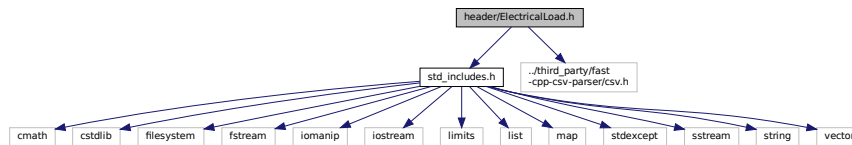
Header file which simply cites the doxygen tool.

Ref: [van Heesch](#). [2023]

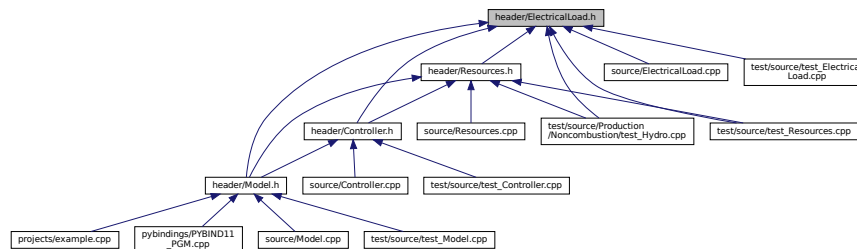
5.3 header/ElectricalLoad.h File Reference

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](#).

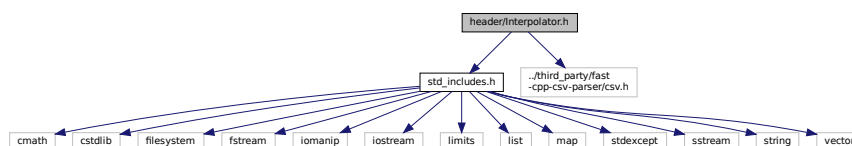
5.3.1 Detailed Description

Header file for the [ElectricalLoad](#) class.

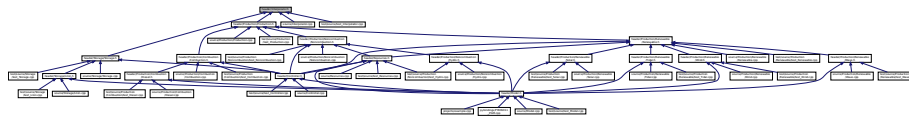
5.4 header/Interpolator.h File Reference

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

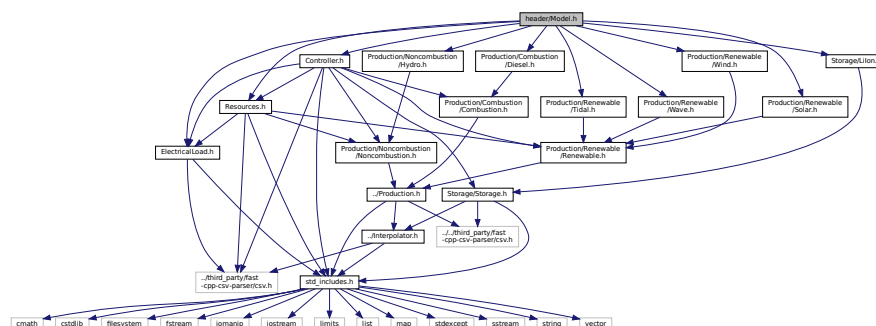
Header file for the [Interpolator](#) class.

5.5 header/Model.h File Reference

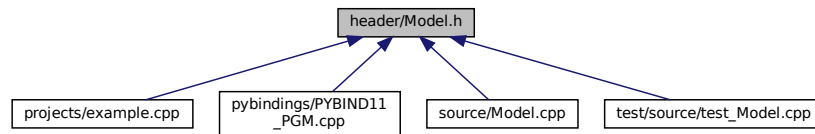
Header file for the [Model](#) class.

```
#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

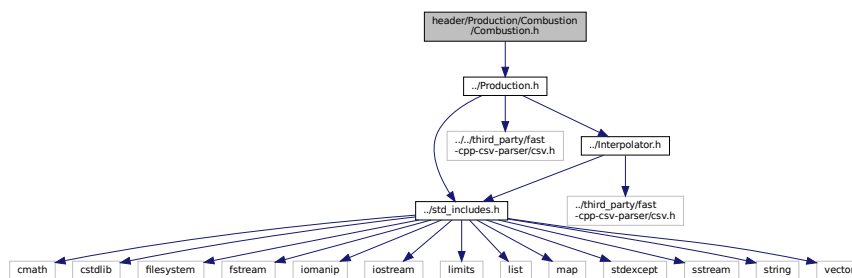
Header file for the [Model](#) class.

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

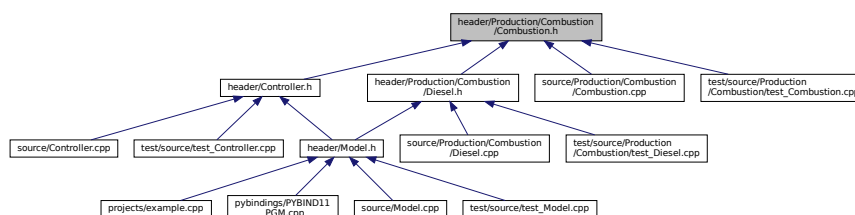
Header file for the [Combustion](#) class.

```
#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

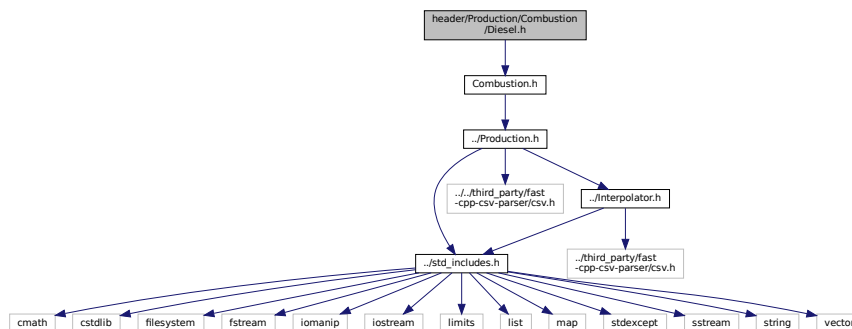
Header file for the [Diesel](#) class.

```

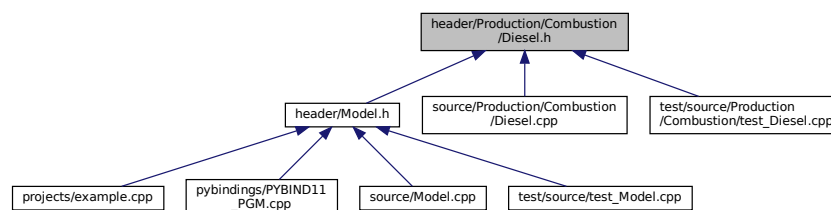
#include "Combustion.h"

```

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.

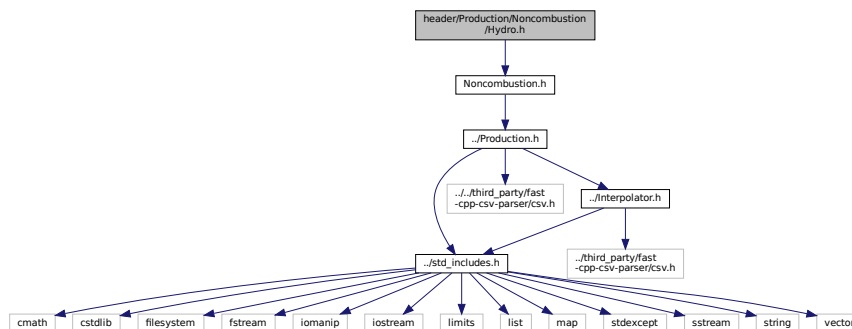
5.7.1 Detailed Description

Header file for the [Diesel](#) class.

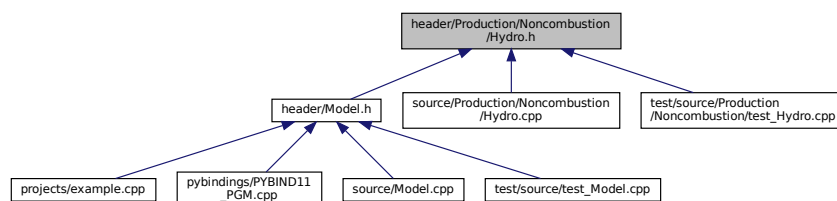
5.8 header/Production/Noncombustion/Hydro.h File Reference

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.

```

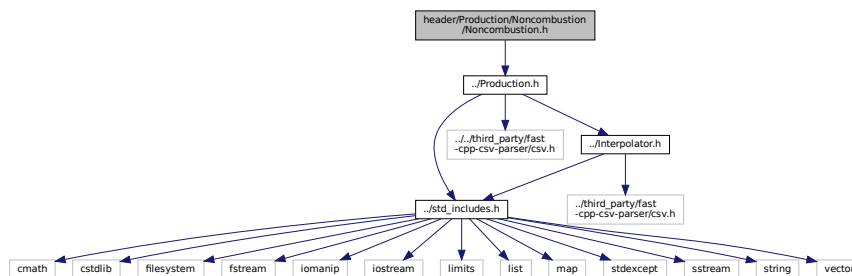
58     {
59         HYDRO_TURBINE_PELTON,
60         HYDRO_TURBINE_FRANCIS,
61         HYDRO_TURBINE_KAPLAN,
62         N_HYDRO_TURBINES
63     };

```

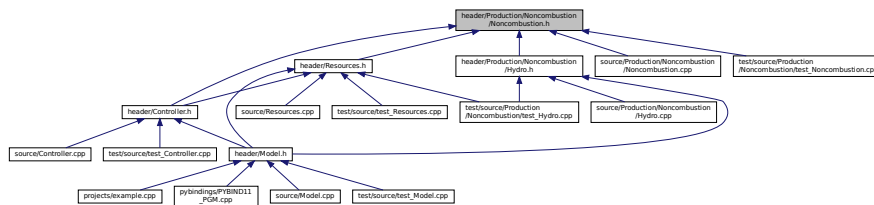
5.9 header/Production/Noncombustion/Noncombustion.h File Reference

```
#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

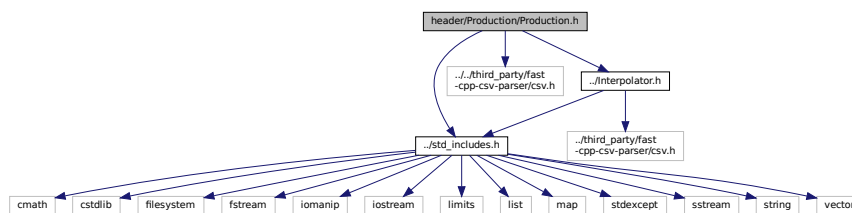
Header file for the `Production` class.

```

#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

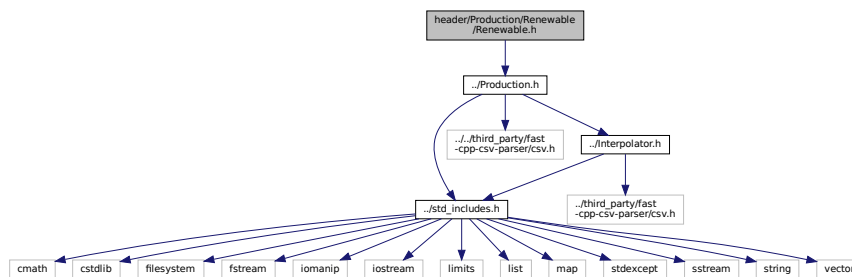
Header file for the [Production](#) class.

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 };

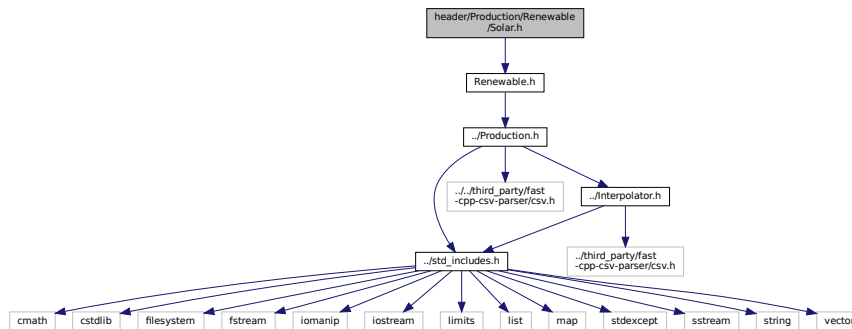
```

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

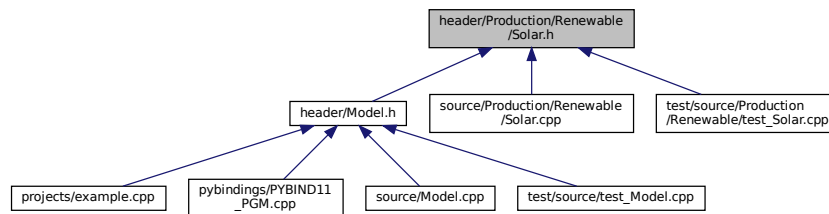
Header file for the `Solar` class.

```
#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.

Enumerations

- enum [SolarPowerProductionModel](#) { [SOLAR_POWER_SIMPLE](#) , [SOLAR_POWER_DETAILED](#) , [N_SOLAR_POWER_PRODUCTION_MODELS](#) }

5.12.1 Detailed Description

Header file for the [Solar](#) class.

5.12.2 Enumeration Type Documentation

5.12.2.1 SolarPowerProductionModel

```
enum SolarPowerProductionModel
```

Enumerator

SOLAR_POWER_SIMPLE	A simple "HOMER-like" power production model.
SOLAR_POWER_DETAILED	A more detailed "PVWatts/SAM-like" production model.
N_SOLAR_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in SolarPowerProductionModel.

```

59                                     {
60     SOLAR_POWER_SIMPLE,
61     SOLAR_POWER_DETAILED,
62     N_SOLAR_POWER_PRODUCTION_MODELS
63 };

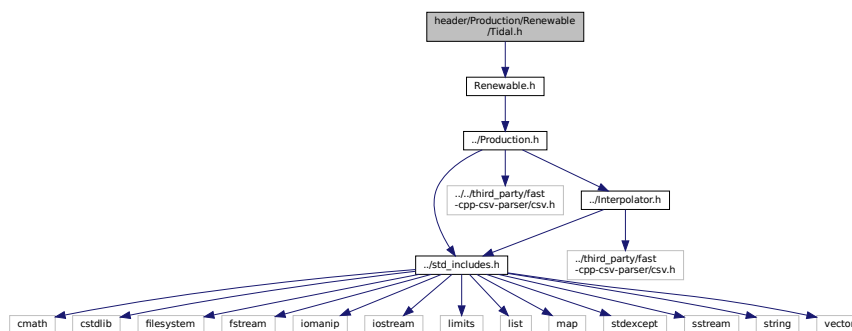
```

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

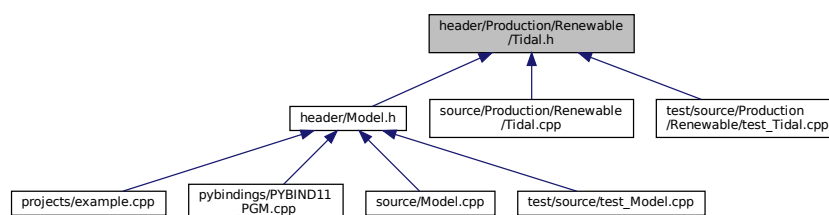
Header file for the [Tidal](#) class.

```
#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

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

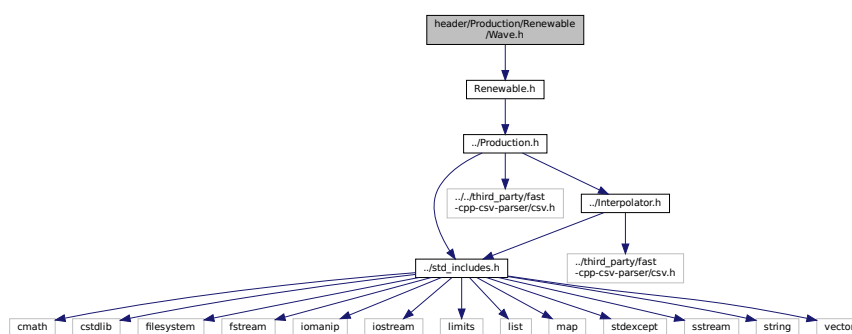
```
59
60     TIDAL_POWER_CUBIC,
61     TIDAL_POWER_EXPONENTIAL,
62     TIDAL_POWER_LOOKUP,
63     N_TIDAL_POWER_PRODUCTION_MODELS
64 };
```

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

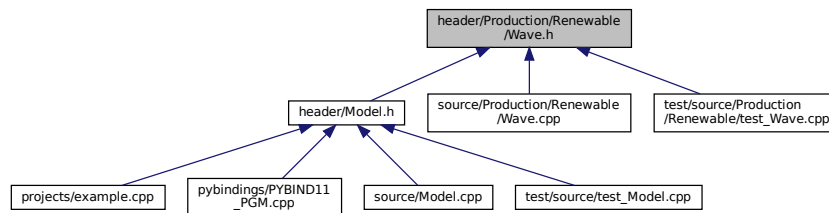
Header file for the `Wave` class.

```
#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 };

```

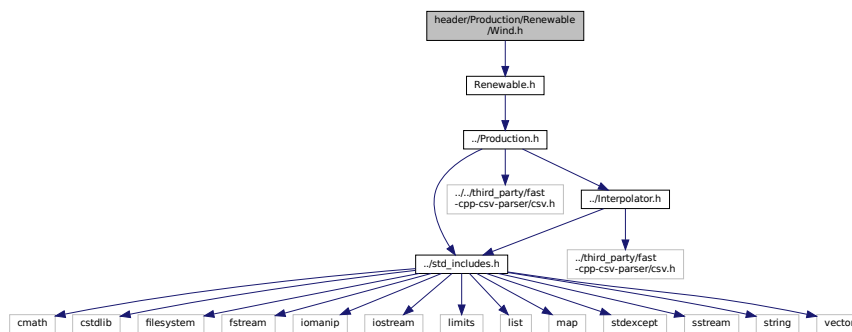
5.15 header/Production/Renewable/Wind.h File Reference

Header file for the [Wind](#) class.

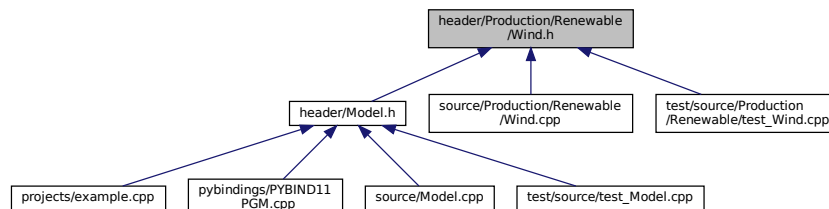
```
#include "Renewable.h"

```

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 .

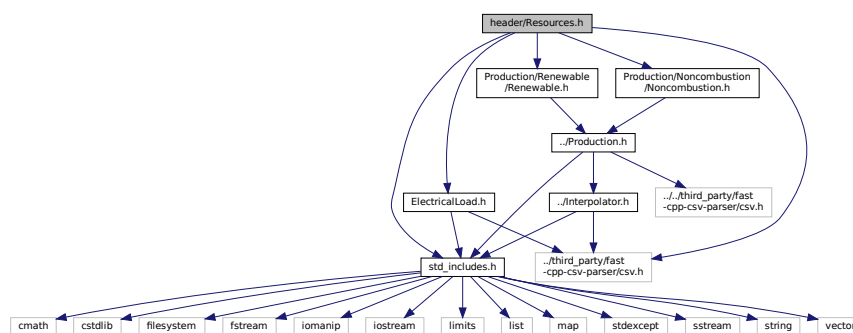
```
59         {
60     WIND_POWER_CUBIC,
61     WIND_POWER_EXPONENTIAL,
62     WIND_POWER_LOOKUP,
63     N_WIND_POWER_PRODUCTION_MODELS
64 };
```

5.16 header/Resources.h File Reference

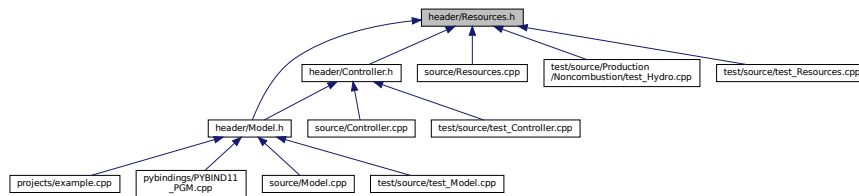
Header file for the [Resources](#) class.

```
#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.

```

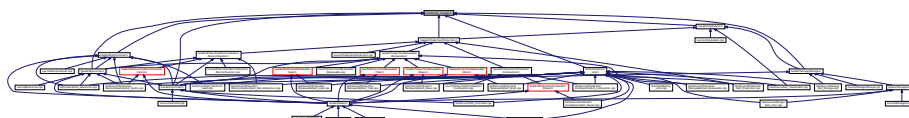
#include <cmath>
#include <cstdlib>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <stdexcept>
#include <sstream>
#include <string>
#include <vector>

```

Include dependency graph for std_includes.h:



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



Macros

- `#define _USE_MATH_DEFINES`

5.17.1 Detailed Description

Header file which simply batches together some standard includes.

5.17.2 Macro Definition Documentation

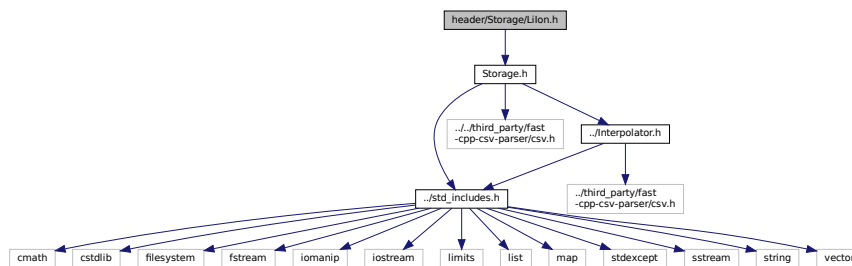
5.17.2.1 _USE_MATH_DEFINES

```
#define _USE_MATH_DEFINES
```

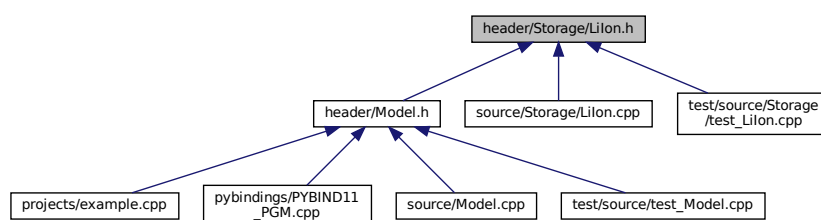
5.18 header/Storage/Lilon.h File Reference

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.

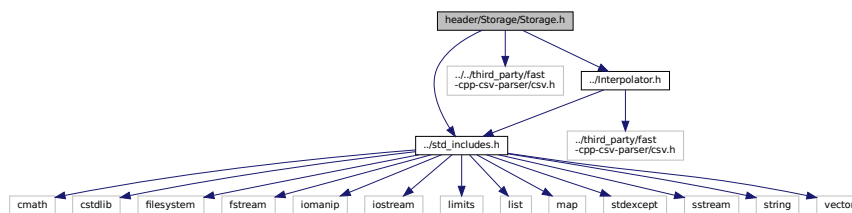
5.18.1 Detailed Description

Header file for the [Lilon](#) class.

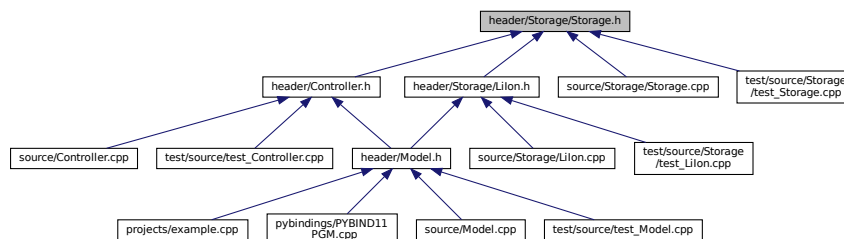
5.19 header/Storage/Storage.h File Reference

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.

Functions

- int `main` (int argc, char **argv)

5.20.1 Function Documentation

5.20.1.1 `main()`

```

int main (
    int argc,
    char ** argv )
51 {
52     /*
53     * 1. construct Model object
54     *
55     * This block constructs a Model object, which is the central container for the
56     * entire microgrid model.
57     *
58     * The first argument that must be provided to the Model constructor is a valid
59     * path (either relative or absolute) to a time series of electrical load data.
60     * For an example of the expected format, see
61     *
62     * data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv
63     *
64     * Note that the length of the given electrical load time series defines the
65     * modelled project life (so if you want to model n years of microgrid operation,
66     * then you must pass a path to n years worth of electrical load data). In addition,
67     * the given electrical load time series defines which points in time are modelled.
68     * As such, all subsequent time series data which is passed in must (1) be of the
69     * same length as the electrical load time series, and (2) provide data for the
70     * same set of points in time. Of course, the electrical load time series can be
71     * of arbitrary length, and it need not be a uniform time series.
72     *
73     * 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     /*
240     * 4. add Hydro object to Model
241     *
242     * This block defines and adds a hydroelectric asset to the Model object.
243     *
244     * In this example, a 300 kW hydroelectric station with a 10,000 m3 reservoir
245     * is defined. The initial reservoir state is set to 50% (so half full), and the
246     * hydroelectric asset is taken to be a sunk asset (so no capital cost incurred
247     * 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

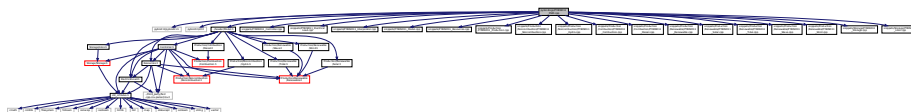
Bindings file for PGMcpp.

```

#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()

```

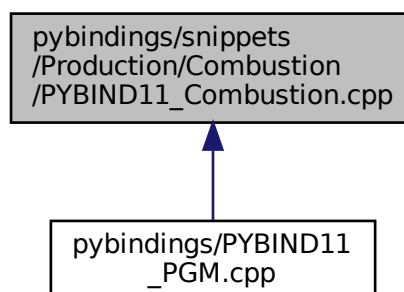
PYBIND11_MODULE (
    PGMcpp ,
    m )
{
56
57
58     #include "snippets/PYBIND11_Controller.cpp"
59     #include "snippets/PYBIND11_ElectricalLoad.cpp"
60     #include "snippets/PYBIND11_Interpolator.cpp"
61     #include "snippets/PYBIND11_Model.cpp"
62     #include "snippets/PYBIND11_Resources.cpp"
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 )
```

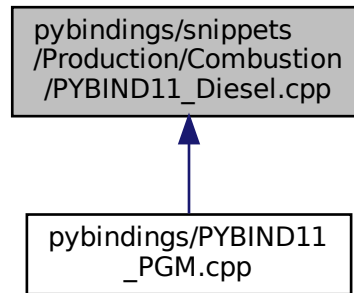
5.22.3 Variable Documentation**5.22.3.1 def_readwrite**

```
&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](#)("CO_emissions_intensity_kgL"
- [&DieselInputs::combustion_inputs](#) &DieselInputs::capital_cost &DieselInputs::fuel_cost_L &DieselInputs::minimum_runtime_hrs &DieselInputs::linear_fuel_intercept_LkWh &DieselInputs::CO_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::CO_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::CO_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::CO_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_kWh & 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 )
```

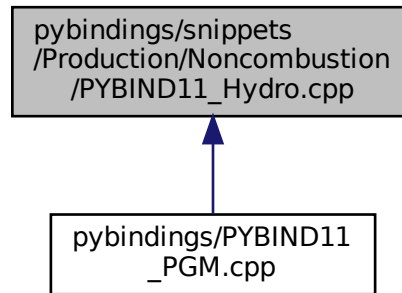
5.23.2.9 def_readwrite() [8/8]

```
& 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 )
```

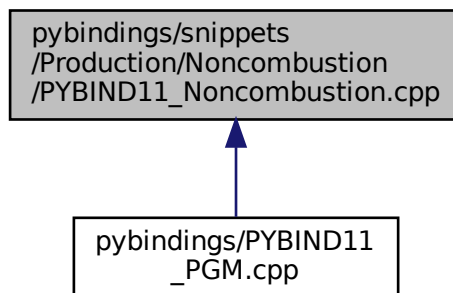
5.24.2.12 value() [2/2]

```
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() )
```

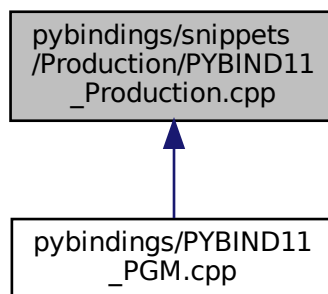
5.25.2.2 value()

```
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
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 (
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 )
```

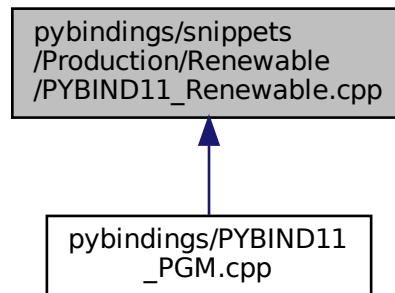
5.26.2.18 def_readwrite() [17/17]

```
& 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())
- [&Renewable::type def_readwrite](#) ("resource_key", &Renewable::resource_key) .def_readwrite("firmness_↵
factor"

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 `def_readwrite()`

```
& Renewable::type def_readwrite (
    "resource_key" ,
    &Renewable::resource_key )
```

5.27.2.3 `value()` [1/2]

```
RenewableType::SOLAR value (
    "TIDAL" ,
    RenewableType::TIDAL )
```

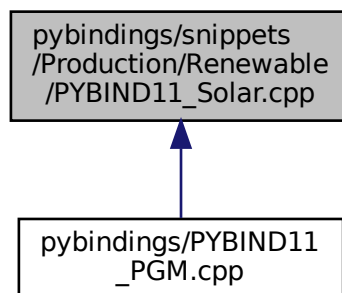
5.27.2.4 `value()` [2/2]

```
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

- `SolarPowerProductionModel::SOLAR_POWER_SIMPLE` value ("SOLAR_POWER_DETAILED", SolarPowerProductionModel::SOLAR_POWER_DETAILED) .value("N_SOLAR_POWER_PRODUCTION_MODELS"
- `&SolarInputs::renewable_inputs` def_readwrite ("resource_key", &SolarInputs::resource_key) .def_readwrite("firmness_factor"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor def_readwrite ("capital_cost", &SolarInputs::capital_cost) .def_readwrite("operation_maintenance_cost_kWh"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor &SolarInputs::operation_maintenance_cost_kWh def_readwrite ("derating", &SolarInputs::derating) .def_readwrite("julian_day"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor &SolarInputs::operation_maintenance_cost_kWh &SolarInputs::julian_day def_readwrite ("latitude_deg", &SolarInputs::latitude_deg) .def_readwrite("longitude_deg"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor &SolarInputs::operation_maintenance_cost_kWh &SolarInputs::julian_day &SolarInputs::longitude_deg def_readwrite ("panel_azimuth_deg", &SolarInputs::panel_azimuth_deg) .def_readwrite("panel_tilt_deg"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor &SolarInputs::operation_maintenance_cost_kWh &SolarInputs::julian_day &SolarInputs::longitude_deg &SolarInputs::panel_tilt_deg def_readwrite ("albedo_ground_reflectance", &SolarInputs::albedo_ground_reflectance) .def_readwrite("power_model"
- `&SolarInputs::renewable_inputs` &SolarInputs::firmness_factor &SolarInputs::operation_maintenance_cost_kWh &SolarInputs::julian_day &SolarInputs::longitude_deg &SolarInputs::panel_tilt_deg &SolarInputs::power_model def (pybind11::init())
- `&Solar::derating` def_readwrite ("power_model", &Solar::power_model) .def_readwrite("power_model_string"

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::firmness_factor & SolarInputs::operation_maintenance_cost_kWh
& SolarInputs::julian_day & SolarInputs::longitude_deg & SolarInputs::panel_tilt_deg & SolarInputs::power_model
def (
    pybind11::init() )
```

5.28.2.2 def_readwrite() [1/7]

```
& SolarInputs::renewable_inputs & SolarInputs::firmness_factor & SolarInputs::operation_maintenance_cost_kWh
& SolarInputs::julian_day & SolarInputs::longitude_deg & SolarInputs::panel_tilt_deg def_readwrite (
    "albedo_ground_reflectance" ,
    &SolarInputs::albedo_ground_reflectance )
```

5.28.2.3 def_readwrite() [2/7]

```
& SolarInputs::renewable_inputs & SolarInputs::firmness_factor def_readwrite (
    "capital_cost" ,
    &SolarInputs::capital_cost )
```

5.28.2.4 def_readwrite() [3/7]

```
& SolarInputs::renewable_inputs & SolarInputs::firmness_factor & SolarInputs::operation_maintenance_cost_kWh
def_readwrite (
    "derating" ,
    &SolarInputs::derating )
```

5.28.2.5 def_readwrite() [4/7]

```
& SolarInputs::renewable_inputs & SolarInputs::firmness_factor & SolarInputs::operation_maintenance_cost_kWh
& SolarInputs::julian_day def_readwrite (
    "latitude_deg" ,
    &SolarInputs::latitude_deg )
```

5.28.2.6 def_readwrite() [5/7]

```
& SolarInputs::renewable_inputs & SolarInputs::firmness_factor & SolarInputs::operation_maintenance_cost_kWh
& SolarInputs::julian_day & SolarInputs::longitude_deg def_readwrite (
    "panel_azimuth_deg" ,
    &SolarInputs::panel_azimuth_deg )
```

5.28.2.7 def_readwrite() [6/7]

```
& Solar::derating def_readwrite (
    "power_model" ,
    &Solar::power_model )
```

5.28.2.8 def_readwrite() [7/7]

```
& SolarInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &SolarInputs::resource_key )
```

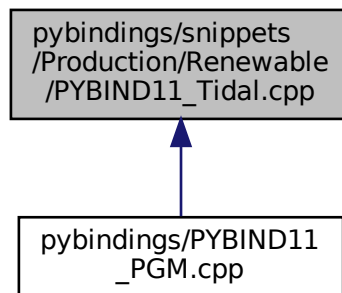
5.28.2.9 value()

```
SolarPowerProductionModel::SOLAR_POWER_SIMPLE value (
    "SOLAR_POWER_DETAILED" ,
    SolarPowerProductionModel::SOLAR_POWER_DETAILED )
```

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](#)("firmness_factor"
- [&TidalInputs::renewable_inputs](#) [&TidalInputs::firmness_factor](#) [def_readwrite](#) ("capital_cost", [&TidalInputs::capital_cost](#)) [.def_readwrite](#)("operation_maintenance_cost_kWh"
- [&TidalInputs::renewable_inputs](#) [&TidalInputs::firmness_factor](#) [&TidalInputs::operation_maintenance_cost_kWh](#) [def_readwrite](#) ("design_speed_ms", [&TidalInputs::design_speed_ms](#)) [.def_readwrite](#)("power_model"
- [&TidalInputs::renewable_inputs](#) [&TidalInputs::firmness_factor](#) [&TidalInputs::operation_maintenance_cost_kWh](#) [&TidalInputs::power_model](#) [def](#) (pybind11::init())
- [&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()`

```
& TidalInputs::renewable_inputs & TidalInputs::firmness_factor & TidalInputs::operation_maintenance_cost_kWh  
& TidalInputs::power_model def (  
    pybind11::init() )
```

5.29.2.2 `def_readwrite()` [1/4]

```
& TidalInputs::renewable_inputs & TidalInputs::firmness_factor def_readwrite (  
    "capital_cost" ,  
    &TidalInputs::capital_cost )
```

5.29.2.3 `def_readwrite()` [2/4]

```
& TidalInputs::renewable_inputs & TidalInputs::firmness_factor & TidalInputs::operation_maintenance_cost_kWh  
def_readwrite (  
    "design_speed_ms" ,  
    &TidalInputs::design_speed_ms )
```

5.29.2.4 `def_readwrite()` [3/4]

```
& Tidal::design_speed_ms def_readwrite (  
    "power_model" ,  
    &Tidal::power_model )
```

5.29.2.5 def_readwrite() [4/4]

```
& TidalInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &TidalInputs::resource_key )
```

5.29.2.6 value() [1/2]

```
TidalPowerProductionModel::TIDAL_POWER_CUBIC TidalPowerProductionModel::TIDAL_POWER_LOOKUP
value (
    "N_TIDAL_POWER_PRODUCTION_MODELS" ,
    TidalPowerProductionModel::N_TIDAL_POWER_PRODUCTION_MODELS )
```

5.29.2.7 value() [2/2]

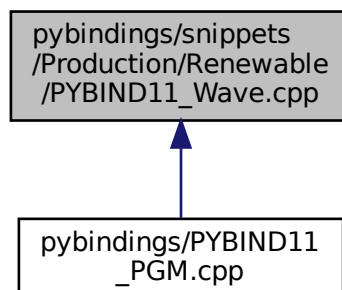
```
TidalPowerProductionModel::TIDAL_POWER_CUBIC value (
    "TIDAL_POWER_EXPONENTIAL" ,
    TidalPowerProductionModel::TIDAL_POWER_EXPONENTIAL )
```

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`("firmness_factor"
- `&WaveInputs::renewable_inputs` `&WaveInputs::firmness_factor` `def_readwrite` ("capital_cost", `&Wave_↵`
`Inputs::capital_cost`) `.def_readwrite`("operation_maintenance_cost_kWh"
- `&WaveInputs::renewable_inputs` `&WaveInputs::firmness_factor` `&WaveInputs::operation_maintenance_cost_kWh`
`def_readwrite` ("design_significant_wave_height_m", `&WaveInputs::design_significant_wave_height_m`)
`.def_readwrite`("design_energy_period_s"
- `&WaveInputs::renewable_inputs` `&WaveInputs::firmness_factor` `&WaveInputs::operation_maintenance_cost_kWh`
`&WaveInputs::design_energy_period_s` `def_readwrite` ("power_model", `&WaveInputs::power_model`) `.def_↵`
`readwrite`("path_2_normalized_performance_matrix"
- `&WaveInputs::renewable_inputs` `&WaveInputs::firmness_factor` `&WaveInputs::operation_maintenance_cost_kWh`
`&WaveInputs::design_energy_period_s` `&WaveInputs::path_2_normalized_performance_matrix` `def` (`pybind11_↵`
`::init()`)
- `&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()`

```
& WaveInputs::renewable_inputs & WaveInputs::firmness_factor & WaveInputs::operation_maintenance_cost_kWh
& WaveInputs::design_energy_period_s & WaveInputs::path_2_normalized_performance_matrix def (
    pybind11::init() )
```

5.30.2.2 `def_readwrite()` [1/5]

```
& WaveInputs::renewable_inputs & WaveInputs::firmness_factor def_readwrite (
    "capital_cost" ,
    &WaveInputs::capital_cost )
```

5.30.2.3 def_readwrite() [2/5]

```
& Wave::design_significant_wave_height_m def_readwrite (
    "design_energy_period_s" ,
    &Wave::design_energy_period_s )
```

5.30.2.4 def_readwrite() [3/5]

```
& WaveInputs::renewable_inputs & WaveInputs::firmness_factor & WaveInputs::operation_maintenance_cost_kWh
def_readwrite (
    "design_significant_wave_height_m" ,
    &WaveInputs::design_significant_wave_height_m )
```

5.30.2.5 def_readwrite() [4/5]

```
& WaveInputs::renewable_inputs & WaveInputs::firmness_factor & WaveInputs::operation_maintenance_cost_kWh
& WaveInputs::design_energy_period_s def_readwrite (
    "power_model" ,
    &WaveInputs::power_model )
```

5.30.2.6 def_readwrite() [5/5]

```
& WaveInputs::renewable_inputs def_readwrite (
    "resource_key" ,
    &WaveInputs::resource_key )
```

5.30.2.7 value() [1/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN WavePowerProductionModel::WAVE_POWER_LOOKUP
value (
    "N_WAVE_POWER_PRODUCTION_MODELS" ,
    WavePowerProductionModel::N_WAVE_POWER_PRODUCTION_MODELS )
```

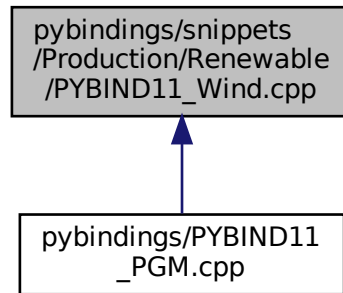
5.30.2.8 value() [2/2]

```
WavePowerProductionModel::WAVE_POWER_GAUSSIAN value (
    "WAVE_POWER_PARABOLOID" ,
    WavePowerProductionModel::WAVE_POWER_PARABOLOID )
```

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_CUBIC](#) [value](#) ("WIND_POWER_EXPONENTIAL", [WindPowerProductionModel::WIND_POWER_EXPONENTIAL](#)) [.value](#)("WIND_POWER_LOOKUP")
- [WindPowerProductionModel::WIND_POWER_CUBIC](#) [WindPowerProductionModel::WIND_POWER_LOOKUP](#) [value](#) ("N_WIND_POWER_PRODUCTION_MODELS", [WindPowerProductionModel::N_WIND_POWER_PRODUCTION_MODELS](#))
- [&WindInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", [&WindInputs::resource_key](#)) [.def_readwrite](#)("firmness_factor")
- [&WindInputs::renewable_inputs](#) [&WindInputs::firmness_factor](#) [def_readwrite](#) ("capital_cost", [&WindInputs::capital_cost](#)) [.def_readwrite](#)("operation_maintenance_cost_kWh")
- [&WindInputs::renewable_inputs](#) [&WindInputs::firmness_factor](#) [&WindInputs::operation_maintenance_cost_kWh](#) [def_readwrite](#) ("design_speed_ms", [&WindInputs::design_speed_ms](#)) [.def_readwrite](#)("power_model")
- [&WindInputs::renewable_inputs](#) [&WindInputs::firmness_factor](#) [&WindInputs::operation_maintenance_cost_kWh](#) [&WindInputs::power_model](#) [def](#) (pybind11::init())
- [&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()

```
& WindInputs::renewable_inputs & WindInputs::firmness_factor & WindInputs::operation_maintenance_cost_kWh  
& WindInputs::power_model def (   
    pybind11::init() )
```

5.31.2.2 def_readwrite() [1/4]

```
& WindInputs::renewable_inputs & WindInputs::firmness_factor def_readwrite (   
    "capital_cost" ,   
    &WindInputs::capital_cost )
```

5.31.2.3 def_readwrite() [2/4]

```
& WindInputs::renewable_inputs & WindInputs::firmness_factor & WindInputs::operation_maintenance_cost_kWh  
def_readwrite (   
    "design_speed_ms" ,   
    &WindInputs::design_speed_ms )
```

5.31.2.4 def_readwrite() [3/4]

```
& Wind::design_speed_ms def_readwrite (   
    "power_model" ,   
    &Wind::power_model )
```

5.31.2.5 def_readwrite() [4/4]

```
& WindInputs::renewable_inputs def_readwrite (   
    "resource_key" ,   
    &WindInputs::resource_key )
```

5.31.2.6 value() [1/2]

```
WindPowerProductionModel::WIND_POWER_CUBIC WindPowerProductionModel::WIND_POWER_LOOKUP value (
    "N_WIND_POWER_PRODUCTION_MODELS" ,
    WindPowerProductionModel::N_WIND_POWER_PRODUCTION_MODELS )
```

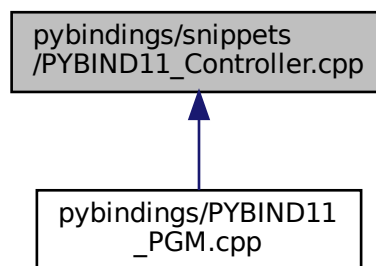
5.31.2.7 value() [2/2]

```
WindPowerProductionModel::WIND_POWER_CUBIC value (
    "WIND_POWER_EXPONENTIAL" ,
    WindPowerProductionModel::WIND_POWER_EXPONENTIAL )
```

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](#)("load↵
_operating_reserve_factor")
- [&Controller::control_mode](#) [&Controller::load_operating_reserve_factor](#) [def_readwrite](#) ("max_operating_↵
reserve_factor", [&Controller::max_operating_reserve_factor](#)) [.def_readwrite](#)("net_load_vec_kW")
- [&Controller::control_mode](#) [&Controller::load_operating_reserve_factor](#) [&Controller::net_load_vec_kW](#)
[def_readwrite](#) ("missed_load_vec_kW", [&Controller::missed_load_vec_kW](#)) [.def_readwrite](#)("combustion↵
_map")
- [&Controller::control_mode](#) [&Controller::load_operating_reserve_factor](#) [&Controller::net_load_vec_kW](#)
[&Controller::combustion_map](#) [def](#) (pybind11::init<>()) [.def](#)("setControlMode")
- [&Controller::control_mode](#) [&Controller::load_operating_reserve_factor](#) [&Controller::net_load_vec_kW](#)
[&Controller::combustion_map](#) [&Controller::setControlMode](#) [def](#) ("init", [&Controller::init](#)) [.def](#)("applyDispatch↵
Control")
- [&Controller::control_mode](#) [&Controller::load_operating_reserve_factor](#) [&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::load_operating_reserve_factor & Controller::net_load_vec_kW
& Controller::combustion_map & Controller::setControlMode & Controller::applyDispatchControl
def (
    "clear" ,
    &Controller::clear )
```

5.32.2.2 `def()` [2/3]

```
& Controller::control_mode & Controller::load_operating_reserve_factor & Controller::net_load_vec_kW
& Controller::combustion_map & Controller::setControlMode def (
    "init" ,
    &Controller::init )
```

5.32.2.3 `def()` [3/3]

```
& Controller::control_mode & Controller::load_operating_reserve_factor & Controller::net_load_vec_kW
& Controller::combustion_map def (
    pybind11::init<> () )
```

5.32.2.4 `def_readwrite()` [1/3]

```
& Controller::control_mode def_readwrite (
    "control_string" ,
    &Controller::control_string )
```

5.32.2.5 def_readwrite() [2/3]

```
& Controller::control_mode & Controller::load_operating_reserve_factor def_readwrite (
    "max_operating_reserve_factor" ,
    &Controller::max_operating_reserve_factor )
```

5.32.2.6 def_readwrite() [3/3]

```
& Controller::control_mode & Controller::load_operating_reserve_factor & Controller::net_load_vec_kW
def_readwrite (
    "missed_load_vec_kW" ,
    &Controller::missed_load_vec_kW )
```

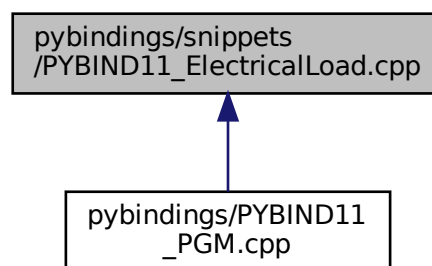
5.32.2.7 value()

```
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_vec_hrs
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 )
```

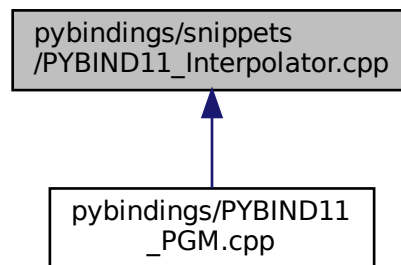

5.33.2.4 def_readwrite() [4/4]

```
& 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::n_points](#) &InterpolatorStruct1D::min_x [def_readwrite](#) ("max_x", &InterpolatorStruct1D::max_x) .[def_readwrite](#) ("y_vec"
- [&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::n_rows](#) &InterpolatorStruct2D::x_vec [def_readwrite](#) ("min_x", &InterpolatorStruct2D::min_x) .[def_readwrite](#) ("max_x"
- [&InterpolatorStruct2D::n_rows](#) &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x [def_readwrite](#) ("y_vec", &InterpolatorStruct2D::y_vec) .[def_readwrite](#) ("min_y"
- [&InterpolatorStruct2D::n_rows](#) &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D::min_y [def_readwrite](#) ("max_y", &InterpolatorStruct2D::max_y) .[def_readwrite](#) ("z_matrix"
- [&Interpolator::interp_map_1D](#) [def_readwrite](#) ("path_map_1D", &Interpolator::path_map_1D) .[def_readwrite](#) ("interp_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 )
```

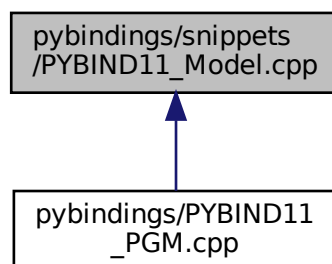
5.34.2.8 `def_readwrite()` [7/7]

```
& 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:

**Functions**

- `&ModelInputs::path_2_electrical_load_time_series def_readwrite` ("control_mode", &ModelInputs::control_mode) `.def_readwrite`("load_operating_reserve_factor"

Variables

- `&ModelInputs::path_2_electrical_load_time_series` `&ModelInputs::load_operating_reserve_factor` `def_↔`
`readwrite("max_operating_reserve_factor", &ModelInputs::max_operating_reserve_factor) .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 Function Documentation

5.35.2.1 `def_readwrite()`

```
& ModelInputs::path_2_electrical_load_time_series def_readwrite (
    "control_mode" ,
    &ModelInputs::control_mode )
```

5.35.3 Variable Documentation

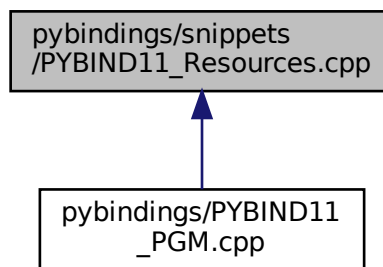
5.35.3.1 `def_readwrite`

```
& ModelInputs::path_2_electrical_load_time_series & ModelInputs::load_operating_reserve_factor
def_readwrite ("max_operating_reserve_factor", &ModelInputs::max_operating_reserve_factor)
.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 )
  
```

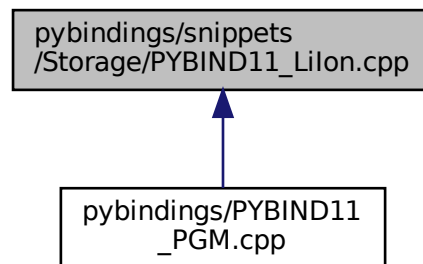
5.36.2.2 def_readwrite() [2/2]

```
& 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_` `_power_capacity_kW"`

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

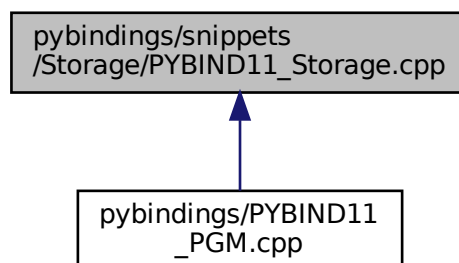
5.37.3.1 `def_readwrite`

```
& 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

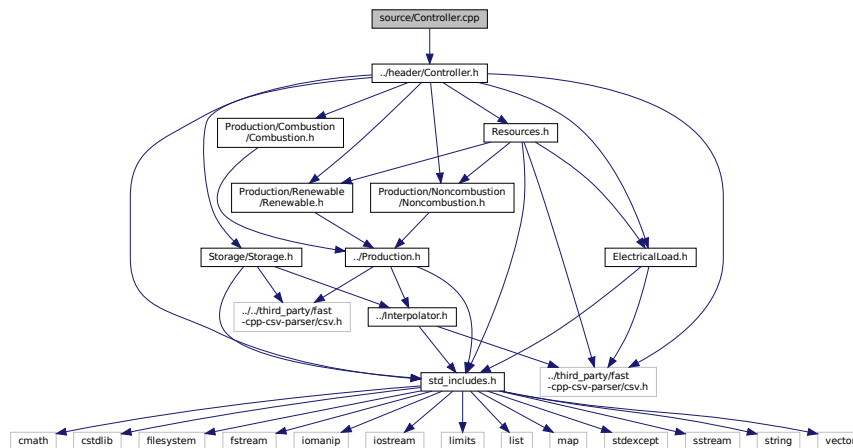
```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW & StorageInputs::nominal_inflation_annual
def_readwrite ("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11
& Storage::type & Storage::print_flag & Storage::is_sunk & Storage::n_replacements & Storage::power_capacity_kW
& 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.

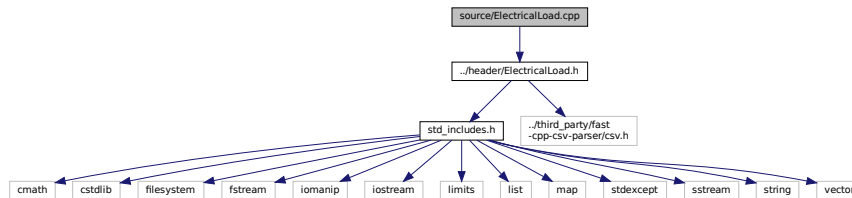
A class which contains a various dispatch control logic. Intended to serve as a component class of [Controller](#).

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.

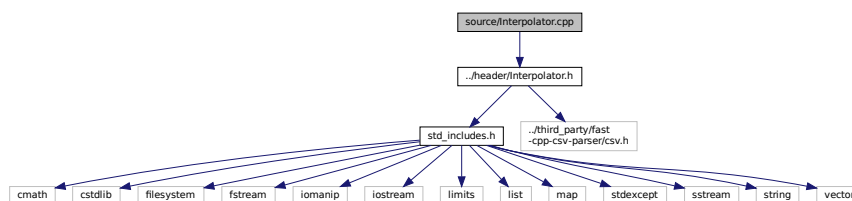
A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

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.

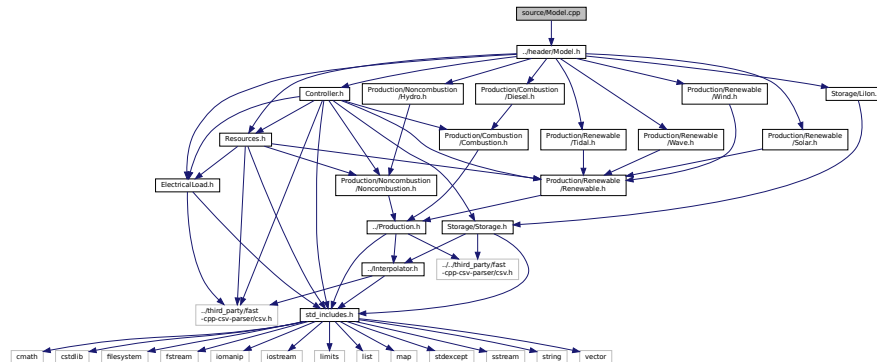
A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

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.

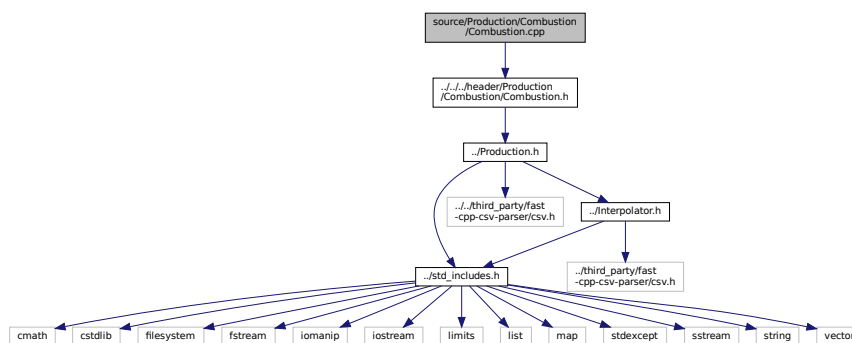
A container class which forms the centre of PGMcpp. The [Model](#) class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

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

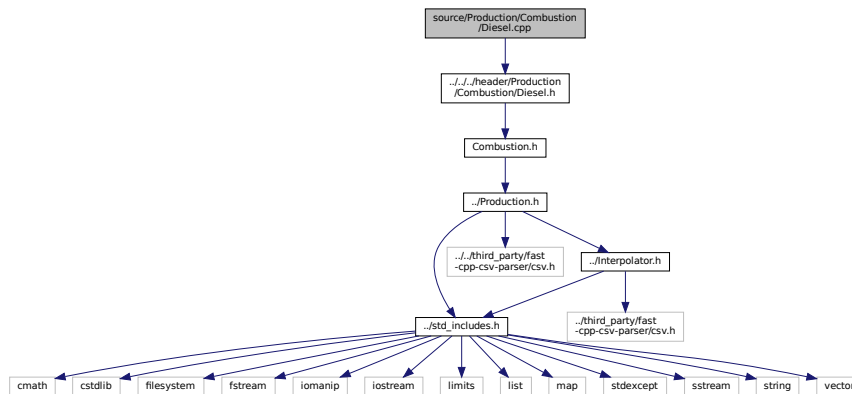
Implementation file for the [Combustion](#) class.

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.

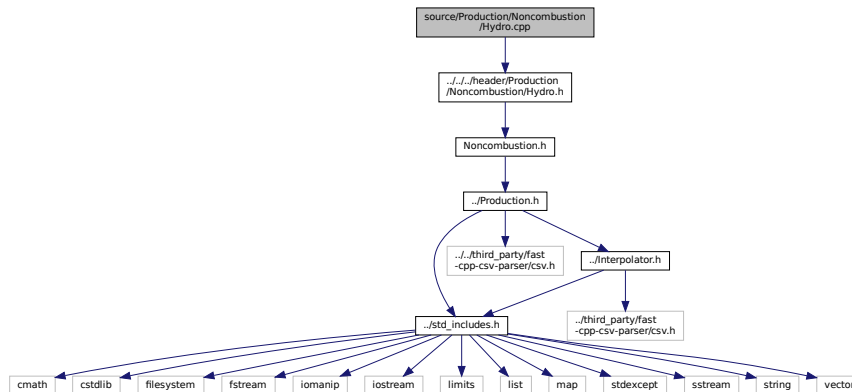
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

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.

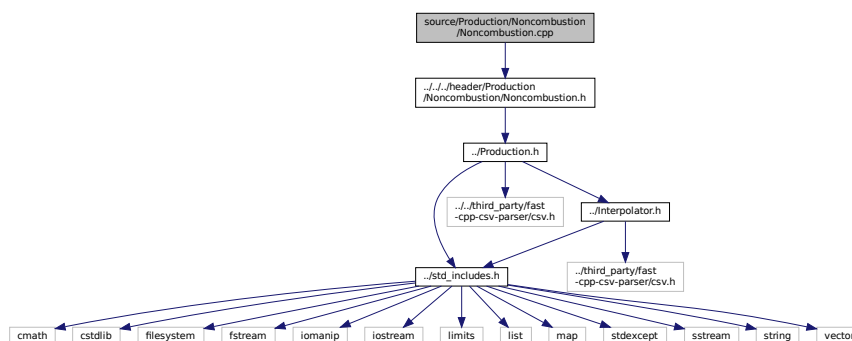
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

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.

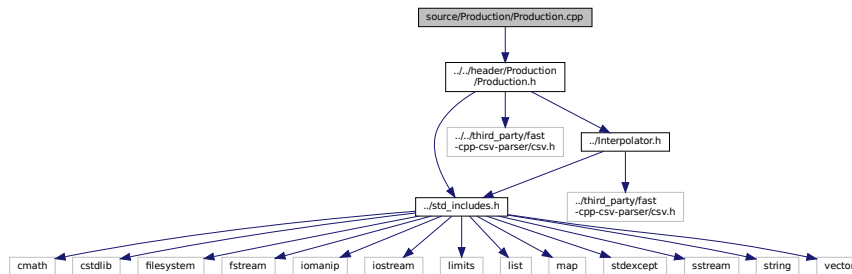
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

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.

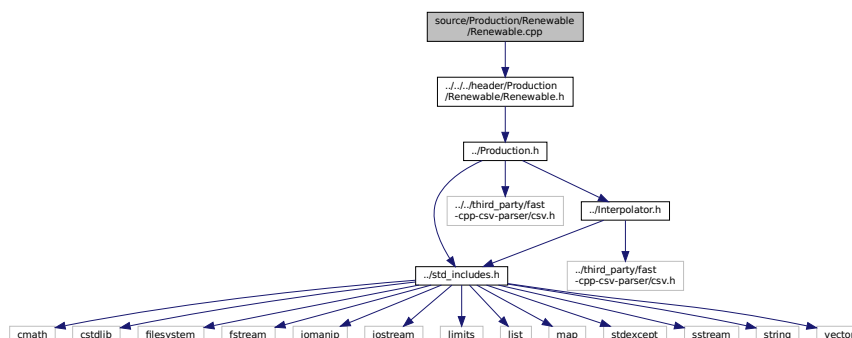
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

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.

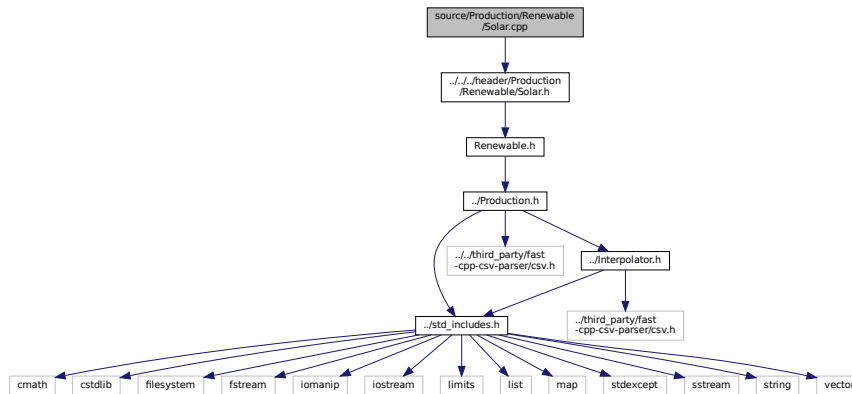
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

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.

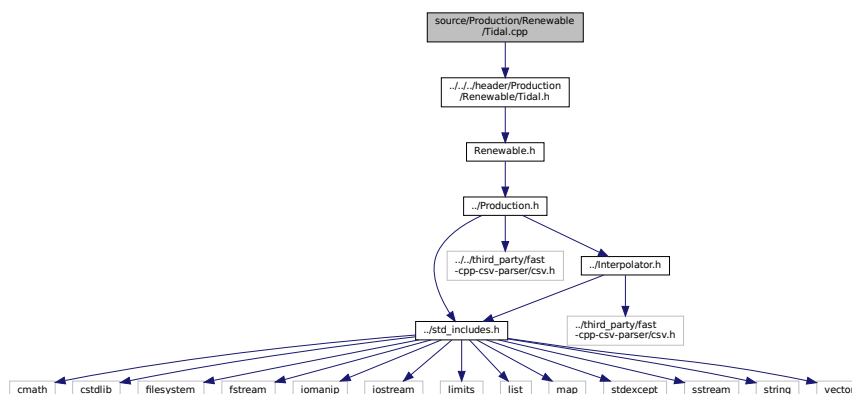
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

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

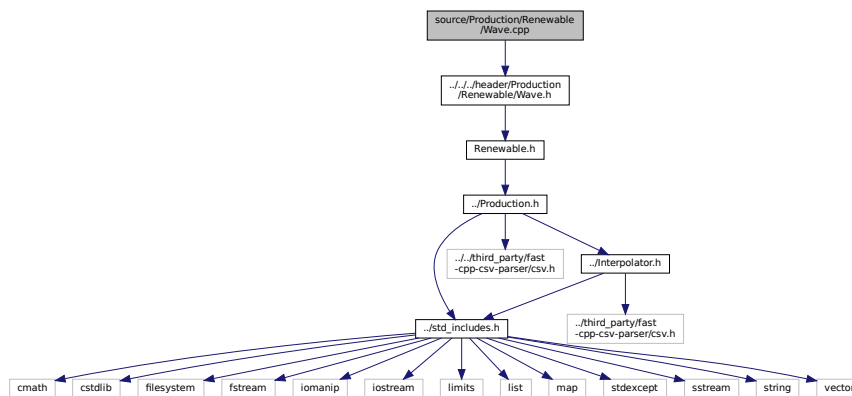
Implementation file for the [Tidal](#) class.

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.

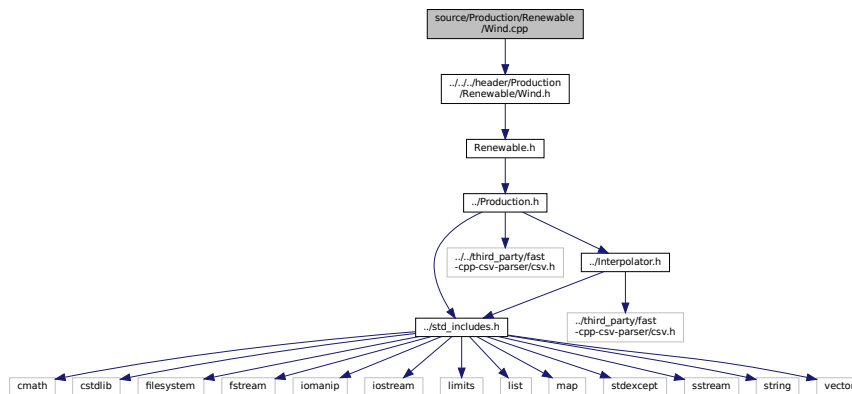
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

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.

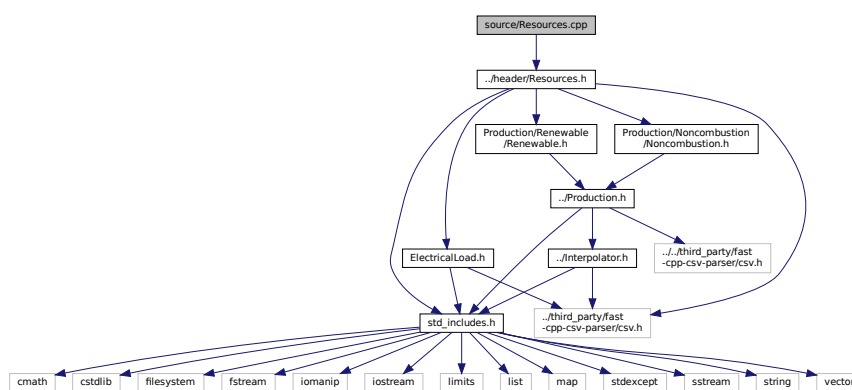
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

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.

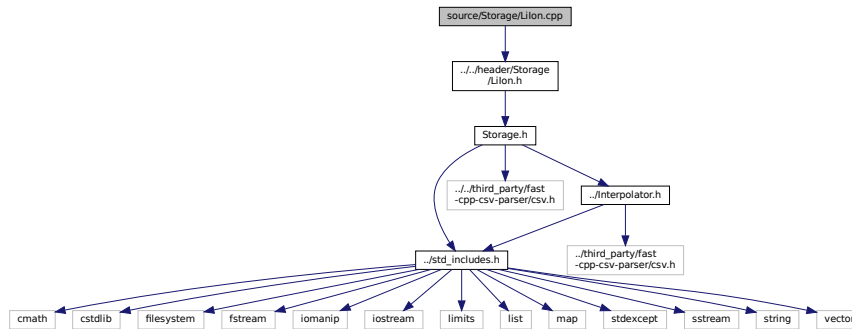
A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

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.

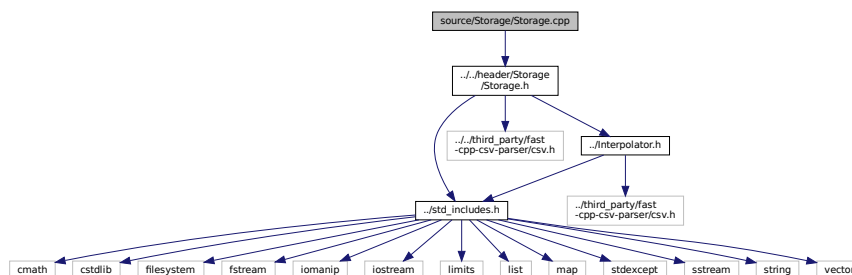
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

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

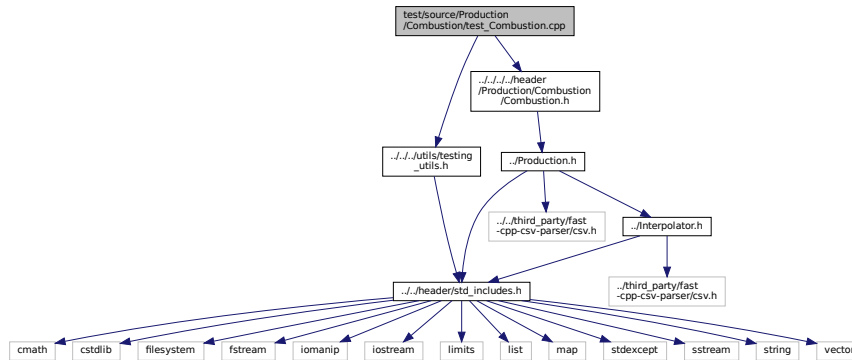
Implementation file for the [Storage](#) class.

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     #ifdef _WIN32
155         std::cout << std::endl;
156     #endif
157
158     srand(time(NULL));
159
160
161     std::vector<double> time_vec_hrs (8760, 0);
162     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
163         time_vec_hrs[i] = i;
164     }
165
166     Combustion* test_combustion_ptr = testConstruct_Combustion(&time_vec_hrs);
167
168
169     try {
170         //...
171     }
172
173
174     catch (...) {
175         delete test_combustion_ptr;
176
177         printGold(" ..... ");
178         printRed("FAIL");
179         std::cout << std::endl;
180         throw;
181     }
182
183
184     delete test_combustion_ptr;
185
186     printGold(" ..... ");
187     printGreen("PASS");
188     std::cout << std::endl;
189     return 0;
190
191 } /* 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

<i>time_vec_hrs_ptr</i>	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

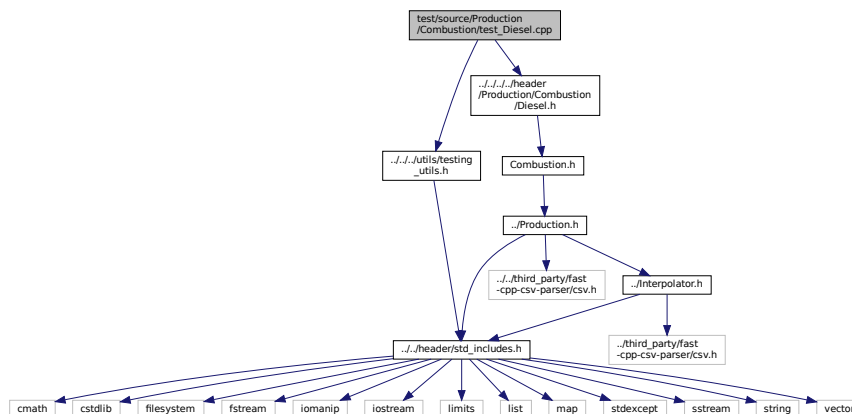
Testing suite for [Diesel](#) class.

```

#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     #ifdef _WIN32
738         std::cout << std::endl;
739     #endif
740
741     srand(time(NULL));
742
743
744     std::vector<double> time_vec_hrs (8760, 0);
745     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
746         time_vec_hrs[i] = i;
747     }
748
749
750     Combustion* test_diesel_ptr = testConstruct_Diesel(&time_vec_hrs);
751     Combustion* test_diesel_lookup_ptr = testConstructLookup_Diesel(&time_vec_hrs);
752
753     try {
754         testBadConstruct_Diesel(&time_vec_hrs);
755
756         testCapacityConstraint_Diesel(test_diesel_ptr);
757         testMinimumLoadRatioConstraint_Diesel(test_diesel_ptr);
758
759         testCommit_Diesel(test_diesel_ptr);
760
761         testMinimumRuntimeConstraint_Diesel(test_diesel_ptr);
762
763         testFuelConsumptionEmissions_Diesel(test_diesel_ptr);
764         testEconomics_Diesel(test_diesel_ptr);
765
766         testFuelLookup_Diesel(test_diesel_lookup_ptr);
767     }
768
769
770     catch (...) {
771         delete test_diesel_ptr;
772         delete test_diesel_lookup_ptr;
773
774         printGold(" ..... ");
775         printRed("FAIL");
776         std::cout << std::endl;
777         throw;
778     }
779
780
781     delete test_diesel_ptr;
782     delete test_diesel_lookup_ptr;
783
784     printGold(" ..... ");
785     printGreen("PASS");
786     std::cout << std::endl;
787     return 0;
788
789 } /* 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

<code>time_vec_hrs_ptr</code>	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

<code>test_diesel_ptr</code>	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, 1, 1, 0,
309         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
310         1, 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.
------------------------------	---

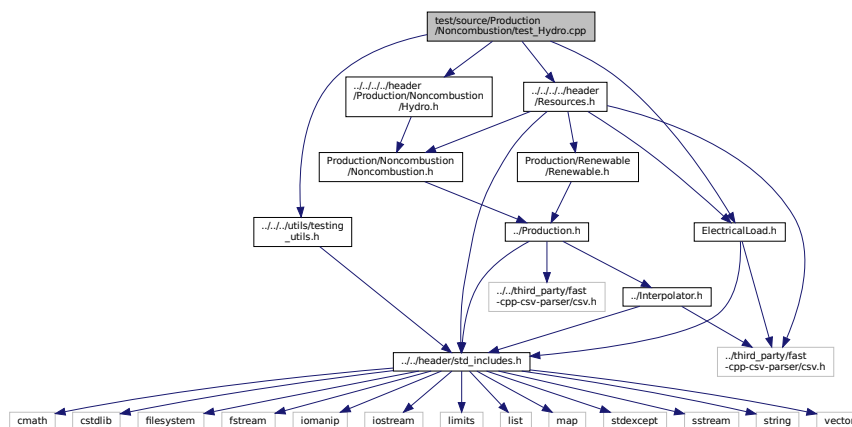
```
405 {
406     std::vector<double> load_vec_kW = {
407         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
408         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
409         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
410         1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
411     };
412
413     std::vector<bool> expected_is_running_vec = {
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     #ifdef _WIN32
```

```

338         std::cout << std::endl;
339     #endif
340
341     srand(time(NULL));
342
343
344     std::vector<double> time_vec_hrs (8760, 0);
345     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
346         time_vec_hrs[i] = i;
347     }
348
349     std::string path_2_electrical_load_time_series =
350         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
351
352     ElectricalLoad* test_electrical_load_ptr =
353         new ElectricalLoad(path_2_electrical_load_time_series);
354
355     Resources* test_resources_ptr = new Resources();
356
357     HydroInputs hydro_inputs;
358     int hydro_resource_key = 0;
359
360     hydro_inputs.reservoir_capacity_m3 = 10000;
361     hydro_inputs.resource_key = hydro_resource_key;
362
363     Noncombustion* test_hydro_ptr = testConstruct_Hydro(hydro_inputs, &time_vec_hrs);
364
365     std::string path_2_hydro_resource_data =
366         "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
367
368     test_resources_ptr->addResource(
369         NoncombustionType::HYDRO,
370         path_2_hydro_resource_data,
371         hydro_resource_key,
372         test_electrical_load_ptr
373     );
374
375
376     try {
377         testEfficiencyInterpolation_Hydro(test_hydro_ptr);
378         testCommit_Hydro(test_hydro_ptr, test_resources_ptr);
379     }
380
381     catch (...) {
382         delete test_electrical_load_ptr;
383         delete test_resources_ptr;
384         delete test_hydro_ptr;
385
386         printGold(" ... ");
387         printRed("FAIL");
388         std::cout << std::endl;
389         throw;
390     }
391
392
393     delete test_electrical_load_ptr;
394     delete test_resources_ptr;
395     delete test_hydro_ptr;
396
397     printGold(" ... ");
398     printGreen("PASS");
399     std::cout << std::endl;
400     return 0;
401 }
402
403 } /* 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() */

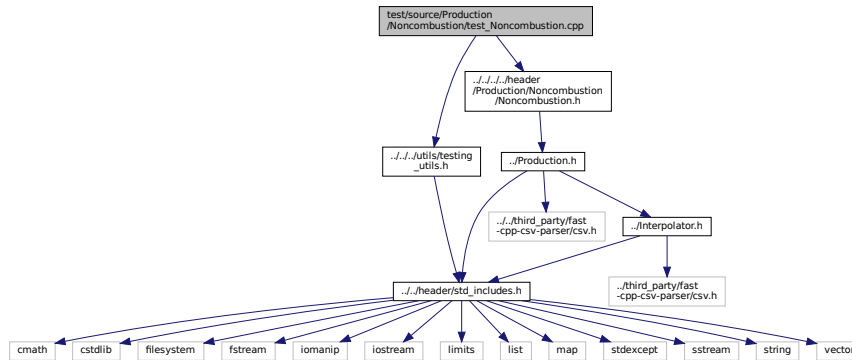
```

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     #ifdef _WIN32
107         std::cout << std::endl;
108     #endif
109
110     srand(time(NULL));
111
112
113     std::vector<double> time_vec_hrs (8760, 0);
114     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
115         time_vec_hrs[i] = i;
116     }
117
118     Noncombustion* test_noncombustion_ptr = testConstruct_Noncombustion(&time_vec_hrs);
119
120
121     try {
122         //...
123     }
124
125
126     catch (...) {
127         delete test_noncombustion_ptr;
128
129         printGold(" ..... ");
130         printRed("FAIL");
131         std::cout << std::endl;
132         throw;
133     }
134
135
136     delete test_noncombustion_ptr;
137
138     printGold(" ..... ");
139     printGreen("PASS");
140     std::cout << std::endl;
141     return 0;
142
143 } /* 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

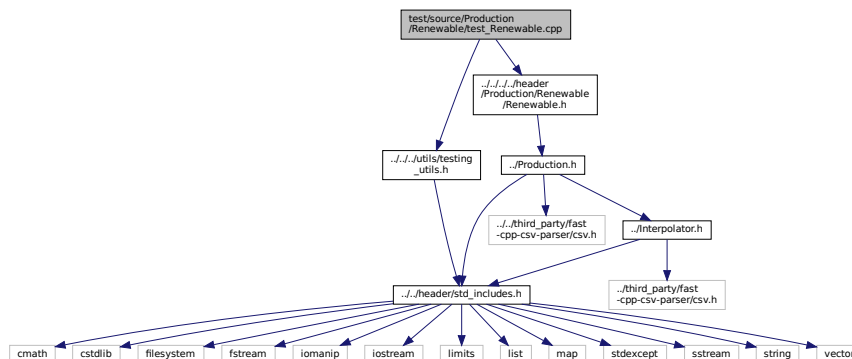
Testing suite for [Renewable](#) class.

```

#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     #ifdef _WIN32
106         std::cout << std::endl;
107     #endif
108
109     srand(time(NULL));
110
111
112     std::vector<double> time_vec_hrs (8760, 0);
113     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
114         time_vec_hrs[i] = i;
115     }
116
117     Renewable* test_renewable_ptr = testConstruct_Renewable(&time_vec_hrs);
118
119
120     try {
121         //...
122     }
123
124
125     catch (...) {
126         delete test_renewable_ptr;
127
128         printGold(" ..... ");
129         printRed("FAIL");
130         std::cout << std::endl;
131         throw;
132     }
133
134
135     delete test_renewable_ptr;
136
137     printGold(" ..... ");
138     printGreen("PASS");
139     std::cout << std::endl;
140     return 0;
141
142 } /* 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() */

```

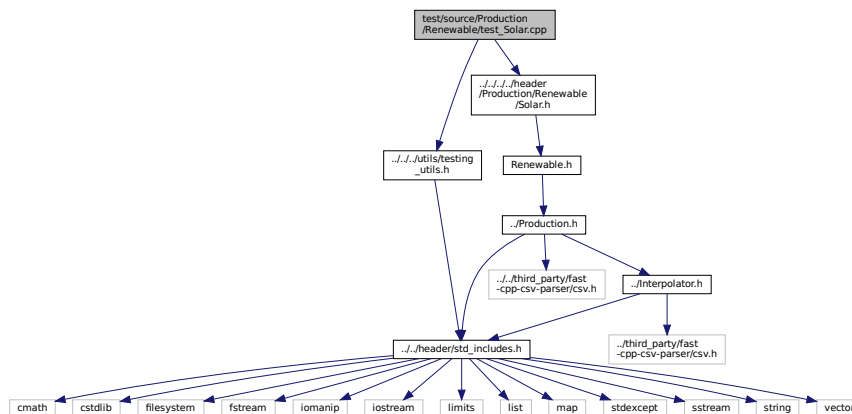
5.61 test/source/Production/Renewable/test_Solar.cpp File Reference

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 `testDetailed_Solar` (void)
- 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");

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

    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
        );

        testDetailed_Solar();

        testProductionConstraint_Solar(test_solar_ptr);

        testCommit_Solar(test_solar_ptr);
        testEconomics_Solar(test_solar_ptr);
    }
}
```

```

712     }
713
714
715     catch (...) {
716         delete test_solar_ptr;
717
718         printGold(" ..... ");
719         printRed("FAIL");
720         std::cout << std::endl;
721         throw;
722     }
723
724
725     delete test_solar_ptr;
726
727     printGold(" ..... ");
728     printGreen("PASS");
729     std::cout << std::endl;
730     return 0;
731
732 } /* 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.
-------------------------	---

```

141 {
142     bool error_flag = true;
143
144     try {
145         SolarInputs bad_solar_inputs;
146         bad_solar_inputs.derating = -1;
147
148         Solar bad_solar(8760, 1, bad_solar_inputs, time_vec_hrs_ptr);
149
150         error_flag = false;
151     } catch (...) {
152         // Task failed successfully! =P
153     }
154     if (not error_flag) {
155         expectedErrorNotDetected(__FILE__, __LINE__);
156     }
157
158     return;
159 } /* 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.
-----------------------	---

```

522 {
523     std::vector<double> dt_vec_hrs (48, 1);
524
525     std::vector<double> load_vec_kW = {
526         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
527         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
528         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
529         1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
530     };
531
532     double load_kW = 0;
533     double production_kW = 0;
534     double roll = 0;
535     double solar_resource_kWm2 = 0;
536
537     for (int i = 0; i < 48; i++) {
538         roll = (double)rand() / RAND_MAX;
539
540         solar_resource_kWm2 = roll;
541
542         roll = (double)rand() / RAND_MAX;
543
544         if (roll <= 0.1) {
545             solar_resource_kWm2 = 0;
546         }
547
548         else if (roll >= 0.95) {
549             solar_resource_kWm2 = 1.25;
550         }
551
552         roll = (double)rand() / RAND_MAX;
553
554         if (roll >= 0.95) {
555             roll = 1.25;
556         }
557
558         load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
559         load_kW = load_vec_kW[i];
560
561         production_kW = test_solar_ptr->computeProductionkW(
562             i,
563             dt_vec_hrs[i],
564             solar_resource_kWm2
565         );
566
567         load_kW = test_solar_ptr->commit(
568             i,
569             dt_vec_hrs[i],
570             production_kW,
571             load_kW
572         );
573
574         // is running (or not) as expected
575         if (solar_resource_kWm2 > 0) {
576             testTruth(
577                 test_solar_ptr->is_running,
578                 __FILE__,
579                 __LINE__
580             );
581         }
582
583         else {
584             testTruth(
585                 not test_solar_ptr->is_running,
586                 __FILE__,
587                 __LINE__
588             );
589         }
590
591         // load_kW <= load_vec_kW (i.e., after vs before)
592         testLessThanOrEqualTo(
593             load_kW,
594             load_vec_kW[i],
595             __FILE__,
596             __LINE__
597         );
598
599         // production = dispatch + storage + curtailment
600         testFloatEquals(
601             test_solar_ptr->production_vec_kW[i] -
602             test_solar_ptr->dispatch_vec_kW[i] -
603             test_solar_ptr->storage_vec_kW[i] -
604             test_solar_ptr->curtailment_vec_kW[i],
605             0,
606             __FILE__,
607             __LINE__
608         );

```

```

609
610     // capacity constraint
611     if (solar_resource_kWm2 > 1) {
612         testFloatEquals(
613             test_solar_ptr->production_vec_kW[i],
614             test_solar_ptr->capacity_kW,
615             __FILE__,
616             __LINE__
617         );
618     }
619 }
620
621 return;
622 } /* 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     testFloatEquals(
116         test_solar_ptr->firmness_factor,
117         0.2,
118         __FILE__,
119         __LINE__
120     );
121
122     return test_solar_ptr;
123 } /* testConstruct_Solar() */

```

5.61.2.5 testDetailed_Solar()

```

void testDetailed_Solar (
    void )
293 {
294     // init time and solar resource vectors
295     std::vector<double> time_vec_hrs = {
296         0,
297         1,
298         2,
299         3,
300         4,
301         5,
302         6,
303         7,
304         8,
305         9,
306         10,
307         11,
308         12,
309         13,
310         14,
311         15,
312         16,
313         17,
314         18,
315         19,
316         20,
317         21,
318         22,
319         23
320     };
321
322     std::vector<double> 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     };
348
349     // init expected results (simple and detailed)

```

```

350     std::vector<double> expected_simple_production_vec_kW = {
351         0,
352         0,
353         0,
354         0,
355         0,
356         0,
357         0.00681362130147212,
358         0.0278673253636,
359         0.1710349828744,
360         0.32798908906576,
361         0.07977081844424,
362         0.7627622099725601,
363         1.83423974324928,
364         1.088573722352,
365         0.20281073022008,
366         0.41655180126568,
367         0.45021269188776,
368         0.0560949377772,
369         0.0136958616712,
370         0,
371         0,
372         0,
373         0,
374         0
375     };
376
377     std::vector<double> expected_detailed_production_vec_kW = {
378         0,
379         0,
380         0,
381         0,
382         0,
383         0,
384         0.007338124437333107,
385         0.03001323298400045,
386         0.1842098680357352,
387         0.3532627387497894,
388         0.085919752082476,
389         0.8215778242841695,
390         1.975723895381408,
391         1.17256966118828,
392         0.2184652818009985,
393         0.4487156859620408,
394         0.4849877212456633,
395         0.06042929047364313,
396         0.01475448450756636,
397         0,
398         0,
399         0,
400         0,
401         0
402     };
403
404     // init Solar (simple)
405     SolarInputs solar_inputs;
406
407     Solar test_solar_simple(
408         time_vec_hrs.size(),
409         1,
410         solar_inputs,
411         &time_vec_hrs
412     );
413
414     // init Solar (detailed)
415     solar_inputs.power_model = SolarPowerProductionModel :: SOLAR_POWER_DETAILED;
416
417     solar_inputs.julian_day = 8766;
418     solar_inputs.latitude_deg = 50;
419     solar_inputs.longitude_deg = -125;
420     solar_inputs.panel_azimuth_deg = 180;
421     solar_inputs.panel_tilt_deg = 30;
422     solar_inputs.albedo_ground_reflectance = 0.5;
423
424     Solar test_solar_detailed(
425         time_vec_hrs.size(),
426         1,
427         solar_inputs,
428         &time_vec_hrs
429     );
430
431     // test simple production
432     double production_kW = 0;
433
434     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
435         production_kW = test_solar_simple.computeProductionkW(
436             i, 1, solar_resource_vec_kWm2[i]

```

```

437         );
438
439         test_solar_simple.commit(
440             i, 1, production_kW, 100
441         );
442
443         testFloatEquals(
444             production_kW,
445             expected_simple_production_vec_kW[i],
446             __FILE__,
447             __LINE__
448         );
449     }
450
451     // test detailed production
452     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
453         production_kW = test_solar_detailed.computeProductionkW(
454             i, 1, solar_resource_vec_kWm2[i]
455         );
456
457         test_solar_detailed.commit(
458             i, 1, production_kW, 100
459         );
460
461         testFloatEquals(
462             production_kW,
463             expected_detailed_production_vec_kW[i],
464             __FILE__,
465             __LINE__
466         );
467     }
468
469 } /* testDetailed_Solar() */

```

5.61.2.6 testEconomics_Solar()

```

void testEconomics_Solar (
    Renewable * test_solar_ptr )
{
    640 {
    641     for (int i = 0; i < 48; i++) {
    642         // resource, O&M > 0 whenever solar is running (i.e., producing)
    643         if (test_solar_ptr->is_running_vec[i]) {
    644             testGreaterThan(
    645                 test_solar_ptr->operation_maintenance_cost_vec[i],
    646                 0,
    647                 __FILE__,
    648                 __LINE__
    649             );
    650         }
    651
    652         // resource, O&M = 0 whenever solar is not running (i.e., not producing)
    653         else {
    654             testFloatEquals(
    655                 test_solar_ptr->operation_maintenance_cost_vec[i],
    656                 0,
    657                 __FILE__,
    658                 __LINE__
    659             );
    660         }
    661     }
    662
    663     return;
    664 } /* testEconomics_Solar() */

```

5.61.2.7 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.
-----------------------	---

```

487 {
488     testFloatEquals(
489         test_solar_ptr->computeProductionkW(0, 1, 2),
490         100,
491         __FILE__,
492         __LINE__
493     );
494
495     testFloatEquals(
496         test_solar_ptr->computeProductionkW(0, 1, -1),
497         0,
498         __FILE__,
499         __LINE__
500     );
501
502     return;
503 } /* testProductionConstraint_Solar() */

```

5.61.2.8 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.

```

186 {
187     SolarInputs solar_inputs;
188
189     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
190         path_2_normalized_production_time_series;
191
192     Solar test_solar_override(
193         time_vec_hrs_ptr->size(),
194         1,
195         solar_inputs,
196         time_vec_hrs_ptr
197     );
198
199
200     std::vector<double> expected_normalized_production_vec = {
201         0.916955708517556,
202         0.90947506148393,
203         0.38425267564517,
204         0.191510884037643,
205         0.803361391862077,
206         0.261511294927198,
207         0.221944653883198,
208         0.858495335855501,
209         0.0162863861443092,
210         0.774345409915512,
211         0.354898664149867,
212         0.11158009453439,
213         0.191670176408956,
214         0.0149072402795702,
215         0.30174228469322,
216         0.0815062957850151,
217         0.776404660266821,
218         0.207069187162109,

```

```

219         0.518926216750454,
220         0.148538109788597,
221         0.443035200791027,
222         0.62119079547209,
223         0.270792717524391,
224         0.761074879460849,
225         0.0545251308358993,
226         0.0895417089500092,
227         0.21787190761933,
228         0.834403724509682,
229         0.908807953036246,
230         0.815888965292123,
231         0.416663215314571,
232         0.523649705576525,
233         0.490890480401437,
234         0.28317138282312,
235         0.877382682055847,
236         0.14972090597986,
237         0.480161632646382,
238         0.0655830129932816,
239         0.41802666403448,
240         0.48692477737368,
241         0.275957323208066,
242         0.228651250718341,
243         0.574371311550247,
244         0.251872481275769,
245         0.802697508767121,
246         0.00130607304363551,
247         0.481240172488057,
248         0.702527508293784
249     };
250
251     for (size_t i = 0; i < expected_normalized_production_vec.size(); i++) {
252         testFloatEquals(
253             test_solar_override.normalized_production_vec[i],
254             expected_normalized_production_vec[i],
255             __FILE__,
256             __LINE__
257         );
258
259         testFloatEquals(
260             test_solar_override.computeProductionkW(i, rand(), rand()),
261             test_solar_override.capacity_kW * expected_normalized_production_vec[i],
262             __FILE__,
263             __LINE__
264         );
265     }
266
267     return;
268 } /* testProductionOverride_Solar() */

```

5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference

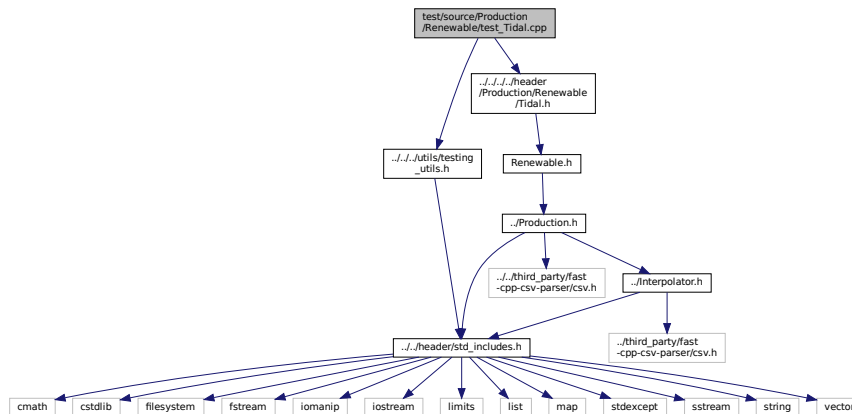
Testing suite for [Tidal](#) class.

```

#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 )
359 {
360     #ifdef _WIN32
361         activateVirtualTerminal();
362     #endif /* _WIN32 */
363
364     printGold("\tTesting Production <-- Renewable <-- Tidal");
365
366     #ifdef _WIN32
367         std::cout << std::endl;
368     #endif
369
370     srand(time(NULL));
371
372
373     std::vector<double> time_vec_hrs (8760, 0);
374     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
375         time_vec_hrs[i] = i;
376     }
377
378     Renewable* test_tidal_ptr = testConstruct_Tidal(&time_vec_hrs);
379
380
381     try {
382         testBadConstruct_Tidal(&time_vec_hrs);
383
384         testProductionConstraint_Tidal(test_tidal_ptr);
385
386         testCommit_Tidal(test_tidal_ptr);
387         testEconomics_Tidal(test_tidal_ptr);
388     }
389
390
391     catch (...) {
392         delete test_tidal_ptr;
393
394         printGold(" ..... ");
395         printRed("FAIL");
396         std::cout << std::endl;
397         throw;
398     }
399
400
401     delete test_tidal_ptr;
402
403     printGold(" ..... ");
404     printGreen("PASS");
405     std::cout << std::endl;
406     return 0;
407
408 } /* 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.
-------------------------	---

```

136 {
137     bool error_flag = true;
138
139     try {
140         TidalInputs bad_tidal_inputs;

```

```

141         bad_tidal_inputs.design_speed_ms = -1;
142
143         Tidal bad_tidal(8760, 1, bad_tidal_inputs, time_vec_hrs_ptr);
144
145         error_flag = false;
146     } catch (...) {
147         // Task failed successfully! =P
148     }
149     if (not error_flag) {
150         expectedErrorNotDetected(__FILE__, __LINE__);
151     }
152
153     return;
154 } /* 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.
-----------------------	---

```

218 {
219     std::vector<double> dt_vec_hrs (48, 1);
220
221     std::vector<double> load_vec_kW = {
222         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
223         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
224         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
225         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
226     };
227
228     double load_kW = 0;
229     double production_kW = 0;
230     double roll = 0;
231     double tidal_resource_ms = 0;
232
233     for (int i = 0; i < 48; i++) {
234         roll = (double)rand() / RAND_MAX;
235
236         tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
237
238         roll = (double)rand() / RAND_MAX;
239
240         if (roll <= 0.1) {
241             tidal_resource_ms = 0;
242         }
243
244         else if (roll >= 0.95) {
245             tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
246         }
247
248         roll = (double)rand() / RAND_MAX;
249
250         if (roll >= 0.95) {
251             roll = 1.25;
252         }
253
254         load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
255         load_kW = load_vec_kW[i];
256
257         production_kW = test_tidal_ptr->computeProductionkW(
258             i,
259             dt_vec_hrs[i],
260             tidal_resource_ms
261         );
262
263         load_kW = test_tidal_ptr->commit(
264             i,

```



```

265         dt_vec_hrs[i],
266         production_kW,
267         load_kW
268     );
269
270     // is running (or not) as expected
271     if (production_kW > 0) {
272         testTruth(
273             test_tidal_ptr->is_running,
274             __FILE__,
275             __LINE__
276         );
277     }
278
279     else {
280         testTruth(
281             not test_tidal_ptr->is_running,
282             __FILE__,
283             __LINE__
284         );
285     }
286
287     // load_kW <= load_vec_kW (i.e., after vs before)
288     testLessThanOrEqualTo(
289         load_kW,
290         load_vec_kW[i],
291         __FILE__,
292         __LINE__
293     );
294
295     // production = dispatch + storage + curtailment
296     testFloatEquals(
297         test_tidal_ptr->production_vec_kW[i] -
298         test_tidal_ptr->dispatch_vec_kW[i] -
299         test_tidal_ptr->storage_vec_kW[i] -
300         test_tidal_ptr->curtailment_vec_kW[i],
301         0,
302         __FILE__,
303         __LINE__
304     );
305 }
306
307 return;
308 } /* 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    testFloatEquals(
111        test_tidal_ptr->firmness_factor,
112        0.8,
113        __FILE__,
114        __LINE__
115    );
116
117    return test_tidal_ptr;
118 } /* testConstruct_Tidal() */

```

5.62.2.5 testEconomics_Tidal()

```

void testEconomics_Tidal (
    Renewable * test_tidal_ptr )
326 {
327     for (int i = 0; i < 48; i++) {
328         // resource, O&M > 0 whenever tidal is running (i.e., producing)
329         if (test_tidal_ptr->is_running_vec[i]) {
330             testGreaterThan(
331                 test_tidal_ptr->operation_maintenance_cost_vec[i],
332                 0,
333                 __FILE__,
334                 __LINE__
335             );
336         }
337
338         // resource, O&M = 0 whenever tidal is not running (i.e., not producing)
339         else {
340             testFloatEquals(
341                 test_tidal_ptr->operation_maintenance_cost_vec[i],
342                 0,
343                 __FILE__,
344                 __LINE__
345             );
346         }
347     }
348
349     return;
350 } /* 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

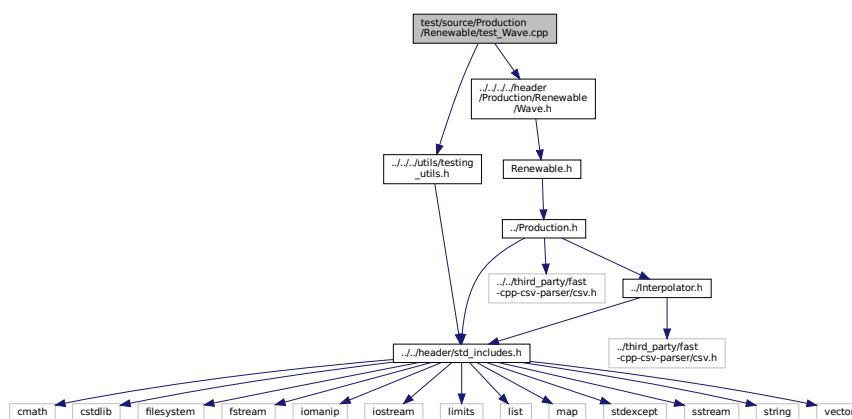
<code>test_tidal_ptr</code>	A Renewable pointer to the test Tidal object.
-----------------------------	---

```
172 {
173     testFloatEquals(
174         test_tidal_ptr->computeProductionkW(0, 1, 1e6),
175         0,
176         __FILE__,
177         __LINE__
178     );
179
180     testFloatEquals(
181         test_tidal_ptr->computeProductionkW(
182             0,
183             1,
184             ((Tidal*)test_tidal_ptr)->design_speed_ms
185         ),
186         test_tidal_ptr->capacity_kW,
187         __FILE__,
188         __LINE__
189     );
190
191     testFloatEquals(
192         test_tidal_ptr->computeProductionkW(0, 1, -1),
193         0,
194         __FILE__,
195         __LINE__
196     );
197
198     return;
199 } /* 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 )
{
    474 {
    475     #ifdef _WIN32
    476         activateVirtualTerminal();
    477     #endif /* _WIN32 */
    478
    479     printGold("\tTesting Production <-- Renewable <-- Wave");
    480
    481     #ifdef _WIN32
    482         std::cout << std::endl;
    483     #endif
    484
    485     srand(time(NULL));
    486
    487
    488     std::vector<double> time_vec_hrs (8760, 0);
    489     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
    490         time_vec_hrs[i] = i;
    491     }
    492
    493     Renewable* test_wave_ptr = testConstruct_Wave(&time_vec_hrs);
    494     Renewable* test_wave_lookup_ptr = testConstructLookup_Wave(&time_vec_hrs);
    495
    496
    497     try {
    498         testBadConstruct_Wave(&time_vec_hrs);
    499
    500         testProductionConstraint_Wave(test_wave_ptr);
```

```

501
502     testCommit_Wave(test_wave_ptr);
503     testEconomics_Wave(test_wave_ptr);
504
505     testProductionLookup_Wave(test_wave_lookup_ptr);
506 }
507
508
509 catch (...) {
510     delete test_wave_ptr;
511     delete test_wave_lookup_ptr;
512
513     printGold(" ..... ");
514     printRed("FAIL");
515     std::cout << std::endl;
516     throw;
517 }
518
519
520 delete test_wave_ptr;
521 delete test_wave_lookup_ptr;
522
523 printGold(" ..... ");
524 printGreen("PASS");
525 std::cout << std::endl;
526 return 0;
527
528 } /* 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.
-------------------------	---

```

165 {
166     bool error_flag = true;
167
168     try {
169         WaveInputs bad_wave_inputs;
170         bad_wave_inputs.design_significant_wave_height_m = -1;
171
172         Wave bad_wave(8760, 1, bad_wave_inputs, time_vec_hrs_ptr);
173
174         error_flag = false;
175     } catch (...) {
176         // Task failed successfully! =P
177     }
178     if (not error_flag) {
179         expectedErrorNotDetected(__FILE__, __LINE__);
180     }
181
182     return;
183 } /* 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.
----------------------------	--

```

236 {
237     std::vector<double> dt_vec_hrs (48, 1);
238
239     std::vector<double> load_vec_kW = {
240         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
241         1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0,
242         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
243         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
244     };
245
246     double load_kW = 0;
247     double production_kW = 0;
248     double roll = 0;
249     double significant_wave_height_m = 0;
250     double energy_period_s = 0;
251
252     for (int i = 0; i < 48; i++) {
253         roll = (double)rand() / RAND_MAX;
254
255         if (roll <= 0.05) {
256             roll = 0;
257         }
258
259         significant_wave_height_m = roll *
260             ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
261
262         roll = (double)rand() / RAND_MAX;
263
264         if (roll <= 0.05) {
265             roll = 0;
266         }
267
268         energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
269
270         roll = (double)rand() / RAND_MAX;
271
272         if (roll >= 0.95) {
273             roll = 1.25;
274         }
275
276         load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
277         load_kW = load_vec_kW[i];
278
279         production_kW = test_wave_ptr->computeProductionkW(
280             i,
281             dt_vec_hrs[i],
282             significant_wave_height_m,
283             energy_period_s
284         );
285
286         load_kW = test_wave_ptr->commit(
287             i,
288             dt_vec_hrs[i],
289             production_kW,
290             load_kW
291         );
292
293         // is running (or not) as expected
294         if (production_kW > 0) {
295             testTruth(
296                 test_wave_ptr->is_running,
297                 __FILE__,
298                 __LINE__
299             );
300         }
301
302         else {
303             testTruth(
304                 not test_wave_ptr->is_running,
305                 __FILE__,
306                 __LINE__
307             );
308         }
309
310         // load_kW <= load_vec_kW (i.e., after vs before)
311         testLessThanOrEqualTo(
312             load_kW,
313             load_vec_kW[i],
314             __FILE__,
315             __LINE__
316         );
317     }

```

```

318         // production = dispatch + storage + curtailment
319         testFloatEquals(
320             test_wave_ptr->production_vec_kW[i] -
321             test_wave_ptr->dispatch_vec_kW[i] -
322             test_wave_ptr->storage_vec_kW[i] -
323             test_wave_ptr->curtailment_vec_kW[i],
324             0,
325             __FILE__,
326             __LINE__
327         );
328     }
329
330     return;
331 } /* 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 testFloatEquals (
111     test_wave_ptr->firmness_factor,
112     0.8,
113     __FILE__,
114     __LINE__
115 );
116
117 return test_wave_ptr;
118 } /* 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.

```

137 {
138     WaveInputs wave_inputs;
139
140     wave_inputs.power_model = WavePowerProductionModel :: WAVE_POWER_LOOKUP;
141     wave_inputs.path_2_normalized_performance_matrix =
142         "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
143
144     Renewable* test_wave_lookup_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);
145
146     return test_wave_lookup_ptr;
147 } /* testConstructLookup_Wave() */

```

5.63.2.6 testEconomics_Wave()

```

void testEconomics_Wave (
    Renewable * test_wave_ptr )
{
    349 {
    350     for (int i = 0; i < 48; i++) {
    351         // resource, O&M > 0 whenever wave is running (i.e., producing)
    352         if (test_wave_ptr->is_running_vec[i]) {
    353             testGreaterThan(
    354                 test_wave_ptr->operation_maintenance_cost_vec[i],
    355                 0,
    356                 __FILE__,
    357                 __LINE__
    358             );
    359         }
    360
    361         // resource, O&M = 0 whenever wave is not running (i.e., not producing)
    362         else {
    363             testFloatEquals(
    364                 test_wave_ptr->operation_maintenance_cost_vec[i],
    365                 0,
    366                 __FILE__,
    367                 __LINE__

```



```

368         );
369     }
370 }
371
372 return;
373 } /* 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.
----------------------	--

```

201 {
202     testFloatEquals (
203         test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
204         0,
205         __FILE__,
206         __LINE__
207     );
208
209     testFloatEquals (
210         test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
211         0,
212         __FILE__,
213         __LINE__
214     );
215
216     return;
217 } /* 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.
-----------------------------	--

```

392 {
393     std::vector<double> significant_wave_height_vec_m = {
394         0.389211848822208,
395         0.836477431896843,
396         1.52738334015579,
397         1.92640601114508,
398         2.27297317532019,
399         2.87416589636605,
400         3.72275770908175,
401         3.95063175885536,
402         4.68097139867404,
403         4.97775020449812,
404         5.55184219980547,
405         6.06566629451658,
406         6.27927876785062,

```

```

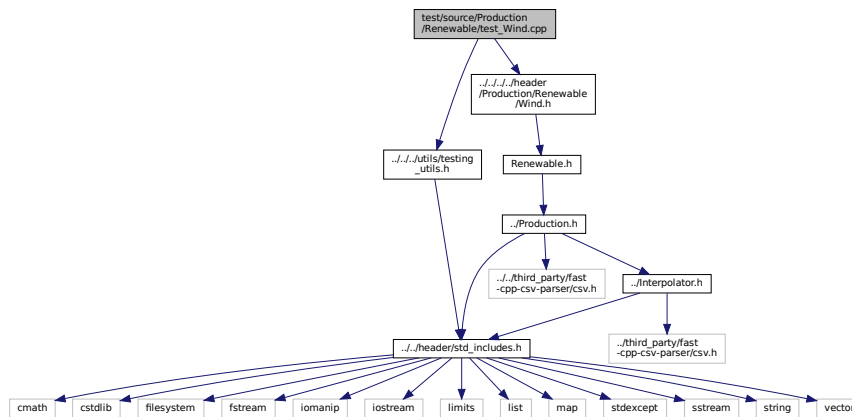
407         6.96218133671013,
408         7.51754442460228
409     };
410
411     std::vector<double> energy_period_vec_s = {
412         5.45741899698926,
413         6.00101329139007,
414         7.50567689404182,
415         8.77681262912881,
416         9.45143678206774,
417         10.7767876462885,
418         11.4795760857165,
419         12.9430684577599,
420         13.303544885703,
421         14.5069863517863,
422         15.1487890438045,
423         16.086524049077,
424         17.176609978648,
425         18.4155153740256,
426         19.1704554940162
427     };
428
429     std::vector<std::vector<double>> expected_normalized_performance_matrix = {
430
431         {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.89961488080808,0.999999999999999},
432         {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.88205808080808,0.999999999999999},
433         {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.833460808080808,0.999999999999999},
434         {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.792400000000000,0.999999999999999},
435         {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.770610000000000,0.999999999999999},
436         {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811000000000,0.999999999999999},
437         {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.705110000000000,0.999999999999999},
438         {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657000000000000,0.999999999999999},
439         {0,0.0196038727057393,0.18122235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.685500000000000},
440         {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.644200000000000},
441         {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265000000000},
442         {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.590100000000000},
443         {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.552720000000000},
444         {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.510000000000000},
445         {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.480000000000000},
446     };
447     for (size_t i = 0; i < energy_period_vec_s.size(); i++) {
448         for (size_t j = 0; j < significant_wave_height_vec_m.size(); j++) {
449             testFloatEquals(
450                 test_wave_lookup_ptr->computeProductionkW(
451                     0,
452                     1,
453                     significant_wave_height_vec_m[j],
454                     energy_period_vec_s[i]
455                 ),
456                 expected_normalized_performance_matrix[i][j] *
457                 test_wave_lookup_ptr->capacity_kW,
458                 __FILE__,
459                 __LINE__
460             );
461         }
462     }
463
464     return;
465 } /* testProductionLookup_Wave() */

```

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 )
359 {
360     #ifdef _WIN32
361         activateVirtualTerminal();
362     #endif /* _WIN32 */
363
364     printGold("\tTesting Production <-- Renewable <-- Wind");
365
366     #ifdef _WIN32
367         std::cout << std::endl;
368     #endif
369
370     srand(time(NULL));
371
372
373     std::vector<double> time_vec_hrs (8760, 0);
374     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
375         time_vec_hrs[i] = i;
376     }
377
378     Renewable* test_wind_ptr = testConstruct_Wind(&time_vec_hrs);
379
380
381     try {
382         testBadConstruct_Wind(&time_vec_hrs);
383
384         testProductionConstraint_Wind(test_wind_ptr);
385
386         testCommit_Wind(test_wind_ptr);
387         testEconomics_Wind(test_wind_ptr);
388     }
389
390
391     catch (...) {
392         delete test_wind_ptr;
393
394         printGold(" ..... ");
395         printRed("FAIL");
396         std::cout << std::endl;
397         throw;
398     }
399
400
401     delete test_wind_ptr;
402
403     printGold(" ..... ");
404     printGreen("PASS");
405     std::cout << std::endl;
406     return 0;
407
408 } /* 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.
-------------------------	---

```

136 {
137     bool error_flag = true;
138
139     try {
140         WindInputs bad_wind_inputs;

```

```

141         bad_wind_inputs.design_speed_ms = -1;
142
143         Wind bad_wind(8760, 1, bad_wind_inputs, time_vec_hrs_ptr);
144
145         error_flag = false;
146     } catch (...) {
147         // Task failed successfully! =P
148     }
149     if (not error_flag) {
150         expectedErrorNotDetected(__FILE__, __LINE__);
151     }
152
153     return;
154 } /* 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.
----------------------	--

```

218 {
219     std::vector<double> dt_vec_hrs (48, 1);
220
221     std::vector<double> load_vec_kW = {
222         1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
223         1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
224         1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
225         1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
226     };
227
228     double load_kW = 0;
229     double production_kW = 0;
230     double roll = 0;
231     double wind_resource_ms = 0;
232
233     for (int i = 0; i < 48; i++) {
234         roll = (double)rand() / RAND_MAX;
235
236         wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
237
238         roll = (double)rand() / RAND_MAX;
239
240         if (roll <= 0.1) {
241             wind_resource_ms = 0;
242         }
243
244         else if (roll >= 0.95) {
245             wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
246         }
247
248         roll = (double)rand() / RAND_MAX;
249
250         if (roll >= 0.95) {
251             roll = 1.25;
252         }
253
254         load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
255         load_kW = load_vec_kW[i];
256
257         production_kW = test_wind_ptr->computeProductionkW(
258             i,
259             dt_vec_hrs[i],
260             wind_resource_ms
261         );
262
263         load_kW = test_wind_ptr->commit(
264             i,

```

```

265         dt_vec_hrs[i],
266         production_kW,
267         load_kW
268     );
269
270     // is running (or not) as expected
271     if (production_kW > 0) {
272         testTruth(
273             test_wind_ptr->is_running,
274             __FILE__,
275             __LINE__
276         );
277     }
278
279     else {
280         testTruth(
281             not test_wind_ptr->is_running,
282             __FILE__,
283             __LINE__
284         );
285     }
286
287     // load_kW <= load_vec_kW (i.e., after vs before)
288     testLessThanOrEqualTo(
289         load_kW,
290         load_vec_kW[i],
291         __FILE__,
292         __LINE__
293     );
294
295     // production = dispatch + storage + curtailment
296     testFloatEquals(
297         test_wind_ptr->production_vec_kW[i] -
298         test_wind_ptr->dispatch_vec_kW[i] -
299         test_wind_ptr->storage_vec_kW[i] -
300         test_wind_ptr->curtailment_vec_kW[i],
301         0,
302         __FILE__,
303         __LINE__
304     );
305 }
306
307 return;
308 } /* 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    testFloatEquals(
111        test_wind_ptr->firmness_factor,
112        0.5,
113        __FILE__,
114        __LINE__
115    );
116
117    return test_wind_ptr;
118 } /* testConstruct_Wind() */

```

5.64.2.5 testEconomics_Wind()

```

void testEconomics_Wind (
    Renewable * test_wind_ptr )
{
    326 {
    327     for (int i = 0; i < 48; i++) {
    328         // resource, O&M > 0 whenever wind is running (i.e., producing)
    329         if (test_wind_ptr->is_running_vec[i]) {
    330             testGreaterThan(
    331                 test_wind_ptr->operation_maintenance_cost_vec[i],
    332                 0,
    333                 __FILE__,
    334                 __LINE__
    335             );
    336         }
    337
    338         // resource, O&M = 0 whenever wind is not running (i.e., not producing)
    339         else {
    340             testFloatEquals(
    341                 test_wind_ptr->operation_maintenance_cost_vec[i],
    342                 0,
    343                 __FILE__,
    344                 __LINE__
    345             );
    346         }
    347     }
    348
    349     return;
    350 } /* 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

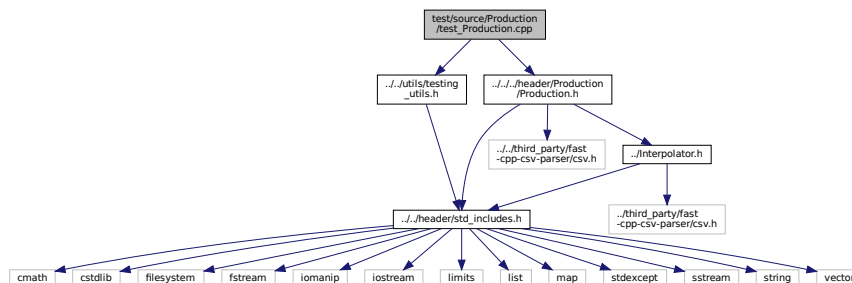
<code>test_wind_ptr</code>	A Renewable pointer to the test Wind object.
----------------------------	--

```
172 {
173     testFloatEquals(
174         test_wind_ptr->computeProductionkW(0, 1, 1e6),
175         0,
176         __FILE__,
177         __LINE__
178     );
179
180     testFloatEquals(
181         test_wind_ptr->computeProductionkW(
182             0,
183             1,
184             ((Wind*)test_wind_ptr)->design_speed_ms
185         ),
186         test_wind_ptr->capacity_kW,
187         __FILE__,
188         __LINE__
189     );
190
191     testFloatEquals(
192         test_wind_ptr->computeProductionkW(0, 1, -1),
193         0,
194         __FILE__,
195         __LINE__
196     );
197
198     return;
199 } /* 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     #ifdef _WIN32
211         std::cout << std::endl;
212     #endif
213
214     srand(time(NULL));
215
216
217     std::vector<double> time_vec_hrs (8760, 0);
218     for (size_t i = 0; i < time_vec_hrs.size(); i++) {
219         time_vec_hrs[i] = i;
220     }
221
222     Production* test_production_ptr = testConstruct_Production(&time_vec_hrs);
223
224
225     try {
226         testBadConstruct_Production(&time_vec_hrs);
227     }
228
229
230     catch (...) {
231         delete test_production_ptr;
232
233         printGold(" ..... ");
234         printRed("FAIL");
235         std::cout << std::endl;
236         throw;
237     }
238
239
240     delete test_production_ptr;
241
242     printGold(" ..... ");
243     printGreen("PASS");
244     std::cout << std::endl;
245     return 0;
246 } /* 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__,
```

```

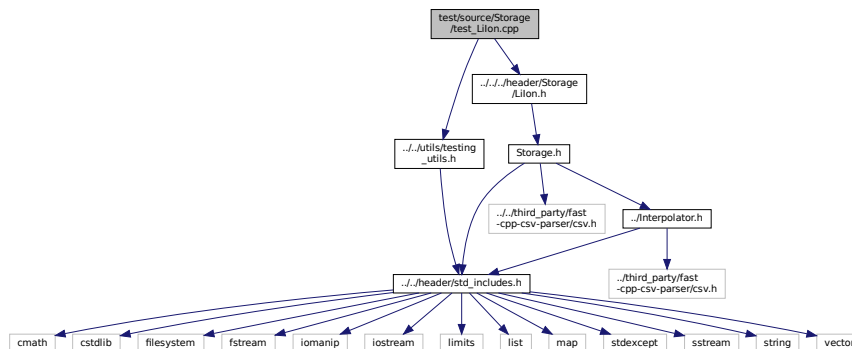
85     __LINE__
86 );
87
88 testFloatEquals(
89     production_inputs.nominal_discount_annual,
90     0.04,
91     __FILE__,
92     __LINE__
93 );
94
95 testFloatEquals(
96     test_production_ptr->n_points,
97     8760,
98     __FILE__,
99     __LINE__
100 );
101
102 testFloatEquals(
103     test_production_ptr->capacity_kW,
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_LiIon.cpp:
```



Functions

- [Storage * testConstruct_LiIon](#) (void)
A function to construct a [LiIon](#) object and spot check some post-construction attributes.
- void [testBadConstruct_LiIon](#) (void)
Function to test the trying to construct a [LiIon](#) object given bad inputs is being handled as expected.
- 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.
- 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.
- int [main](#) (int argc, char **argv)

5.66.1 Detailed Description

Testing suite for [LiIon](#) class.

A suite of tests for the [LiIon](#) 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     #ifdef _WIN32
339         std::cout << std::endl;
340     #endif
341
342     srand(time(NULL));
343
344
345     Storage* test_liion_ptr = testConstruct_LiIon();
346
347
348     try {
349         testBadConstruct_LiIon();
350
351         testCommitCharge_LiIon(test_liion_ptr);
352         testCommitDischarge_LiIon(test_liion_ptr);
353     }
354
355
356     catch (...) {
357         delete test_liion_ptr;
358
359         printGold(" ..... ");
360         printRed("FAIL");
361         std::cout << std::endl;
362         throw;
363     }
364
365
366     delete test_liion_ptr;
367
368     printGold(" ..... ");
369     printGreen("PASS");
370     std::cout << std::endl;
371     return 0;
372
373 } /* 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 LiIon 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,

```

```

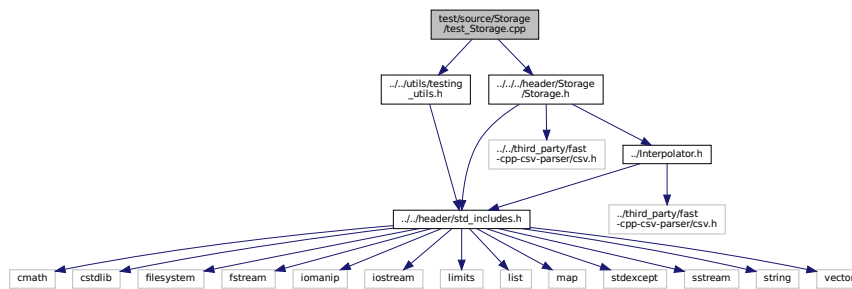
76         0.5,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         ((LiIon*)test_liion_ptr)->min_SOC,
83         0.15,
84         __FILE__,
85         __LINE__
86     );
87
88     testFloatEquals(
89         ((LiIon*)test_liion_ptr)->hysteresis_SOC,
90         0.5,
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     #ifdef _WIN32
    169         std::cout << std::endl;
```

```

170     #endif
171
172     srand(time(NULL));
173
174
175     Storage* test_storage_ptr = testConstruct_Storage();
176
177
178     try {
179         testBadConstruct_Storage();
180     }
181
182
183     catch (...) {
184         delete test_storage_ptr;
185
186         printGold(" ..... ");
187         printRed("FAIL");
188         std::cout << std::endl;
189         throw;
190     }
191
192
193     delete test_storage_ptr;
194
195     printGold(" ..... ");
196     printGreen("PASS");
197     std::cout << std::endl;
198     return 0;
199
200 } /* 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

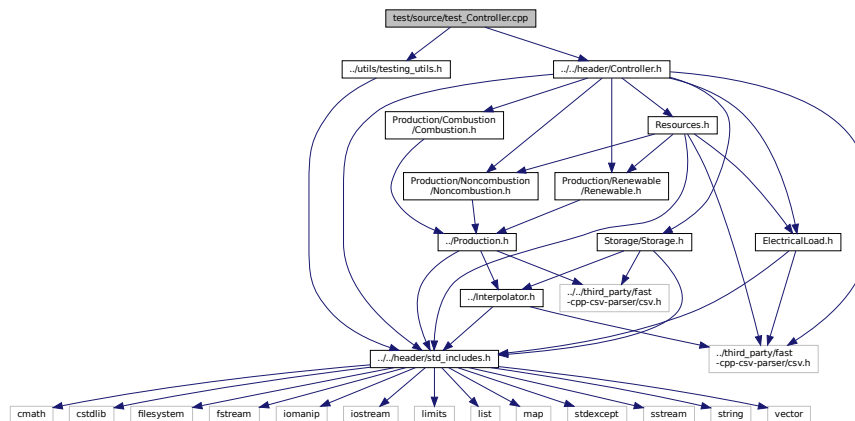
Testing suite for [Controller](#) class.

```

#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
    80     printGold("\tTesting Controller");
    81
    82     #ifdef _WIN32
    83         std::cout << std::endl;
    84     #endif
    85
    86     srand(time(NULL));
    87
    88
    89     Controller* test_controller_ptr = testConstruct_Controller();
    90 }
```

```

91
92     try {
93         //...
94     }
95
96
97     catch (...) {
98         delete test_controller_ptr;
99
100         printGold(" ..... ");
101         printRed("FAIL");
102         std::cout << std::endl;
103         throw;
104     }
105
106
107     delete test_controller_ptr;
108
109     printGold(" ..... ");
110     printGreen("PASS");
111     std::cout << std::endl;
112     return 0;
113 } /* 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

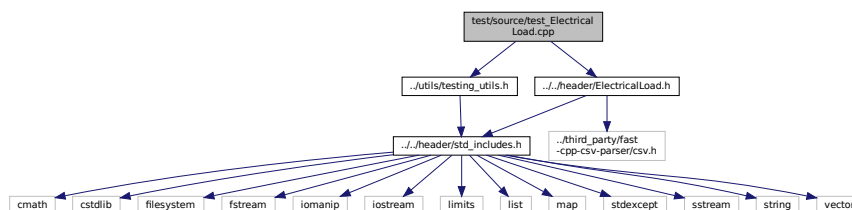
Testing suite for [ElectricalLoad](#) class.

```

#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     #ifdef _WIN32
    256         std::cout << std::endl;
    257     #endif
    258
    259     srand(time(NULL));
    260
    261
    262     ElectricalLoad* test_electrical_load_ptr = testConstruct_ElectricalLoad();
    263
    264
    265     try {
    266         testPostConstructionAttributes_ElectricalLoad(test_electrical_load_ptr);
    267         testDataRead_ElectricalLoad(test_electrical_load_ptr);
    268     }
    269
    270
    271     catch (...) {
    272         delete test_electrical_load_ptr;
    273
    274         printGold(" ..... ");
    275         printRed("FAIL");
    276         std::cout << std::endl;
    277         throw;
    278     }
    279
    280
    281     delete test_electrical_load_ptr;
    282
    283     printGold(" ..... ");
    284     printGreen("PASS");
    285     std::cout << std::endl;
    286     return 0;
    287 } /* 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

<i>test_electrical_load_ptr</i>	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

<code>test_electrical_load_ptr</code>	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

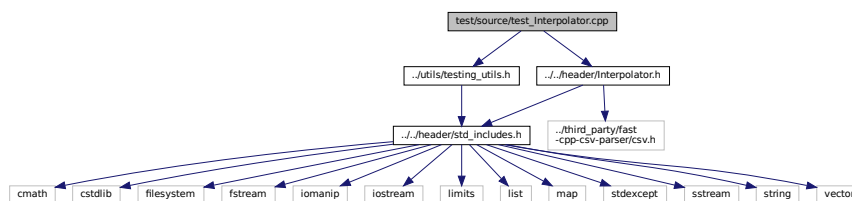
Testing suite for [Interpolator](#) class.

```

#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     #ifdef _WIN32
733         std::cout << std::endl;
734     #endif
735
736     srand(time(NULL));
737
738
739     Interpolator* test_interpolator_ptr = testConstruct_Interpolator();
740
741
742     try {
743         int data_key_1D = 1;
744         std::string path_2_data_1D =
745             "data/test/interpolation/diesel_fuel_curve.csv";
746
747         testDataRead1D_Interpolator(test_interpolator_ptr, data_key_1D, path_2_data_1D);
748         testBadIndexing1D_Interpolator(test_interpolator_ptr, -99);
749         testInvalidInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
750         testInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
751
752
753         int data_key_2D = 2;
754         std::string path_2_data_2D =
755             "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
756
757         testDataRead2D_Interpolator(test_interpolator_ptr, data_key_2D, path_2_data_2D);
758         testInvalidInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
759         testInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
760     }
761
762

```

```

763     catch (...) {
764         delete test_interpolator_ptr;
765
766         printGold(" ..... ");
767         printRed("FAIL");
768         std::cout << std::endl;
769         throw;
770     }
771
772
773     delete test_interpolator_ptr;
774
775     printGold(" ..... ");
776     printGreen("PASS");
777     std::cout << std::endl;
778     return 0;
779 } /* 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, 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, 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, 0.803685624999999, 0.816095625,
550         0.819070625},
551         {0, 0, 0.02583125, 0.14036125, 0.24490125, 0.33945125, 0.42401125, 0.49858125, 0.56316125,
552         0.61775125, 0.66235125, 0.69696125, 0.72158125, 0.73621125, 0.74085125, 0.73550125},
553         {0, 0, 0.007196875, 0.120061875, 0.222381875, 0.314156875, 0.395386875, 0.466071875,
554         0.526211875, 0.575806875, 0.614856875, 0.643361875, 0.661321875, 0.668736875, 0.665606875,
555         0.651931875},
556         {0, 0, 0, 0.0997625, 0.1998625, 0.2888625, 0.3667625, 0.4335625, 0.4892625, 0.5338625,
557         0.5673625, 0.5897625, 0.6010625, 0.6012625, 0.5903625, 0.5683625}
558     };
559
560     for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
561         for (int j = 0; j < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; j++) {
562             testFloatEquals(
563                 test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[i][j],
564                 expected_z_matrix[i][j],
565                 __FILE__,
566                 __LINE__
567             );
568         }
569     }
570
571     return;
572 } /* 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         {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.89961488
689         {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
690         {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
691         {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
692         {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
693         {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
694         {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
695         {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
696         {0,0.0196038727057393,0.18122235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
697         {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
698         {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
699         {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
700         {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
701         {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
702         {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
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
617     }
618     if (not error_flag) {
619         expectedErrorNotDetected(__FILE__, __LINE__);
620     }
621
622     return;
623 } /* testInvalidInterpolation2D_Interpolator() */

```

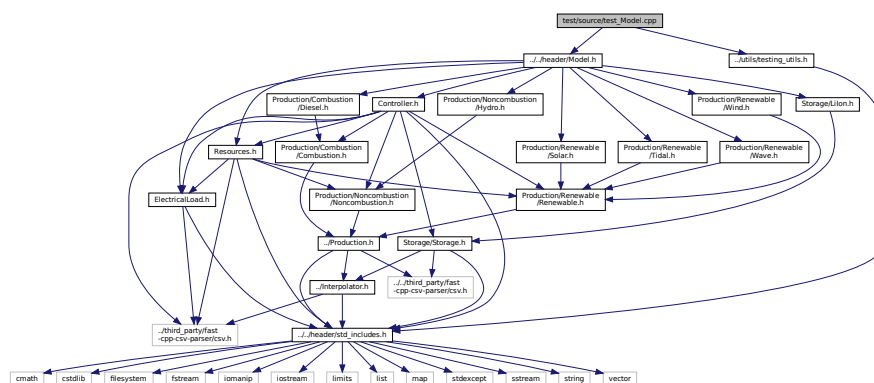
5.71 test/source/test_Model.cpp File Reference

Testing suite for [Model](#) class.

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

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

Include dependency graph for test_Model.cpp:



Functions

- [Model](#) * [testConstruct_Model](#) ([ModelInputs](#) test_model_inputs)
- void [testBadConstruct_Model](#) (void)

- Function to check if passing bad [ModelInputs](#) to the [Model](#) constructor is handled appropriately.

 - void [testPostConstructionAttributes_Model](#) ([Model](#) *test_model_ptr)

A function to check the values of various post-construction attributes.
- void [testElectricalLoadData_Model](#) ([Model](#) *test_model_ptr)

Function to check the values read into the [ElectricalLoad](#) component of the test [Model](#) object.
- 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.
- 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.
- 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.
- 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.
- 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 [testOperatingReserve_Model](#) ([Model](#) *test_model_ptr)

Function to check that the post-run state is consistent with the intended operating reserve (or "spinning reserve") logic.
- 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 )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif /* _WIN32 */

    printGold("\tTesting Model");

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

    std::cout << std::flush;

    srand(time(NULL));

    std::string path_2_electrical_load_time_series =
        "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

    ModelInputs test_model_inputs;
    test_model_inputs.path_2_electrical_load_time_series =
        path_2_electrical_load_time_series;

    Model* test_model_ptr = testConstruct_Model(test_model_inputs);

    try {
        testBadConstruct_Model();
        testPostConstructionAttributes_Model(test_model_ptr);
        testElectricalLoadData_Model(test_model_ptr);

        int solar_resource_key = 0;
        std::string path_2_solar_resource_data =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

        testAddSolarResource_Model(
            test_model_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_Model(
            test_model_ptr,
            path_2_tidal_resource_data,
```

```

1753         tidal_resource_key
1754     );
1755
1756
1757     int wave_resource_key = 2;
1758     std::string path_2_wave_resource_data =
1759         "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
1760
1761     testAddWaveResource_Model(
1762         test_model_ptr,
1763         path_2_wave_resource_data,
1764         wave_resource_key
1765     );
1766
1767
1768     int wind_resource_key = 3;
1769     std::string path_2_wind_resource_data =
1770         "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
1771
1772     testAddWindResource_Model(
1773         test_model_ptr,
1774         path_2_wind_resource_data,
1775         wind_resource_key
1776     );
1777
1778
1779     int hydro_resource_key = 4;
1780     std::string path_2_hydro_resource_data =
1781         "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
1782
1783     testAddHydroResource_Model(
1784         test_model_ptr,
1785         path_2_hydro_resource_data,
1786         hydro_resource_key
1787     );
1788
1789
1790     std::string path_2_normalized_production_time_series =
1791         "data/test/normalized_production/normalized_solar_production.csv";
1792
1793     // looping solely for the sake of profiling (also tests reset(), which is
1794     // needed for wrapping PGMcpp in an optimizer)
1795     int n_times = 1000;
1796     for (int i = 0; i < n_times; i++) {
1797         test_model_ptr->reset();
1798
1799         testAddHydro_Model(test_model_ptr, hydro_resource_key);
1800         testAddDiesel_Model(test_model_ptr);
1801         testAddSolar_Model(test_model_ptr, solar_resource_key);
1802
1803         testAddSolar_productionOverride_Model(
1804             test_model_ptr,
1805             path_2_normalized_production_time_series
1806         );
1807
1808         testAddTidal_Model(test_model_ptr, tidal_resource_key);
1809         testAddWave_Model(test_model_ptr, wave_resource_key);
1810         testAddWind_Model(test_model_ptr, wind_resource_key);
1811
1812         testAddLiIon_Model(test_model_ptr);
1813
1814         test_model_ptr->run();
1815     }
1816
1817
1818     testLoadBalance_Model(test_model_ptr);
1819     testOperatingReserve_Model(test_model_ptr);
1820     testEconomics_Model(test_model_ptr);
1821     testFuelConsumptionEmissions_Model(test_model_ptr);
1822
1823     test_model_ptr->writeResults("test/test_results/");
1824 }
1825
1826
1827 catch (...) {
1828     delete test_model_ptr;
1829
1830     printGold(" ..... ");
1831     printRed("FAIL");
1832     std::cout << std::endl;
1833     throw;
1834 }
1835
1836
1837 delete test_model_ptr;
1838
1839 printGold(" ..... ");

```

```

1840     printGreen("PASS");
1841     std::cout << std::endl;
1842     return 0;
1843 } /* 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.
-----------------------	---

```

932 {
933     DieselInputs diesel_inputs;
934     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
935     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
936
937     test_model_ptr->addDiesel(diesel_inputs);
938
939     testFloatEquals(
940         test_model_ptr->combustion_ptr_vec.size(),
941         1,
942         __FILE__,
943         __LINE__
944     );
945
946     testFloatEquals(
947         test_model_ptr->combustion_ptr_vec[0]->type,
948         CombustionType :: DIESEL,
949         __FILE__,
950         __LINE__
951     );
952
953     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
954
955     test_model_ptr->addDiesel(diesel_inputs);
956
957     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
958
959     test_model_ptr->addDiesel(diesel_inputs);
960
961     testFloatEquals(
962         test_model_ptr->combustion_ptr_vec.size(),
963         3,
964         __FILE__,
965         __LINE__
966     );
967
968     std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
969
970     for (int i = 0; i < 3; i++) {
971         testFloatEquals(
972             test_model_ptr->combustion_ptr_vec[i]->capacity_kW,
973             expected_diesel_capacity_vec_kW[i],
974             __FILE__,
975             __LINE__
976         );
977     }
978
979     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
980
981     for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
982         test_model_ptr->addDiesel(diesel_inputs);
983     }
984
985     return;
986 } /* 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.

```
882 {
883     HydroInputs hydro_inputs;
884     hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
885     hydro_inputs.reservoir_capacity_m3 = 100000;
886     hydro_inputs.init_reservoir_state = 0.5;
887     hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
888     hydro_inputs.resource_key = hydro_resource_key;
889
890     test_model_ptr->addHydro(hydro_inputs);
891
892     testFloatEquals(
893         test_model_ptr->noncombustion_ptr_vec.size(),
894         1,
895         __FILE__,
896         __LINE__
897     );
898
899     testFloatEquals(
900         test_model_ptr->noncombustion_ptr_vec[0]->type,
901         NoncombustionType :: HYDRO,
902         __FILE__,
903         __LINE__
904     );
905
906     testFloatEquals(
907         test_model_ptr->noncombustion_ptr_vec[0]->resource_key,
908         hydro_resource_key,
909         __FILE__,
910         __LINE__
911     );
912
913     return;
914 } /* 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.

```
787 {
```

```

788     test_model_ptr->addResource(
789         NoncombustionType :: HYDRO,
790         path_2_hydro_resource_data,
791         hydro_resource_key
792     );
793
794     std::vector<double> expected_hydro_resource_vec_ms = {
795         2167.91531556942,
796         2046.58261560569,
797         2007.85941123153,
798         2000.11477247929,
799         1917.50527264453,
800         1963.97311577093,
801         1908.46985899809,
802         1886.5267112678,
803         1965.26388854254,
804         1953.64692935289,
805         2084.01504296306,
806         2272.46796101188,
807         2520.29645627096,
808         2715.203242423,
809         2720.36633563203,
810         3130.83228077221,
811         3289.59741021591,
812         3981.45195965772,
813         5295.45929491303,
814         7084.47124360523,
815         7709.20557708454,
816         7436.85238642936,
817         7235.49173429668,
818         6710.14695517339,
819         6015.71085806577,
820         5279.97001316337,
821         4877.24870889801,
822         4421.60569340303,
823         3919.49483690424,
824         3498.70270322341,
825         3274.10813058883,
826         3147.61233529349,
827         2904.94693324343,
828         2805.55738101,
829         2418.32535637171,
830         2398.96375630723,
831         2260.85100182222,
832         2157.58912702878,
833         2019.47637254377,
834         1913.63295220712,
835         1863.29279076589,
836         1748.41395678279,
837         1695.49224555317,
838         1599.97501375715,
839         1559.96103873397,
840         1505.74855473274,
841         1438.62833664765,
842         1384.41585476901
843     };
844
845     for (size_t i = 0; i < expected_hydro_resource_vec_ms.size(); i++) {
846         testFloatEquals(
847             test_model_ptr->resources.resource_map_1D[hydro_resource_key][i],
848             expected_hydro_resource_vec_ms[i],
849             __FILE__,
850             __LINE__
851         );
852     }
853
854     return;
855 } /* 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.
-----------------------	---

```

1258 {
1259     LiIonInputs liion_inputs;
1260
1261     test_model_ptr->addLiIon(liion_inputs);
1262
1263     testFloatEquals(
1264         test_model_ptr->storage_ptr_vec.size(),
1265         1,
1266         __FILE__,
1267         __LINE__
1268     );
1269
1270     testFloatEquals(
1271         test_model_ptr->storage_ptr_vec[0]->type,
1272         StorageType :: LIION,
1273         __FILE__,
1274         __LINE__
1275     );
1276
1277     return;
1278 } /* 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.

```

1013 {
1014     SolarInputs solar_inputs;
1015     solar_inputs.resource_key = solar_resource_key;
1016
1017     test_model_ptr->addSolar(solar_inputs);
1018
1019     testFloatEquals(
1020         test_model_ptr->renewable_ptr_vec.size(),
1021         1,
1022         __FILE__,
1023         __LINE__
1024     );
1025
1026     testFloatEquals(
1027         test_model_ptr->renewable_ptr_vec[0]->type,
1028         RenewableType :: SOLAR,
1029         __FILE__,
1030         __LINE__
1031     );
1032
1033     return;
1034 } /* 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.

```

1061 {
1062     SolarInputs solar_inputs;
1063     solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
1064         path_2_normalized_production_time_series;
1065
1066     test_model_ptr->addSolar(solar_inputs);
1067
1068     testFloatEquals(
1069         test_model_ptr->renewable_ptr_vec.size(),
1070         2,
1071         __FILE__,
1072         __LINE__
1073     );
1074
1075     testFloatEquals(
1076         test_model_ptr->renewable_ptr_vec[1]->type,
1077         RenewableType :: SOLAR,
1078         __FILE__,
1079         __LINE__
1080     );
1081
1082     testTruth(
1083         test_model_ptr->renewable_ptr_vec[1]->normalized_production_series_given,
1084         __FILE__,
1085         __LINE__
1086     );
1087
1088     testTruth(
1089         test_model_ptr->renewable_ptr_vec[1]->path_2_normalized_production_time_series ==
1090         path_2_normalized_production_time_series,
1091         __FILE__,
1092         __LINE__
1093     );
1094
1095     return;
1096 } /* 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.

```

329 {
330     test_model_ptr->addResource(
331         RenewableType :: SOLAR,
332         path_2_solar_resource_data,
333         solar_resource_key
334     );
335
336     std::vector<double> expected_solar_resource_vec_kWm2 = {
337         0,
338         0,
339         0,
340         0,
341         0,
342         0,
343         8.51702662684015E-05,
344         0.000348341567045,
345         0.00213793728593,
346         0.004099863613322,
347         0.000997135230553,
348         0.009534527624657,
349         0.022927996790616,
350         0.0136071715294,
351         0.002535134127751,
352         0.005206897515821,
353         0.005627658648597,
354         0.000701186722215,
355         0.00017119827089,
356         0,
357         0,
358         0,
359         0,
360         0,
361         0,
362         0,
363         0,
364         0,
365         0,
366         0,
367         0,
368         0.000141055102242,
369         0.00084525014743,
370         0.024893647822702,
371         0.091245556190749,
372         0.158722176731637,
373         0.152859680515876,
374         0.149922903895116,
375         0.13049996570866,
376         0.03081254222795,
377         0.001218928911125,
378         0.000206092647423,
379         0,
380         0,
381         0,
382         0,
383         0,
384         0
385     };
386
387     for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
388         testFloatEquals(
389             test_model_ptr->resources.resource_map_1D[solar_resource_key][i],
390             expected_solar_resource_vec_kWm2[i],
391             __FILE__,
392             __LINE__
393         );
394     }
395
396     return;
397 } /* 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.

```

1123 {
1124     TidalInputs tidal_inputs;
1125     tidal_inputs.resource_key = tidal_resource_key;
1126
1127     test_model_ptr->addTidal(tidal_inputs);
1128
1129     testFloatEquals(
1130         test_model_ptr->renewable_ptr_vec.size(),
1131         3,
1132         __FILE__,
1133         __LINE__
1134     );
1135
1136     testFloatEquals(
1137         test_model_ptr->renewable_ptr_vec[2]->type,
1138         RenewableType :: TIDAL,
1139         __FILE__,
1140         __LINE__
1141     );
1142
1143     return;
1144 } /* 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.

```

429 {
430     test_model_ptr->addResource(
431         RenewableType :: TIDAL,
432         path_2_tidal_resource_data,
433         tidal_resource_key
434     );
435
436     std::vector<double> expected_tidal_resource_vec_ms = {
437         0.347439913040533,
438         0.770545522195602,
439         0.731352084836198,
440         0.293389814389542,
441         0.209959110813115,
442         0.610609623896497,
443         1.78067162013604,
444         2.53522775118089,
445         2.75966627832024,
446         2.52101111143895,
447         2.05389330201031,
448         1.3461515862445,
449         0.28909254878384,
450         0.897754086048563,
451         1.71406453837407,
452         1.85047408742869,
453         1.71507908595979,

```

```

454         1.33540349705416,
455         0.434586143463003,
456         0.500623815700637,
457         1.37172172646733,
458         1.68294125491228,
459         1.56101300975417,
460         1.04925834219412,
461         0.211395463930223,
462         1.03720048903385,
463         1.85059536356448,
464         1.85203242794517,
465         1.4091471616277,
466         0.767776539039899,
467         0.251464906990961,
468         1.47018469375652,
469         2.36260493698197,
470         2.46653750048625,
471         2.12851908739291,
472         1.62783753197988,
473         0.734594890957439,
474         0.441886297300355,
475         1.6574418350918,
476         2.0684558286637,
477         1.87717416992136,
478         1.58871262337931,
479         1.03451227609235,
480         0.193371305159817,
481         0.976400122458815,
482         1.6583227369707,
483         1.76690616570953,
484         1.54801328553115
485     };
486
487     for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
488         testFloatEquals(
489             test_model_ptr->resources.resource_map_1D[resource_key][i],
490             expected_tidal_resource_vec_ms[i],
491             __FILE__,
492             __LINE__
493         );
494     }
495
496     return;
497 } /* 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.

```

1171 {
1172     WaveInputs wave_inputs;
1173     wave_inputs.resource_key = wave_resource_key;
1174
1175     test_model_ptr->addWave(wave_inputs);
1176
1177     testFloatEquals(
1178         test_model_ptr->renewable_ptr_vec.size(),
1179         4,
1180         __FILE__,
1181         __LINE__
1182     );
1183
1184     testFloatEquals(

```

```

1185         test_model_ptr->renewable_ptr_vec[3]->type,
1186         RenewableType :: WAVE,
1187         __FILE__,
1188         __LINE__
1189     );
1190
1191     return;
1192 } /* 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.

```

529 {
530     test_model_ptr->addResource (
531         RenewableType :: WAVE,
532         path_2_wave_resource_data,
533         wave_resource_key
534     );
535
536     std::vector<double> expected_significant_wave_height_vec_m = {
537         4.26175222125028,
538         4.25020976167872,
539         4.25656524330349,
540         4.27193854786718,
541         4.28744955711233,
542         4.29421815278154,
543         4.2839937266082,
544         4.25716982457976,
545         4.22419391611483,
546         4.19588925217606,
547         4.17338788587412,
548         4.14672746914214,
549         4.10560041173665,
550         4.05074966447193,
551         3.9953696962433,
552         3.95316976150866,
553         3.92771018142378,
554         3.91129562488595,
555         3.89558312094911,
556         3.87861093931749,
557         3.86538307240754,
558         3.86108961027929,
559         3.86459448853189,
560         3.86796474016882,
561         3.86357412779993,
562         3.85554872014731,
563         3.86044266668675,
564         3.89445961915999,
565         3.95554798115731,
566         4.02265508610476,
567         4.07419587011404,
568         4.10314247143958,
569         4.11738045085928,
570         4.12554995596708,
571         4.12923992001675,
572         4.1229292327442,
573         4.10123955307441,
574         4.06748827895363,
575         4.0336230651344,

```



```

576         4.01134236393876,
577         4.00136570034559,
578         3.99368787690411,
579         3.97820924247644,
580         3.95369335178055,
581         3.92742545608532,
582         3.90683362771686,
583         3.89331520944006,
584         3.88256045801583
585     };
586
587     std::vector<double> expected_energy_period_vec_s = {
588         10.4456008226821,
589         10.4614151137651,
590         10.4462827795433,
591         10.4127692097884,
592         10.3734397942723,
593         10.3408599227669,
594         10.32637292093,
595         10.3245412676322,
596         10.310409818185,
597         10.2589529840966,
598         10.1728100603103,
599         10.0862908658929,
600         10.03480243813,
601         10.023673635806,
602         10.0243418565116,
603         10.0063487117653,
604         9.96050302286607,
605         9.9011999635568,
606         9.84451822125472,
607         9.79726875879626,
608         9.75614594835158,
609         9.7173447961368,
610         9.68342904390577,
611         9.66380508567062,
612         9.6674009575699,
613         9.68927134575103,
614         9.70979984863046,
615         9.70967357906908,
616         9.68983025704562,
617         9.6722855524805,
618         9.67973599910003,
619         9.71977125328293,
620         9.78450442291421,
621         9.86532355233449,
622         9.96158937600019,
623         10.0807018356507,
624         10.2291022504937,
625         10.39458528356,
626         10.5464393581004,
627         10.6553277500484,
628         10.7245553190084,
629         10.7893127285064,
630         10.8846512240849,
631         11.0148158739075,
632         11.1544325654719,
633         11.2772785848343,
634         11.3744362756187,
635         11.4533643503183
636     };
637
638     for (size_t i = 0; i < expected_energy_period_vec_s.size(); i++) {
639         testFloatEquals(
640             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][0],
641             expected_significant_wave_height_vec_m[i],
642             __FILE__,
643             __LINE__
644         );
645
646         testFloatEquals(
647             test_model_ptr->resources.resource_map_2D[wave_resource_key][i][1],
648             expected_energy_period_vec_s[i],
649             __FILE__,
650             __LINE__
651         );
652     }
653
654     return;
655 } /* 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.

```
1219 {
1220     WindInputs wind_inputs;
1221     wind_inputs.resource_key = wind_resource_key;
1222
1223     test_model_ptr->addWind(wind_inputs);
1224
1225     testFloatEquals(
1226         test_model_ptr->renewable_ptr_vec.size(),
1227         5,
1228         __FILE__,
1229         __LINE__
1230     );
1231
1232     testFloatEquals(
1233         test_model_ptr->renewable_ptr_vec[4]->type,
1234         RenewableType :: WIND,
1235         __FILE__,
1236         __LINE__
1237     );
1238
1239     return;
1240 } /* 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.

```
687 {
688     test_model_ptr->addResource(
689         RenewableType :: WIND,
690         path_2_wind_resource_data,
691         wind_resource_key
692     );
693
694     std::vector<double> expected_wind_resource_vec_ms = {
695         6.88566688469997,
696         5.02177105466549,
697         3.74211715899568,
698         5.67169579985362,
```

```

699         4.90670669971858,
700         4.29586955031368,
701         7.41155377205065,
702         10.2243290476943,
703         13.1258696725555,
704         13.7016198628274,
705         16.2481482330233,
706         16.5096744355418,
707         13.4354482206162,
708         14.0129230731609,
709         14.5554549260515,
710         13.4454539065912,
711         13.3447169512094,
712         11.7372615098554,
713         12.7200070078013,
714         10.6421127908149,
715         6.09869498990661,
716         5.66355596602321,
717         4.97316966910831,
718         3.48937138360567,
719         2.15917470979169,
720         1.29061103587027,
721         3.43475751425219,
722         4.11706326260927,
723         4.28905275747408,
724         5.75850263196241,
725         8.98293663055264,
726         11.7069822941315,
727         12.4031987075858,
728         15.4096570910089,
729         16.6210843829552,
730         13.3421219142573,
731         15.2112831900548,
732         18.350864533037,
733         15.8751799822971,
734         15.3921198799796,
735         15.9729192868434,
736         12.4728950178772,
737         10.177050481096,
738         10.7342247355551,
739         8.98846695631389,
740         4.14671169124739,
741         3.17256452697149,
742         3.40036336968628
743     };
744
745     for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
746         testFloatEquals(
747             test_model_ptr->resources.resource_map_1D[wind_resource_key][i],
748             expected_wind_resource_vec_ms[i],
749             __FILE__,
750             __LINE__
751         );
752     }
753
754     return;
755 } /* 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.

```

105 {
106     bool error_flag = true;
107
108     try {
109         ModelInputs bad_model_inputs; // path_2_electrical_load_time_series left empty
110
111         Model bad_model(bad_model_inputs);
112
113         error_flag = false;
114     } catch (...) {
115         // Task failed successfully! =P
116     }
117     if (not error_flag) {
118         expectedErrorNotDetected(__FILE__, __LINE__);
119     }
120 }

```

```

119     }
120
121     try {
122         ModelInputs bad_model_inputs;
123         bad_model_inputs.path_2_electrical_load_time_series =
124             "data/test/electrical_load/bad_path_";
125         bad_model_inputs.path_2_electrical_load_time_series += std::to_string(rand());
126         bad_model_inputs.path_2_electrical_load_time_series += ".csv";
127
128         Model bad_model(bad_model_inputs);
129
130         error_flag = false;
131     } catch (...) {
132         // Task failed successfully! =P
133     }
134     if (not error_flag) {
135         expectedErrorNotDetected(__FILE__, __LINE__);
136     }
137
138     return;
139 }

```

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     testFloatEquals(
75         test_model_ptr->controller.load_operating_reserve_factor,
76         0.2,
77         __FILE__,
78         __LINE__
79     );
80
81     testFloatEquals(
82         test_model_ptr->controller.max_operating_reserve_factor,
83         1,
84         __FILE__,
85         __LINE__
86     );
87
88     return test_model_ptr;
89 } /* 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.
-----------------------	---

```

1610 {
1611     testGreaterThan(

```

```

1612         test_model_ptr->net_present_cost,
1613         0,
1614         __FILE__,
1615         __LINE__
1616     );
1617
1618     testGreaterThan(
1619         test_model_ptr->levellized_cost_of_energy_kWh,
1620         0,
1621         __FILE__,
1622         __LINE__
1623     );
1624
1625     return;
1626 } /* 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.
-----------------------	---

```

212 {
213     std::vector<double> expected_dt_vec_hrs (48, 1);
214
215     std::vector<double> expected_time_vec_hrs = {
216         0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
217         12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
218         24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
219         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
220     };
221
222     std::vector<double> expected_load_vec_kW = {
223         360.253836463674,
224         355.171277826775,
225         353.776453532298,
226         353.75405737934,
227         346.592867404975,
228         340.132411175118,
229         337.354867340578,
230         340.644115618736,
231         363.639028500678,
232         378.787797779238,
233         372.215798201712,
234         395.093925731298,
235         402.325427142659,
236         386.907725462306,
237         380.709170928091,
238         372.062070914977,
239         372.328646856954,
240         391.841444284136,
241         394.029351759596,
242         383.369407765254,
243         381.093099675206,
244         382.604158946193,
245         390.744843709034,
246         383.13949492437,
247         368.150393976985,
248         364.629744480226,
249         363.572736804082,
250         359.854924202248,
251         355.207590170267,
252         349.094656012401,
253         354.365935871597,
254         343.380608328546,
255         404.673065729266,
256         486.296896820126,
257         480.225974100847,
258         457.318764401085,
259         418.177339948609,

```

```

260         414.399018364126,
261         409.678420185754,
262         404.768766016563,
263         401.699589920585,
264         402.44339040654,
265         398.138372541906,
266         396.010498627646,
267         390.165117432277,
268         375.850429417013,
269         365.567100746484,
270         365.429624610923
271     };
272
273     for (int i = 0; i < 48; i++) {
274         testFloatEquals(
275             test_model_ptr->electrical_load.dt_vec_hrs[i],
276             expected_dt_vec_hrs[i],
277             __FILE__,
278             __LINE__
279         );
280
281         testFloatEquals(
282             test_model_ptr->electrical_load.time_vec_hrs[i],
283             expected_time_vec_hrs[i],
284             __FILE__,
285             __LINE__
286         );
287
288         testFloatEquals(
289             test_model_ptr->electrical_load.load_vec_kW[i],
290             expected_load_vec_kW[i],
291             __FILE__,
292             __LINE__
293         );
294     }
295
296     return;
297 } /* 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.
-----------------------	---

```

1643 {
1644     testGreaterThan(
1645         test_model_ptr->total_fuel_consumed_L,
1646         0,
1647         __FILE__,
1648         __LINE__
1649     );
1650
1651     testGreaterThan(
1652         test_model_ptr->total_emissions.CO2_kg,
1653         0,
1654         __FILE__,
1655         __LINE__
1656     );
1657
1658     testGreaterThan(
1659         test_model_ptr->total_emissions.CO_kg,
1660         0,
1661         __FILE__,
1662         __LINE__
1663     );
1664
1665     testGreaterThan(
1666         test_model_ptr->total_emissions.NOx_kg,
1667         0,

```

```

1668     __FILE__,
1669     __LINE__
1670 );
1671
1672 testGreaterThan(
1673     test_model_ptr->total_emissions.SOx_kg,
1674     0,
1675     __FILE__,
1676     __LINE__
1677 );
1678
1679 testGreaterThan(
1680     test_model_ptr->total_emissions.CH4_kg,
1681     0,
1682     __FILE__,
1683     __LINE__
1684 );
1685
1686 testGreaterThan(
1687     test_model_ptr->total_emissions.PM_kg,
1688     0,
1689     __FILE__,
1690     __LINE__
1691 );
1692
1693 return;
1694 } /* 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

<i>test_model_ptr</i>	A pointer to the test Model object.
-----------------------	---

```

1297 {
1298     double load_kW = 0;
1299
1300     Combustion* combustion_ptr;
1301     Noncombustion* noncombustion_ptr;
1302     Renewable* renewable_ptr;
1303     Storage* storage_ptr;
1304
1305     for (int i = 0; i < test_model_ptr->electrical_load.n_points; i++) {
1306         testLessThanOrEqualTo(
1307             test_model_ptr->controller.net_load_vec_kW[i],
1308             test_model_ptr->electrical_load.max_load_kW,
1309             __FILE__,
1310             __LINE__
1311         );
1312
1313         load_kW = test_model_ptr->electrical_load.load_vec_kW[i];
1314
1315         for (size_t j = 0; j < test_model_ptr->combustion_ptr_vec.size(); j++) {
1316             combustion_ptr = test_model_ptr->combustion_ptr_vec[j];
1317
1318             testGreaterThanOrEqualTo(
1319                 combustion_ptr->production_vec_kW[i],
1320                 0,
1321                 __FILE__,
1322                 __LINE__
1323             );
1324
1325             testGreaterThanOrEqualTo(
1326                 combustion_ptr->dispatch_vec_kW[i],
1327                 0,
1328                 __FILE__,
1329                 __LINE__
1330             );

```

```

1331
1332     testGreaterThanOrEqualTo(
1333         combustion_ptr->curtailment_vec_kW[i],
1334         0,
1335         __FILE__,
1336         __LINE__
1337     );
1338
1339     testGreaterThanOrEqualTo(
1340         combustion_ptr->storage_vec_kW[i],
1341         0,
1342         __FILE__,
1343         __LINE__
1344     );
1345
1346     testFloatEquals(
1347         combustion_ptr->production_vec_kW[i] -
1348         combustion_ptr->dispatch_vec_kW[i] -
1349         combustion_ptr->curtailment_vec_kW[i] -
1350         combustion_ptr->storage_vec_kW[i],
1351         0,
1352         __FILE__,
1353         __LINE__
1354     );
1355
1356     load_kW -= combustion_ptr->dispatch_vec_kW[i];
1357 }
1358
1359 for (size_t j = 0; j < test_model_ptr->noncombustion_ptr_vec.size(); j++) {
1360     noncombustion_ptr = test_model_ptr->noncombustion_ptr_vec[j];
1361
1362     testGreaterThanOrEqualTo(
1363         noncombustion_ptr->production_vec_kW[i],
1364         0,
1365         __FILE__,
1366         __LINE__
1367     );
1368
1369     testGreaterThanOrEqualTo(
1370         noncombustion_ptr->dispatch_vec_kW[i],
1371         0,
1372         __FILE__,
1373         __LINE__
1374     );
1375
1376     testGreaterThanOrEqualTo(
1377         noncombustion_ptr->curtailment_vec_kW[i],
1378         0,
1379         __FILE__,
1380         __LINE__
1381     );
1382
1383     testGreaterThanOrEqualTo(
1384         noncombustion_ptr->storage_vec_kW[i],
1385         0,
1386         __FILE__,
1387         __LINE__
1388     );
1389
1390     testFloatEquals(
1391         noncombustion_ptr->production_vec_kW[i] -
1392         noncombustion_ptr->dispatch_vec_kW[i] -
1393         noncombustion_ptr->curtailment_vec_kW[i] -
1394         noncombustion_ptr->storage_vec_kW[i],
1395         0,
1396         __FILE__,
1397         __LINE__
1398     );
1399
1400     load_kW -= noncombustion_ptr->dispatch_vec_kW[i];
1401 }
1402
1403 for (size_t j = 0; j < test_model_ptr->renewable_ptr_vec.size(); j++) {
1404     renewable_ptr = test_model_ptr->renewable_ptr_vec[j];
1405
1406     testGreaterThanOrEqualTo(
1407         renewable_ptr->production_vec_kW[i],
1408         0,
1409         __FILE__,
1410         __LINE__
1411     );
1412
1413     testGreaterThanOrEqualTo(
1414         renewable_ptr->dispatch_vec_kW[i],
1415         0,
1416         __FILE__,
1417         __LINE__

```



```

1418         );
1419
1420         testGreaterThanOrEqualTo(
1421             renewable_ptr->curtailment_vec_kW[i],
1422             0,
1423             __FILE__,
1424             __LINE__
1425         );
1426
1427         testGreaterThanOrEqualTo(
1428             renewable_ptr->storage_vec_kW[i],
1429             0,
1430             __FILE__,
1431             __LINE__
1432         );
1433
1434         testFloatEquals(
1435             renewable_ptr->production_vec_kW[i] -
1436             renewable_ptr->dispatch_vec_kW[i] -
1437             renewable_ptr->curtailment_vec_kW[i] -
1438             renewable_ptr->storage_vec_kW[i],
1439             0,
1440             __FILE__,
1441             __LINE__
1442         );
1443
1444         load_kW -= renewable_ptr->dispatch_vec_kW[i];
1445     }
1446
1447     for (size_t j = 0; j < test_model_ptr->storage_ptr_vec.size(); j++) {
1448         storage_ptr = test_model_ptr->storage_ptr_vec[j];
1449
1450         testGreaterThanOrEqualTo(
1451             storage_ptr->charging_power_vec_kW[i],
1452             0,
1453             __FILE__,
1454             __LINE__
1455         );
1456
1457         testGreaterThanOrEqualTo(
1458             storage_ptr->discharging_power_vec_kW[i],
1459             0,
1460             __FILE__,
1461             __LINE__
1462         );
1463
1464         testTruth(
1465             not (
1466                 storage_ptr->charging_power_vec_kW[i] > 0 and
1467                 storage_ptr->discharging_power_vec_kW[i] > 0
1468             ),
1469             __FILE__,
1470             __LINE__
1471         );
1472
1473         load_kW -= storage_ptr->discharging_power_vec_kW[i];
1474     }
1475
1476     testLessThanOrEqualTo(
1477         load_kW,
1478         1e-6,
1479         __FILE__,
1480         __LINE__
1481     );
1482
1483     testLessThanOrEqualTo(
1484         test_model_ptr->controller.missed_load_vec_kW[i],
1485         0,
1486         __FILE__,
1487         __LINE__
1488     );
1489 }
1490
1491 testFloatEquals(
1492     test_model_ptr->total_dispatch_discharge_kWh,
1493     2263351.62026685,
1494     __FILE__,
1495     __LINE__
1496 );
1497
1498 return;
1499 } /* testLoadBalance_Model() */

```

5.71.2.21 testOperatingReserve_Model()

```
void testOperatingReserve_Model (
    Model * test_model_ptr )
```

Function to check that the post-run state is consistent with the intended operating reserve (or "spinning reserve") logic.

Parameters

<code>test_model_ptr</code>	A pointer to the test Model object.
-----------------------------	---

```
1517 {
1518     double load_kW = 0;
1519     double operating_reserve_kW = 0;
1520
1521     Combustion* combustion_ptr;
1522     Noncombustion* noncombustion_ptr;
1523     Renewable* renewable_ptr;
1524     Storage* storage_ptr;
1525
1526     for (int i = 0; i < test_model_ptr->electrical_load.n_points; i++) {
1527         // 1. compute operating reserve
1528         load_kW = test_model_ptr->electrical_load.load_vec_kW[i];
1529
1530         operating_reserve_kW =
1531             test_model_ptr->controller.load_operating_reserve_factor * load_kW;
1532
1533         for (size_t j = 0; j < test_model_ptr->renewable_ptr_vec.size(); j++) {
1534             renewable_ptr = test_model_ptr->renewable_ptr_vec[j];
1535
1536             operating_reserve_kW += (1 - renewable_ptr->firmness_factor) *
1537                 renewable_ptr->production_vec_kW[i];
1538
1539             testGreaterThanOrEqualTo(
1540                 renewable_ptr->production_vec_kW[i],
1541                 0,
1542                 __FILE__,
1543                 __LINE__
1544             );
1545         }
1546
1547         if (
1548             operating_reserve_kW >
1549             test_model_ptr->controller.max_operating_reserve_factor * load_kW
1550         ) {
1551             operating_reserve_kW =
1552                 test_model_ptr->controller.max_operating_reserve_factor * load_kW;
1553         }
1554
1555         testGreaterThanOrEqualTo(
1556             operating_reserve_kW,
1557             0,
1558             __FILE__,
1559             __LINE__
1560         );
1561
1562         // 2. deduct Storage discharge from operating reserve
1563         for (size_t j = 0; j < test_model_ptr->storage_ptr_vec.size(); j++) {
1564             storage_ptr = test_model_ptr->storage_ptr_vec[j];
1565
1566             operating_reserve_kW -= storage_ptr->discharging_power_vec_kW[i];
1567         }
1568
1569         // 3. deduct Noncombustion dispatch from operating reserve
1570         for (size_t j = 0; j < test_model_ptr->noncombustion_ptr_vec.size(); j++) {
1571             noncombustion_ptr = test_model_ptr->noncombustion_ptr_vec[j];
1572
1573             operating_reserve_kW -= noncombustion_ptr->dispatch_vec_kW[i];
1574         }
1575
1576         // 4. deduct Combustion dispatch from operating reserve
1577         for (size_t j = 0; j < test_model_ptr->combustion_ptr_vec.size(); j++) {
1578             combustion_ptr = test_model_ptr->combustion_ptr_vec[j];
1579
1580             operating_reserve_kW -= combustion_ptr->dispatch_vec_kW[i];
1581         }
1582
1583         // 5. assert remaining operating reserve is zero (+/- tolerance)
1584         testLessThanOrEqualTo(
```

```

1585         operating_reserve_kW,
1586         1e-6,
1587         __FILE__,
1588         __LINE__
1589     );
1590 }
1591
1592 return;
1593 } /* testOperatingReserve_Model() */

```

5.71.2.22 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.
-----------------------	---

```

156 {
157     testFloatEquals (
158         test_model_ptr->electrical_load.n_points,
159         8760,
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals (
165         test_model_ptr->electrical_load.n_years,
166         0.999886,
167         __FILE__,
168         __LINE__
169     );
170
171     testFloatEquals (
172         test_model_ptr->electrical_load.min_load_kW,
173         82.1211213927802,
174         __FILE__,
175         __LINE__
176     );
177
178     testFloatEquals (
179         test_model_ptr->electrical_load.mean_load_kW,
180         258.373472633202,
181         __FILE__,
182         __LINE__
183     );
184
185
186     testFloatEquals (
187         test_model_ptr->electrical_load.max_load_kW,
188         500,
189         __FILE__,
190         __LINE__
191     );
192
193     return;
194 } /* testPostConstructionAttributes_Model() */

```

5.72 test/source/test_Resources.cpp File Reference

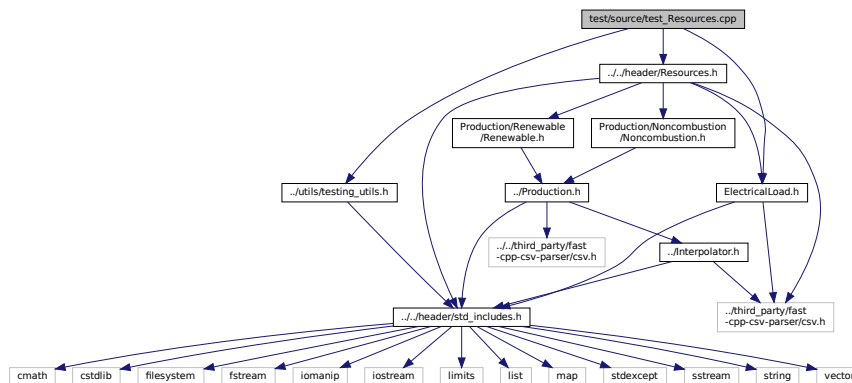
Testing suite for [Resources](#) class.

```

#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 )
{
    783 {
    784     #ifdef _WIN32
    785         activateVirtualTerminal();
    786     #endif /* _WIN32 */
    787
    788     printGold("\tTesting Resources");
    789
    790     #ifdef _WIN32
    791         std::cout « std::endl;
    792     #endif
    793
    794     srand(time(NULL));
    795
    796
    797     std::string path_2_electrical_load_time_series =
    798         "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
    799
    800     ElectricalLoad* test_electrical_load_ptr =
    801         new ElectricalLoad(path_2_electrical_load_time_series);
    802
    803     Resources* test_resources_ptr = testConstruct_Resources();
    804
    805
    806     try {
    807         int solar_resource_key = 0;
    808         std::string path_2_solar_resource_data =
    809             "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
    810
    811         testAddSolarResource_Resources(
    812             test_resources_ptr,
    813             test_electrical_load_ptr,
    814             path_2_solar_resource_data,
    815             solar_resource_key
    816         );
    817
    818         testBadAdd_Resources(
    819             test_resources_ptr,
    820             test_electrical_load_ptr,
    821             path_2_solar_resource_data,
    822             solar_resource_key
    823         );
    824
    825
    826         int tidal_resource_key = 1;
    827         std::string path_2_tidal_resource_data =
    828             "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
    829
    830         testAddTidalResource_Resources(
    831             test_resources_ptr,
    832             test_electrical_load_ptr,
    833             path_2_tidal_resource_data,
    834             tidal_resource_key
    835         );
    836
    837
    838         int wave_resource_key = 2;
    839         std::string path_2_wave_resource_data =
    840             "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
    841
    842         testAddWaveResource_Resources(
    843             test_resources_ptr,
    844             test_electrical_load_ptr,
    845             path_2_wave_resource_data,
    846             wave_resource_key
    847         );
    848
    849
    850         int wind_resource_key = 3;
    851         std::string path_2_wind_resource_data =
    852             "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
    853
    854         testAddWindResource_Resources(
    855             test_resources_ptr,
    856             test_electrical_load_ptr,
    857             path_2_wind_resource_data,
    858             wind_resource_key
    859         );
    860
    861
    862         int hydro_resource_key = 4;

```

```

863         std::string path_2_hydro_resource_data =
864             "data/test/resources/hydro_inflow_peak-2000m3hr_1yr_dt-1hr.csv";
865
866         testAddHydroResource_Resources(
867             test_resources_ptr,
868             test_electrical_load_ptr,
869             path_2_hydro_resource_data,
870             hydro_resource_key
871         );
872     }
873
874
875     catch (...) {
876         delete test_electrical_load_ptr;
877         delete test_resources_ptr;
878
879         printGold(" ..... ");
880         printRed("FAIL");
881         std::cout << std::endl;
882         throw;
883     }
884
885
886     delete test_electrical_load_ptr;
887     delete test_resources_ptr;
888
889     printGold(" ..... ");
890     printGreen("PASS");
891     std::cout << std::endl;
892     return 0;
893 } /* 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.

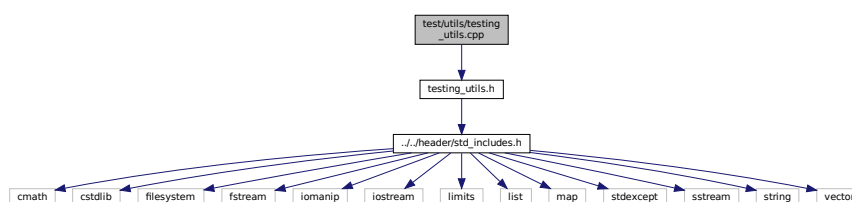
```
64 {
65     Resources* test_resources_ptr = new Resources();
66
67     testFloatEquals(
68         test_resources_ptr->resource_map_1D.size(),
69         0,
70         __FILE__,
71         __LINE__
72     );
73
74     testFloatEquals(
75         test_resources_ptr->path_map_1D.size(),
76         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() */

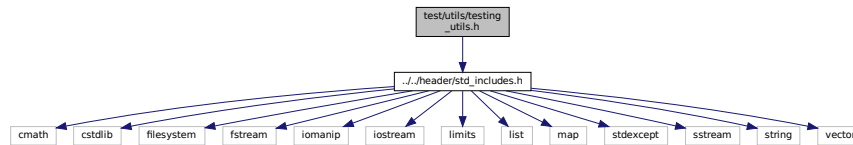
```

5.74 test/utils/testing_utils.h File Reference

Header file for various PGMcpp testing utilities.

```
#include "../..//header/std_includes.h"
```

Include dependency graph for testing_utils.h:



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



Macros

- `#define FLOAT_TOLERANCE 1e-6`
A tolerance for application to floating point equality tests.

Functions

- void [printGreen](#) (std::string)
A function that sends green text to std::cout.
- void [printGold](#) (std::string)
A function that sends gold text to std::cout.
- void [printRed](#) (std::string)
A function that sends red text to std::cout.
- void [testFloatEquals](#) (double, double, std::string, int)
Tests for the equality of two floating point numbers x and y (to within `FLOAT_TOLERANCE`).
- void [testGreaterThan](#) (double, double, std::string, int)
Tests if $x > y$.
- void [testGreaterThanOrEqualTo](#) (double, double, std::string, int)
Tests if $x \geq y$.
- void [testLessThan](#) (double, double, std::string, int)
Tests if $x < y$.
- void [testLessThanOrEqualTo](#) (double, double, std::string, int)
Tests if $x \leq y$.
- void [testTruth](#) (bool, std::string, int)
Tests if the given statement is true.
- void [expectedErrorNotDetected](#) (std::string, int)
A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.74.1 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. 255, 256
- 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. 256
- CIMAC. Guide to Diesel Exhaust Emissions Control of NOx, SOx, Particulates, Smoke, and CO2. Technical report, Conseil International des Machines à Combustion, 2008. Included: docs/refs/diesel_emissions_ref_2.pdf. 60
- P. Gilman, A. Dobos, N. DiOrio, J. Freeman, S. Janzou, and D. Ryberg. SAM Photovoltaic Model Technical Reference Update. Technical report, NREL, 2018. URL <https://research-hub.nrel.gov/en/publications/sam-photovoltaic-model-technical-reference-2016-update>. 211, 212, 213, 214, 215, 217, 218, 219, 220, 222, 223, 224
- HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 172, 242
- HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 16, 159, 172, 173, 241, 242
- HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 51, 60
- HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 51, 60
- HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 51, 60
- 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, 242
- HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 173, 241
- HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 172, 242
- W. Jakob. pybind11 — Seamless operability between C++11 and Python, 2023. URL <https://pybind11.readthedocs.io/en/stable/>. 329, 331, 334, 337, 339, 342, 346, 348, 351, 353, 355, 358, 360, 361, 364, 365, 367, 370

- 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. 255, 256
- 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. 258, 274, 275
- 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. 288
- NRCan. Auto\$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 60
- 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. 273
- M.H. Safaripour and M.A. Mehrabian. Predicting the direct, diffuse, and global solar radiations on a horizontal surface and comparing with real data. *Heat Mass Transfer*, 47, 2011. doi: 10.1007/s00231-011-0814-8. 214
- A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023a. Included: docs/refs/battery_degradation.pdf. 116, 118, 129
- A. Truelove. Hydro Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023b. Included: docs/refs/hydro.pdf. 75, 77, 78, 79, 81
- 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. 257, 271, 289
- D. van Heesch. Doxygen: Generate documentation from source code, 2023. URL <https://www.doxygen.nl>. 302
- 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. 256
- U. Zafar. Literature Review of Wind Turbines. *Bauhaus Universität*, 2018. URL https://www.researchgate.net/publication/329680977_Literature_Review_of_Wind_Turbines. 288

Index

- __USE_MATH_DEFINES
 - std_includes.h, [322](#)
- __checkBounds1D
 - Interpolator, [93](#)
- __checkBounds2D
 - Interpolator, [94](#)
- __checkDataKey1D
 - Interpolator, [95](#)
- __checkDataKey2D
 - Interpolator, [95](#)
- __checkInputs
 - Combustion, [14](#)
 - Diesel, [49](#)
 - Hydro, [74](#)
 - Lilon, [113](#)
 - Model, [136](#)
 - Noncombustion, [157](#)
 - Production, [167](#)
 - Renewable, [184](#)
 - Solar, [209](#)
 - Storage, [239](#)
 - Tidal, [255](#)
 - Wave, [270](#)
 - Wind, [287](#)
- __checkNormalizedProduction
 - Production, [168](#)
- __checkResourceKey1D
 - Resources, [192](#), [193](#)
- __checkResourceKey2D
 - Resources, [193](#)
- __checkTimePoint
 - Production, [169](#)
 - Resources, [194](#)
- __computeCubicProductionkW
 - Tidal, [256](#)
 - Wind, [288](#)
- __computeDetailedProductionkW
 - Solar, [211](#)
- __computeEconomics
 - Model, [137](#)
- __computeExponentialProductionkW
 - Tidal, [256](#)
 - Wind, [289](#)
- __computeFuelAndEmissions
 - Model, [137](#)
- __computeGaussianProductionkW
 - Wave, [271](#)
- __computeLevellizedCostOfEnergy
 - Model, [138](#)
- __computeLookupProductionkW
 - Tidal, [257](#)
 - Wave, [272](#)
 - Wind, [290](#)
- __computeNetPresentCost
 - Model, [139](#)
- __computeParaboloidProductionkW
 - Wave, [272](#)
- __computeRealDiscountAnnual
 - Storage, [240](#)
- __computeRenewableProduction
 - Controller, [28](#)
- __computeSimpleProductionkW
 - Solar, [211](#)
- __constructCombustionMap
 - Controller, [28](#)
- __flowToPower
 - Hydro, [75](#)
- __getAcceptableFlow
 - Hydro, [75](#)
- __getAngleOfIncidenceRad
 - Solar, [212](#)
- __getAvailableFlow
 - Hydro, [76](#)
- __getBcal
 - Lilon, [115](#)
- __getBeamIrradiancekWm2
 - Solar, [213](#)
- __getDataStringMatrix
 - Interpolator, [96](#)
- __getDeclinationRad
 - Solar, [213](#)
- __getDiffuseHorizontalIrradiancekWm2
 - Solar, [214](#)
- __getDiffuseIrradiancekWm2
 - Solar, [214](#)
- __getDirectNormalIrradiancekWm2
 - Solar, [215](#)
- __getEacal
 - Lilon, [116](#)
- __getEclipticLongitudeRad
 - Solar, [215](#)
- __getEfficiencyFactor
 - Hydro, [76](#)
- __getGenericCapitalCost
 - Diesel, [50](#)
 - Hydro, [77](#)
 - Lilon, [116](#)
 - Solar, [216](#)

- Tidal, [258](#)
- Wave, [274](#)
- Wind, [290](#)
- __getGenericFuelIntercept
 - Diesel, [51](#)
- __getGenericFuelSlope
 - Diesel, [51](#)
- __getGenericOpMaintCost
 - Diesel, [51](#)
 - Hydro, [77](#)
 - Lilon, [117](#)
 - Solar, [216](#)
 - Tidal, [258](#)
 - Wave, [274](#)
 - Wind, [290](#)
- __getGreenwichMeanSiderialTimeHrs
 - Solar, [216](#)
- __getGroundReflectedIrradiancekWm2
 - Solar, [217](#)
- __getHourAngleRad
 - Solar, [217](#)
- __getInterpolationIndex
 - Interpolator, [96](#)
- __getLocalMeanSiderialTimeHrs
 - Solar, [218](#)
- __getMaximumFlowm3hr
 - Hydro, [78](#)
- __getMeanAnomalyRad
 - Solar, [219](#)
- __getMeanLongitudeDeg
 - Solar, [219](#)
- __getMinimumFlowm3hr
 - Hydro, [78](#)
- __getObliquityOfEclipticRad
 - Solar, [220](#)
- __getPlaneOfArrayIrradiancekWm2
 - Solar, [220](#)
- __getRenewableProduction
 - Controller, [30](#)
- __getRightAscensionRad
 - Solar, [222](#)
- __getSolarAltitudeRad
 - Solar, [222](#)
- __getSolarAzimuthRad
 - Solar, [223](#)
- __getSolarZenithRad
 - Solar, [224](#)
- __handleCombustionDispatch
 - Controller, [31](#)
- __handleDegradation
 - Lilon, [117](#)
- __handleNoncombustionDispatch
 - Controller, [34](#)
- __handleRenewableDispatch
 - Controller, [35](#)
- __handleStartStop
 - Diesel, [52](#)
 - Noncombustion, [157](#)
 - Renewable, [184](#)
- __handleStorageCharging
 - Controller, [35](#)
- __handleStorageDischarging
 - Controller, [37](#)
- __initInterpolator
 - Hydro, [78](#)
- __isNonNumeric
 - Interpolator, [97](#)
- __modelDegradation
 - Lilon, [118](#)
- __powerToFlow
 - Hydro, [80](#)
- __readData1D
 - Interpolator, [97](#)
- __readData2D
 - Interpolator, [98](#)
- __readHydroResource
 - Resources, [195](#)
- __readNormalizedProductionData
 - Production, [169](#)
- __readSolarResource
 - Resources, [196](#)
- __readTidalResource
 - Resources, [197](#)
- __readWaveResource
 - Resources, [198](#)
- __readWindResource
 - Resources, [199](#)
- __splitCommaSeparatedString
 - Interpolator, [100](#)
- __throwLengthError
 - Production, [170](#)
 - Resources, [200](#)
- __throwReadError
 - Interpolator, [101](#)
- __toggleDepleted
 - Lilon, [118](#)
- __updateState
 - Hydro, [81](#)
- __writeSummary
 - Combustion, [14](#)
 - Diesel, [53](#)
 - Hydro, [82](#)
 - Lilon, [119](#)
 - Model, [139](#)
 - Noncombustion, [158](#)
 - Renewable, [185](#)
 - Solar, [225](#)
 - Storage, [241](#)
 - Tidal, [258](#)
 - Wave, [275](#)
 - Wind, [291](#)
- __writeTimeSeries
 - Combustion, [15](#)
 - Diesel, [55](#)
 - Hydro, [83](#)
 - Lilon, [120](#)

- Model, 142
- Noncombustion, 158
- Renewable, 185
- Solar, 226
- Storage, 241
- Tidal, 260
- Wave, 276
- Wind, 292
- ~Combustion
 - Combustion, 13
- ~Controller
 - Controller, 27
- ~Diesel
 - Diesel, 48
- ~ElectricalLoad
 - ElectricalLoad, 64
- ~Hydro
 - Hydro, 74
- ~Interpolator
 - Interpolator, 93
- ~Lilon
 - Lilon, 113
- ~Model
 - Model, 136
- ~Noncombustion
 - Noncombustion, 157
- ~Production
 - Production, 167
- ~Renewable
 - Renewable, 184
- ~Resources
 - Resources, 192
- ~Solar
 - Solar, 209
- ~Storage
 - Storage, 239
- ~Tidal
 - Tidal, 255
- ~Wave
 - Wave, 270
- ~Wind
 - Wind, 287
- addData1D
 - Interpolator, 101
- addData2D
 - Interpolator, 102
- addDiesel
 - Model, 143
- addHydro
 - Model, 144
- addLilon
 - Model, 144
- addResource
 - Model, 144, 145
 - Resources, 200, 201
- addSolar
 - Model, 145
- addTidal
 - Model, 146
- addWave
 - Model, 146
- addWind
 - Model, 147
- albedo_ground_reflectance
 - Solar, 229
 - SolarInputs, 233
- applyDispatchControl
 - Controller, 38
- capacity_kW
 - Production, 174
 - ProductionInputs, 180
- capital_cost
 - DieselInputs, 60
 - HydroInputs, 90
 - LilonInputs, 130
 - Production, 174
 - SolarInputs, 233
 - Storage, 244
 - TidalInputs, 265
 - WaveInputs, 282
 - WindInputs, 298
- capital_cost_vec
 - Production, 174
 - Storage, 245
- CH4_emissions_intensity_kgL
 - Combustion, 20
 - DieselInputs, 60
- CH4_emissions_vec_kg
 - Combustion, 20
- CH4_kg
 - Emissions, 68
- charge_kWh
 - Storage, 245
- charge_vec_kWh
 - Storage, 245
- charging_efficiency
 - Lilon, 124
 - LilonInputs, 130
- charging_power_vec_kW
 - Storage, 245
- clear
 - Controller, 41
 - ElectricalLoad, 64
 - Model, 147
 - Resources, 202
- CO2_emissions_intensity_kgL
 - Combustion, 20
 - DieselInputs, 60
- CO2_emissions_vec_kg
 - Combustion, 20
- CO2_kg
 - Emissions, 68
- CO_emissions_intensity_kgL
 - Combustion, 20
 - DieselInputs, 61
- CO_emissions_vec_kg

- Combustion, 20
- CO_kg
 - Emissions, 68
- 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, 306
 - DIESEL, 306
 - FUEL_MODE_LINEAR, 308
 - FUEL_MODE_LOOKUP, 308
 - FuelMode, 306
 - N_COMBUSTION_TYPES, 306
 - N_FUEL_MODES, 308
- combustion_inputs
 - DieselInputs, 61
- combustion_map
 - Controller, 43
- combustion_ptr_vec
 - Model, 150
- 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, 306
- commit
 - Combustion, 15
 - Diesel, 56
 - Hydro, 84
 - Noncombustion, 158, 159
 - Production, 170
 - Renewable, 185
 - Solar, 227
 - Tidal, 260
 - Wave, 277
 - Wind, 293
- commitCharge
 - Lilon, 121
 - Storage, 241
- commitDischarge
 - Lilon, 121
 - Storage, 241
- computeEconomics
 - Combustion, 16
 - Noncombustion, 159
 - Production, 171
 - Renewable, 186
 - Storage, 242
- computeFuelAndEmissions
 - Combustion, 16
- computeProductionkW
 - Renewable, 186, 187
 - Solar, 228
 - Tidal, 261
 - Wave, 278
 - Wind, 294
- computeRealDiscountAnnual
 - Production, 172
- control_mode
 - Controller, 43
 - ModelInputs, 153
- control_string
 - Controller, 43
- Controller, 26
 - __computeRenewableProduction, 28
 - __constructCombustionMap, 28
 - __getRenewableProduction, 30
 - __handleCombustionDispatch, 31
 - __handleNoncombustionDispatch, 34
 - __handleRenewableDispatch, 35
 - __handleStorageCharging, 35
 - __handleStorageDischarging, 37
 - ~Controller, 27
 - applyDispatchControl, 38
 - clear, 41
 - combustion_map, 43
 - control_mode, 43

- control_string, 43
- Controller, 27
- init, 41
- load_operating_reserve_factor, 44
- max_operating_reserve_factor, 44
- missed_load_vec_kW, 44
- net_load_vec_kW, 44
- required_operating_reserve_kW, 44
- setControlMode, 42
- setLoadOperatingReserveFactor, 42
- setMaxOperatingReserveFactor, 43
- storage_discharge_bool_vec, 44
- controller
 - Model, 150
- Controller.h
 - ControlMode, 302
 - CYCLE_CHARGING, 302
 - LOAD_FOLLOWING, 302
 - N_CONTROL_MODES, 302
- ControlMode
 - Controller.h, 302
- curtailment_vec_kW
 - Production, 174
- CYCLE_CHARGING
 - Controller.h, 302
- cycle_charging_setpoint
 - Combustion, 21
 - CombustionInputs, 25
- def
 - PYBIND11_Combustion.cpp, 331
 - PYBIND11_Controller.cpp, 358
 - PYBIND11_Diesel.cpp, 334
 - PYBIND11_Hydro.cpp, 337
 - PYBIND11_Interpolator.cpp, 362
 - PYBIND11_Noncombustion.cpp, 340
 - PYBIND11_Production.cpp, 342
 - PYBIND11_Renewable.cpp, 346
 - PYBIND11_Solar.cpp, 348
 - PYBIND11_Tidal.cpp, 351
 - PYBIND11_Wave.cpp, 353
 - PYBIND11_Wind.cpp, 356
- def_readwrite
 - PYBIND11_Combustion.cpp, 331, 332
 - PYBIND11_Controller.cpp, 358, 359
 - PYBIND11_Diesel.cpp, 334, 335
 - PYBIND11_ElectricalLoad.cpp, 360
 - PYBIND11_Hydro.cpp, 337, 338
 - PYBIND11_Interpolator.cpp, 362, 363
 - PYBIND11_Lilon.cpp, 367–369
 - PYBIND11_Model.cpp, 364
 - PYBIND11_Production.cpp, 342–345
 - PYBIND11_Renewable.cpp, 346
 - PYBIND11_Resources.cpp, 365
 - PYBIND11_Solar.cpp, 348, 349
 - PYBIND11_Storage.cpp, 370, 371
 - PYBIND11_Tidal.cpp, 351
 - PYBIND11_Wave.cpp, 353, 354
 - PYBIND11_Wind.cpp, 356
- degradation_a_cal
 - Lilon, 124
 - LilonInputs, 130
- degradation_alpha
 - Lilon, 125
 - LilonInputs, 130
- degradation_B_hat_cal_0
 - Lilon, 125
 - LilonInputs, 130
- degradation_beta
 - Lilon, 125
 - LilonInputs, 130
- degradation_Ea_cal_0
 - Lilon, 125
 - LilonInputs, 131
- degradation_r_cal
 - Lilon, 125
 - LilonInputs, 131
- degradation_s_cal
 - Lilon, 125
 - LilonInputs, 131
- derating
 - Solar, 229
 - SolarInputs, 233
- design_energy_period_s
 - Wave, 280
 - WaveInputs, 282
- design_significant_wave_height_m
 - Wave, 280
 - WaveInputs, 282
- design_speed_ms
 - Tidal, 263
 - TidalInputs, 265
 - Wind, 296
 - WindInputs, 298
- DIESEL
 - Combustion.h, 306
- Diesel, 45
 - __checkInputs, 49
 - __getGenericCapitalCost, 50
 - __getGenericFuelIntercept, 51
 - __getGenericFuelSlope, 51
 - __getGenericOpMaintCost, 51
 - __handleStartStop, 52
 - __writeSummary, 53
 - __writeTimeSeries, 55
 - ~Diesel, 48
 - commit, 56
 - Diesel, 47
 - handleReplacement, 57
 - minimum_load_ratio, 58
 - minimum_runtime_hrs, 58
 - requestProductionkW, 57
 - time_since_last_start_hrs, 58
- DieselInputs, 59
 - capital_cost, 60
 - CH4_emissions_intensity_kgL, 60
 - CO2_emissions_intensity_kgL, 60

- CO_emissions_intensity_kgL, 61
- combustion_inputs, 61
- fuel_cost_L, 61
- linear_fuel_intercept_LkWh, 61
- linear_fuel_slope_LkWh, 61
- minimum_load_ratio, 61
- minimum_runtime_hrs, 62
- NOx_emissions_intensity_kgL, 62
- operation_maintenance_cost_kWh, 62
- PM_emissions_intensity_kgL, 62
- replace_running_hrs, 62
- SOx_emissions_intensity_kgL, 62
- discharging_efficiency
 - Lilon, 126
 - LilonInputs, 131
- discharging_power_vec_kW
 - Storage, 245
- dispatch_vec_kW
 - Production, 174
- dt_vec_hrs
 - ElectricalLoad, 66
- dynamic_energy_capacity_kWh
 - Lilon, 126
- dynamic_power_capacity_kW
 - Lilon, 126
- electrical_load
 - Model, 150
- ElectricalLoad, 63
 - ~ElectricalLoad, 64
 - clear, 64
 - dt_vec_hrs, 66
 - ElectricalLoad, 64
 - load_vec_kW, 66
 - max_load_kW, 66
 - mean_load_kW, 66
 - min_load_kW, 67
 - n_points, 67
 - n_years, 67
 - path_2_electrical_load_time_series, 67
 - readLoadData, 65
 - time_vec_hrs, 67
- Emissions, 68
 - CH4_kg, 68
 - CO2_kg, 68
 - CO_kg, 68
 - NOx_kg, 69
 - PM_kg, 69
 - SOx_kg, 69
- energy_capacity_kWh
 - Storage, 245
 - StorageInputs, 250
- example.cpp
 - main, 325
- expectedErrorNotDetected
 - testing_utils.cpp, 495
 - testing_utils.h, 502
- firmness_factor
 - Renewable, 188
 - SolarInputs, 233
 - TidalInputs, 265
 - WaveInputs, 282
 - WindInputs, 298
- FLOAT_TOLERANCE
 - testing_utils.h, 502
- FLOW_TO_POWER_INTERP_KEY
 - Hydro.h, 310
- fluid_density_kgm3
 - Hydro, 86
 - HydroInputs, 90
- fuel_consumption_vec_L
 - Combustion, 21
- fuel_cost_L
 - Combustion, 21
 - DieselInputs, 61
- fuel_cost_vec
 - Combustion, 21
- fuel_mode
 - Combustion, 21
 - CombustionInputs, 25
- FUEL_MODE_LINEAR
 - Combustion.h, 308
- FUEL_MODE_LOOKUP
 - Combustion.h, 308
- fuel_mode_str
 - Combustion, 21
- FuelMode
 - Combustion.h, 306
- gas_constant_JmolK
 - Lilon, 126
 - LilonInputs, 131
- GENERATOR_EFFICIENCY_INTERP_KEY
 - Hydro.h, 310
- getAcceptablekW
 - Lilon, 122
 - Storage, 243
- getAvailablekW
 - Lilon, 123
 - Storage, 243
- getEmissionskg
 - Combustion, 17
- getFuelConsumptionL
 - Combustion, 17
- getProductionkW
 - Production, 173
- handleReplacement
 - Combustion, 18
 - Diesel, 57
 - Hydro, 85
 - Lilon, 124
 - Noncombustion, 160
 - Production, 173
 - Renewable, 187
 - Solar, 229
 - Storage, 243

- Tidal, 262
- Wave, 279
- Wind, 295
- header/Controller.h, 301
- header/doxygen_cite.h, 302
- header/ElectricalLoad.h, 303
- header/Interpolator.h, 303
- header/Model.h, 304
- header/Production/Combustion/Combustion.h, 305
- header/Production/Combustion/Diesel.h, 308
- header/Production/Noncombustion/Hydro.h, 309
- header/Production/Noncombustion/Noncombustion.h, 311
- header/Production/Production.h, 312
- header/Production/Renewable/Renewable.h, 313
- header/Production/Renewable/Solar.h, 314
- header/Production/Renewable/Tidal.h, 316
- header/Production/Renewable/Wave.h, 317
- header/Production/Renewable/Wind.h, 319
- header/Resources.h, 320
- header/std_includes.h, 321
- header/Storage/Lilon.h, 322
- header/Storage/Storage.h, 323
- HYDRO
 - Noncombustion.h, 312
- Hydro, 70
 - __checkInputs, 74
 - __flowToPower, 75
 - __getAcceptableFlow, 75
 - __getAvailableFlow, 76
 - __getEfficiencyFactor, 76
 - __getGenericCapitalCost, 77
 - __getGenericOpMaintCost, 77
 - __getMaximumFlowm3hr, 78
 - __getMinimumFlowm3hr, 78
 - __initInterpolator, 78
 - __powerToFlow, 80
 - __updateState, 81
 - __writeSummary, 82
 - __writeTimeSeries, 83
 - ~Hydro, 74
 - commit, 84
 - fluid_density_kgm3, 86
 - handleReplacement, 85
 - Hydro, 72
 - init_reservoir_state, 87
 - maximum_flow_m3hr, 87
 - minimum_flow_m3hr, 87
 - minimum_power_kW, 87
 - net_head_m, 87
 - requestProductionkW, 85
 - reservoir_capacity_m3, 87
 - spill_rate_vec_m3hr, 88
 - stored_volume_m3, 88
 - stored_volume_vec_m3, 88
 - turbine_flow_vec_m3hr, 88
 - turbine_type, 88
- Hydro.h
 - FLOW_TO_POWER_INTERP_KEY, 310
 - GENERATOR_EFFICIENCY_INTERP_KEY, 310
 - HYDRO_TURBINE_FRANCIS, 310
 - HYDRO_TURBINE_KAPLAN, 310
 - HYDRO_TURBINE_PELTON, 310
 - HydroInterpKeys, 310
 - HydroTurbineType, 310
 - N_HYDRO_INTERP_KEYS, 310
 - N_HYDRO_TURBINES, 310
 - TURBINE_EFFICIENCY_INTERP_KEY, 310
- HYDRO_TURBINE_FRANCIS
 - Hydro.h, 310
- HYDRO_TURBINE_KAPLAN
 - Hydro.h, 310
- HYDRO_TURBINE_PELTON
 - Hydro.h, 310
- HydroInputs, 89
 - capital_cost, 90
 - fluid_density_kgm3, 90
 - init_reservoir_state, 90
 - net_head_m, 90
 - noncombustion_inputs, 90
 - operation_maintenance_cost_kWh, 90
 - reservoir_capacity_m3, 91
 - resource_key, 91
 - turbine_type, 91
- HydroInterpKeys
 - Hydro.h, 310
- HydroTurbineType
 - Hydro.h, 310
- hysteresis_SOC
 - Lilon, 126
 - LilonInputs, 131
- init
 - Controller, 41
- init_reservoir_state
 - Hydro, 87
 - HydroInputs, 90
- init_SOC
 - Lilon, 126
 - LilonInputs, 132
- interp1D
 - Interpolator, 102
- interp2D
 - Interpolator, 103
- interp_map_1D
 - Interpolator, 104
- interp_map_2D
 - Interpolator, 104
- Interpolator, 91
 - __checkBounds1D, 93
 - __checkBounds2D, 94
 - __checkDataKey1D, 95
 - __checkDataKey2D, 95
 - __getDataStringMatrix, 96
 - __getInterpolationIndex, 96
 - __isNonNumeric, 97
 - __readData1D, 97

- __readData2D, 98
- __splitCommaSeparatedString, 100
- __throwReadError, 101
- ~Interpolator, 93
- addData1D, 101
- addData2D, 102
- interp1D, 102
- interp2D, 103
- interp_map_1D, 104
- interp_map_2D, 104
- Interpolator, 93
- path_map_1D, 104
- path_map_2D, 104
- interpolator
 - Production, 175
 - Storage, 246
- InterpolatorStruct1D, 105
 - max_x, 105
 - min_x, 105
 - n_points, 105
 - x_vec, 106
 - y_vec, 106
- InterpolatorStruct2D, 106
 - max_x, 107
 - max_y, 107
 - min_x, 107
 - min_y, 107
 - n_cols, 107
 - n_rows, 107
 - x_vec, 108
 - y_vec, 108
 - z_matrix, 108
- is_depleted
 - Storage, 246
- is_running
 - Production, 175
- is_running_vec
 - Production, 175
- is_sunk
 - Production, 175
 - ProductionInputs, 180
 - Storage, 246
 - StorageInputs, 250
- julian_day
 - Solar, 230
 - SolarInputs, 234
- latitude_deg
 - Solar, 230
 - SolarInputs, 234
- latitude_rad
 - Solar, 230
- levellized_cost_of_energy_kWh
 - Model, 150
 - Production, 175
 - Storage, 246
- LIION
 - Storage.h, 324
- Lilon, 109
 - __checkInputs, 113
 - __getBcal, 115
 - __getEacal, 116
 - __getGenericCapitalCost, 116
 - __getGenericOpMaintCost, 117
 - __handleDegradation, 117
 - __modelDegradation, 118
 - __toggleDepleted, 118
 - __writeSummary, 119
 - __writeTimeSeries, 120
 - ~Lilon, 113
 - charging_efficiency, 124
 - commitCharge, 121
 - commitDischarge, 121
 - degradation_a_cal, 124
 - degradation_alpha, 125
 - degradation_B_hat_cal_0, 125
 - degradation_beta, 125
 - degradation_Ea_cal_0, 125
 - degradation_r_cal, 125
 - degradation_s_cal, 125
 - discharging_efficiency, 126
 - dynamic_energy_capacity_kWh, 126
 - dynamic_power_capacity_kW, 126
 - gas_constant_JmolK, 126
 - getAcceptablekW, 122
 - getAvailablekW, 123
 - handleReplacement, 124
 - hysteresis_SOC, 126
 - init_SOC, 126
 - Lilon, 111
 - max_SOC, 127
 - min_SOC, 127
 - power_degradation_flag, 127
 - replace_SOH, 127
 - SOH, 127
 - SOH_vec, 127
 - temperature_K, 128
- LilonInputs, 128
 - capital_cost, 130
 - charging_efficiency, 130
 - degradation_a_cal, 130
 - degradation_alpha, 130
 - degradation_B_hat_cal_0, 130
 - degradation_beta, 130
 - degradation_Ea_cal_0, 131
 - degradation_r_cal, 131
 - degradation_s_cal, 131
 - discharging_efficiency, 131
 - gas_constant_JmolK, 131
 - hysteresis_SOC, 131
 - init_SOC, 132
 - max_SOC, 132
 - min_SOC, 132
 - operation_maintenance_cost_kWh, 132
 - power_degradation_flag, 132
 - replace_SOH, 132

- storage_inputs, 133
- temperature_K, 133
- linear_fuel_intercept_LkWh
 - Combustion, 22
 - DieselInputs, 61
- linear_fuel_slope_LkWh
 - Combustion, 22
 - DieselInputs, 61
- LOAD_FOLLOWING
 - Controller.h, 302
- load_operating_reserve_factor
 - Controller, 44
 - ModelInputs, 153
- load_vec_kW
 - ElectricalLoad, 66
- longitude_deg
 - Solar, 230
 - SolarInputs, 234
- longitude_rad
 - Solar, 230
- main
 - example.cpp, 325
 - test_Combustion.cpp, 381
 - test_Controller.cpp, 444
 - test_Diesel.cpp, 385
 - test_ElectricalLoad.cpp, 446
 - test_Hydro.cpp, 395
 - test_Interpolator.cpp, 450
 - test_Lilon.cpp, 436
 - test_Model.cpp, 462
 - test_Noncombustion.cpp, 400
 - test_Production.cpp, 433
 - test_Renewable.cpp, 403
 - test_Resources.cpp, 484
 - test_Solar.cpp, 405
 - test_Storage.cpp, 441
 - test_Tidal.cpp, 414
 - test_Wave.cpp, 420
 - test_Wind.cpp, 427
- max_load_kW
 - ElectricalLoad, 66
- max_operating_reserve_factor
 - Controller, 44
 - ModelInputs, 153
- max_SOC
 - Lilon, 127
 - LilonInputs, 132
- max_x
 - InterpolatorStruct1D, 105
 - InterpolatorStruct2D, 107
- max_y
 - InterpolatorStruct2D, 107
- maximum_flow_m3hr
 - Hydro, 87
- mean_load_kW
 - ElectricalLoad, 66
- min_load_kW
 - ElectricalLoad, 67
- min_SOC
 - Lilon, 127
 - LilonInputs, 132
- min_x
 - InterpolatorStruct1D, 105
 - InterpolatorStruct2D, 107
- min_y
 - InterpolatorStruct2D, 107
- minimum_flow_m3hr
 - Hydro, 87
- minimum_load_ratio
 - Diesel, 58
 - DieselInputs, 61
- minimum_power_kW
 - Hydro, 87
- minimum_runtime_hrs
 - Diesel, 58
 - DieselInputs, 62
- missed_load_vec_kW
 - Controller, 44
- Model, 133
 - __checkInputs, 136
 - __computeEconomics, 137
 - __computeFuelAndEmissions, 137
 - __computeLevellizedCostOfEnergy, 138
 - __computeNetPresentCost, 139
 - __writeSummary, 139
 - __writeTimeSeries, 142
 - ~Model, 136
 - addDiesel, 143
 - addHydro, 144
 - addLilon, 144
 - addResource, 144, 145
 - addSolar, 145
 - addTidal, 146
 - addWave, 146
 - addWind, 147
 - clear, 147
 - combustion_ptr_vec, 150
 - controller, 150
 - electrical_load, 150
 - levellized_cost_of_energy_kWh, 150
 - Model, 135, 136
 - net_present_cost, 151
 - noncombustion_ptr_vec, 151
 - renewable_ptr_vec, 151
 - reset, 147
 - resources, 151
 - run, 148
 - storage_ptr_vec, 151
 - total_dispatch_discharge_kWh, 151
 - total_emissions, 152
 - total_fuel_consumed_L, 152
 - total_renewable_dispatch_kWh, 152
 - writeResults, 149
- ModelInputs, 152
 - control_mode, 153
 - load_operating_reserve_factor, 153

- max_operating_reserve_factor, [153](#)
- path_2_electrical_load_time_series, [153](#)
- n_cols
 - InterpolatorStruct2D, [107](#)
- N_COMBUSTION_TYPES
 - Combustion.h, [306](#)
- N_CONTROL_MODES
 - Controller.h, [302](#)
- N_FUEL_MODES
 - Combustion.h, [308](#)
- N_HYDRO_INTERP_KEYS
 - Hydro.h, [310](#)
- N_HYDRO_TURBINES
 - Hydro.h, [310](#)
- N_NONCOMBUSTION_TYPES
 - Noncombustion.h, [312](#)
- n_points
 - ElectricalLoad, [67](#)
 - InterpolatorStruct1D, [105](#)
 - Production, [175](#)
 - Storage, [246](#)
- N_RENEWABLE_TYPES
 - Renewable.h, [314](#)
- n_replacements
 - Production, [176](#)
 - Storage, [246](#)
- n_rows
 - InterpolatorStruct2D, [107](#)
- N_SOLAR_POWER_PRODUCTION_MODELS
 - Solar.h, [316](#)
- n_starts
 - Production, [176](#)
- N_STORAGE_TYPES
 - Storage.h, [324](#)
- N_TIDAL_POWER_PRODUCTION_MODELS
 - Tidal.h, [317](#)
- N_WAVE_POWER_PRODUCTION_MODELS
 - Wave.h, [318](#)
- N_WIND_POWER_PRODUCTION_MODELS
 - Wind.h, [320](#)
- n_years
 - ElectricalLoad, [67](#)
 - Production, [176](#)
 - Storage, [247](#)
- net_head_m
 - Hydro, [87](#)
 - HydroInputs, [90](#)
- net_load_vec_kW
 - Controller, [44](#)
- net_present_cost
 - Model, [151](#)
 - Production, [176](#)
 - Storage, [247](#)
- nominal_discount_annual
 - Production, [176](#)
 - ProductionInputs, [180](#)
 - Storage, [247](#)
 - StorageInputs, [250](#)
- nominal_fuel_escalation_annual
 - Combustion, [22](#)
 - CombustionInputs, [25](#)
- nominal_inflation_annual
 - Production, [176](#)
 - ProductionInputs, [180](#)
 - Storage, [247](#)
 - StorageInputs, [250](#)
- 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, [312](#)
 - N_NONCOMBUSTION_TYPES, [312](#)
 - NoncombustionType, [312](#)
- noncombustion_inputs
 - HydroInputs, [90](#)
- noncombustion_ptr_vec
 - Model, [151](#)
- NoncombustionInputs, [162](#)
 - production_inputs, [162](#)
- NoncombustionType
 - Noncombustion.h, [312](#)
- normalized_production_series_given
 - Production, [177](#)
- normalized_production_vec
 - Production, [177](#)
- NOx_emissions_intensity_kgL
 - Combustion, [22](#)
 - DieselInputs, [62](#)
- NOx_emissions_vec_kg
 - Combustion, [22](#)
- NOx_kg
 - Emissions, [69](#)
- operation_maintenance_cost_kWh
 - DieselInputs, [62](#)
 - HydroInputs, [90](#)
 - LilnInputs, [132](#)
 - Production, [177](#)
 - SolarInputs, [234](#)
 - Storage, [247](#)
 - TidalInputs, [265](#)
 - WaveInputs, [282](#)
 - WindInputs, [298](#)
- operation_maintenance_cost_vec
 - Production, [177](#)
 - Storage, [247](#)

- panel_azimuth_deg
 - Solar, [230](#)
 - SolarInputs, [234](#)
- panel_azimuth_rad
 - Solar, [231](#)
- panel_tilt_deg
 - Solar, [231](#)
 - SolarInputs, [234](#)
- panel_tilt_rad
 - Solar, [231](#)
- path_2_electrical_load_time_series
 - ElectricalLoad, [67](#)
 - ModelInputs, [153](#)
- path_2_fuel_interp_data
 - CombustionInputs, [25](#)
- path_2_normalized_performance_matrix
 - WaveInputs, [283](#)
- path_2_normalized_production_time_series
 - Production, [177](#)
 - ProductionInputs, [180](#)
- path_map_1D
 - Interpolator, [104](#)
 - Resources, [202](#)
- path_map_2D
 - Interpolator, [104](#)
 - Resources, [203](#)
- PM_emissions_intensity_kgL
 - Combustion, [22](#)
 - DieselInputs, [62](#)
- PM_emissions_vec_kg
 - Combustion, [23](#)
- PM_kg
 - Emissions, [69](#)
- power_capacity_kW
 - Storage, [248](#)
 - StorageInputs, [250](#)
- power_degradation_flag
 - Lilon, [127](#)
 - LilonInputs, [132](#)
- power_kW
 - Storage, [248](#)
- power_model
 - Solar, [231](#)
 - SolarInputs, [235](#)
 - Tidal, [263](#)
 - TidalInputs, [265](#)
 - Wave, [280](#)
 - WaveInputs, [283](#)
 - Wind, [296](#)
 - WindInputs, [298](#)
- power_model_string
 - Solar, [231](#)
 - Tidal, [263](#)
 - Wave, [280](#)
 - Wind, [296](#)
- print_flag
 - Production, [177](#)
 - ProductionInputs, [180](#)
- Storage, [248](#)
- StorageInputs, [250](#)
- printGold
 - testing_utils.cpp, [496](#)
 - testing_utils.h, [502](#)
- printGreen
 - testing_utils.cpp, [496](#)
 - testing_utils.h, [503](#)
- printRed
 - testing_utils.cpp, [496](#)
 - testing_utils.h, [503](#)
- 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, [190](#)

- 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, 324
- PYBIND11_Combustion.cpp
 - def, 331
 - def_readwrite, 331, 332
 - value, 332
- PYBIND11_Controller.cpp
 - def, 358
 - def_readwrite, 358, 359
 - value, 359
- PYBIND11_Diesel.cpp
 - def, 334
 - def_readwrite, 334, 335
- PYBIND11_ElectricalLoad.cpp
 - def_readwrite, 360
- PYBIND11_Hydro.cpp
 - def, 337
 - def_readwrite, 337, 338
 - value, 338, 339
- PYBIND11_Interpolator.cpp
 - def, 362
 - def_readwrite, 362, 363
- PYBIND11_Lilon.cpp
 - def_readwrite, 367–369
- PYBIND11_Model.cpp
 - def_readwrite, 364
- PYBIND11_MODULE
 - PYBIND11_PGM.cpp, 330
- PYBIND11_Noncombustion.cpp
 - def, 340
 - value, 340
- PYBIND11_PGM.cpp
 - PYBIND11_MODULE, 330
- PYBIND11_Production.cpp
 - def, 342
 - def_readwrite, 342–345
- PYBIND11_Renewable.cpp
 - def, 346
 - def_readwrite, 346
 - value, 347
- PYBIND11_Resources.cpp
 - def_readwrite, 365
- PYBIND11_Solar.cpp
 - def, 348
 - def_readwrite, 348, 349
 - value, 349
- PYBIND11_Storage.cpp
 - def_readwrite, 370, 371
 - value, 370
- PYBIND11_Tidal.cpp
 - def, 351
 - def_readwrite, 351
 - value, 352
- PYBIND11_Wave.cpp
 - def, 353
 - def_readwrite, 353, 354
 - value, 354
- PYBIND11_Wind.cpp
 - def, 356
 - def_readwrite, 356
 - value, 356, 357
- pybindings/PYBIND11_PGM.cpp, 329
- pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp, 330
- pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp, 333
- pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp, 336
- pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp, 339
- pybindings/snippets/Production/PYBIND11_Production.cpp, 340
- pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp, 346
- pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp, 347
- pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp, 350
- pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp, 352
- pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp, 355
- pybindings/snippets/PYBIND11_Controller.cpp, 357
- pybindings/snippets/PYBIND11_ElectricalLoad.cpp, 359
- pybindings/snippets/PYBIND11_Interpolator.cpp, 361
- pybindings/snippets/PYBIND11_Model.cpp, 363
- pybindings/snippets/PYBIND11_Resources.cpp, 365
- pybindings/snippets/Storage/PYBIND11_Lilon.cpp, 366
- pybindings/snippets/Storage/PYBIND11_Storage.cpp, 369
- readLoadData
 - ElectricalLoad, 65
- real_discount_annual
 - Production, 178
 - Storage, 248
- 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

- firmness_factor, 188
- handleReplacement, 187
- Renewable, 183
- resource_key, 188
- type, 189
- writeResults, 187
- Renewable.h
 - N_RENEWABLE_TYPES, 314
 - RenewableType, 314
 - SOLAR, 314
 - TIDAL, 314
 - WAVE, 314
 - WIND, 314
- renewable_inputs
 - SolarInputs, 235
 - TidalInputs, 266
 - WaveInputs, 283
 - WindInputs, 299
- renewable_ptr_vec
 - Model, 151
- RenewableInputs, 189
 - production_inputs, 190
- RenewableType
 - Renewable.h, 314
- replace_running_hrs
 - DieselInputs, 62
 - Production, 178
 - ProductionInputs, 181
- replace_SOH
 - Lilon, 127
 - LilonInputs, 132
- requestProductionkW
 - Combustion, 18
 - Diesel, 57
 - Hydro, 85
 - Noncombustion, 160
- required_operating_reserve_kW
 - Controller, 44
- reservoir_capacity_m3
 - Hydro, 87
 - HydroInputs, 91
- reset
 - Model, 147
- resource_key
 - HydroInputs, 91
 - Noncombustion, 161
 - Renewable, 188
 - SolarInputs, 235
 - TidalInputs, 266
 - WaveInputs, 283
 - WindInputs, 299
- resource_map_1D
 - Resources, 203
- resource_map_2D
 - Resources, 203
- Resources, 190
 - __checkResourceKey1D, 192, 193
 - __checkResourceKey2D, 193
 - __checkTimePoint, 194
 - __readHydroResource, 195
 - __readSolarResource, 196
 - __readTidalResource, 197
 - __readWaveResource, 198
 - __readWindResource, 199
 - __throwLengthError, 200
 - ~Resources, 192
 - addResource, 200, 201
 - clear, 202
 - path_map_1D, 202
 - path_map_2D, 203
 - resource_map_1D, 203
 - resource_map_2D, 203
 - Resources, 191
 - string_map_1D, 203
 - string_map_2D, 203
- resources
 - Model, 151
- run
 - Model, 148
- running_hours
 - Production, 178
- setControlMode
 - Controller, 42
- setLoadOperatingReserveFactor
 - Controller, 42
- setMaxOperatingReserveFactor
 - Controller, 43
- SOH
 - Lilon, 127
- SOH_vec
 - Lilon, 127
- SOLAR
 - Renewable.h, 314
- Solar, 204
 - __checkInputs, 209
 - __computeDetailedProductionkW, 211
 - __computeSimpleProductionkW, 211
 - __getAngleOfIncidenceRad, 212
 - __getBeamIrradiancekWm2, 213
 - __getDeclinationRad, 213
 - __getDiffuseHorizontalIrradiancekWm2, 214
 - __getDiffuseIrradiancekWm2, 214
 - __getDirectNormalIrradiancekWm2, 215
 - __getEclipticLongitudeRad, 215
 - __getGenericCapitalCost, 216
 - __getGenericOpMaintCost, 216
 - __getGreenwichMeanSiderialTimeHrs, 216
 - __getGroundReflectedIrradiancekWm2, 217
 - __getHourAngleRad, 217
 - __getLocalMeanSiderialTimeHrs, 218
 - __getMeanAnomalyRad, 219
 - __getMeanLongitudeDeg, 219
 - __getObliquityOfEclipticRad, 220
 - __getPlaneOfArrayIrradiancekWm2, 220
 - __getRightAscensionRad, 222
 - __getSolarAltitudeRad, 222

- __getSolarAzimuthRad, 223
- __getSolarZenithRad, 224
- __writeSummary, 225
- __writeTimeSeries, 226
- ~Solar, 209
- albedo_ground_reflectance, 229
- commit, 227
- computeProductionkW, 228
- derating, 229
- handleReplacement, 229
- julian_day, 230
- latitude_deg, 230
- latitude_rad, 230
- longitude_deg, 230
- longitude_rad, 230
- panel_azimuth_deg, 230
- panel_azimuth_rad, 231
- panel_tilt_deg, 231
- panel_tilt_rad, 231
- power_model, 231
- power_model_string, 231
- Solar, 207
- Solar.h
 - N_SOLAR_POWER_PRODUCTION_MODELS, 316
 - SOLAR_POWER_DETAILED, 316
 - SOLAR_POWER_SIMPLE, 316
 - SolarPowerProductionModel, 315
- SOLAR_POWER_DETAILED
 - Solar.h, 316
- SOLAR_POWER_SIMPLE
 - Solar.h, 316
- SolarInputs, 232
 - albedo_ground_reflectance, 233
 - capital_cost, 233
 - derating, 233
 - firmness_factor, 233
 - julian_day, 234
 - latitude_deg, 234
 - longitude_deg, 234
 - operation_maintenance_cost_kWh, 234
 - panel_azimuth_deg, 234
 - panel_tilt_deg, 234
 - power_model, 235
 - renewable_inputs, 235
 - resource_key, 235
- SolarPowerProductionModel
 - Solar.h, 315
- source/Controller.cpp, 371
- source/ElectricalLoad.cpp, 372
- source/Interpolator.cpp, 372
- source/Model.cpp, 373
- source/Production/Combustion/Combustion.cpp, 373
- source/Production/Combustion/Diesel.cpp, 374
- source/Production/Noncombustion/Hydro.cpp, 374
- source/Production/Noncombustion/Noncombustion.cpp, 375
- source/Production/Production.cpp, 376
- source/Production/Renewable/Renewable.cpp, 376
- source/Production/Renewable/Solar.cpp, 377
- source/Production/Renewable/Tidal.cpp, 377
- source/Production/Renewable/Wave.cpp, 378
- source/Production/Renewable/Wind.cpp, 378
- source/Resources.cpp, 379
- source/Storage/Lilon.cpp, 380
- source/Storage/Storage.cpp, 380
- SOx_emissions_intensity_kgL
 - Combustion, 23
 - DieselInputs, 62
- SOx_emissions_vec_kg
 - Combustion, 23
- SOx_kg
 - Emissions, 69
- spill_rate_vec_m3hr
 - Hydro, 88
- std_includes.h
 - _USE_MATH_DEFINES, 322
- Storage, 236
 - __checkInputs, 239
 - __computeRealDiscountAnnual, 240
 - __writeSummary, 241
 - __writeTimeSeries, 241
 - ~Storage, 239
 - capital_cost, 244
 - capital_cost_vec, 245
 - charge_kWh, 245
 - charge_vec_kWh, 245
 - charging_power_vec_kW, 245
 - commitCharge, 241
 - commitDischarge, 241
 - computeEconomics, 242
 - discharging_power_vec_kW, 245
 - energy_capacity_kWh, 245
 - getAcceptablekW, 243
 - getAvailablekW, 243
 - handleReplacement, 243
 - interpolator, 246
 - is_depleted, 246
 - is_sunk, 246
 - levellized_cost_of_energy_kWh, 246
 - n_points, 246
 - n_replacements, 246
 - n_years, 247
 - net_present_cost, 247
 - nominal_discount_annual, 247
 - nominal_inflation_annual, 247
 - operation_maintenance_cost_kWh, 247
 - operation_maintenance_cost_vec, 247
 - power_capacity_kW, 248
 - power_kW, 248
 - print_flag, 248
 - real_discount_annual, 248
 - Storage, 238
 - total_discharge_kWh, 248
 - type, 248
 - type_str, 249

- writeResults, [243](#)
- Storage.h
 - LIION, [324](#)
 - N_STORAGE_TYPES, [324](#)
 - StorageType, [324](#)
- storage_discharge_bool_vec
 - Controller, [44](#)
- storage_inputs
 - LilonInputs, [133](#)
- storage_ptr_vec
 - Model, [151](#)
- storage_vec_kW
 - Production, [178](#)
- StorageInputs, [249](#)
 - energy_capacity_kWh, [250](#)
 - is_sunk, [250](#)
 - nominal_discount_annual, [250](#)
 - nominal_inflation_annual, [250](#)
 - power_capacity_kW, [250](#)
 - print_flag, [250](#)
- StorageType
 - Storage.h, [324](#)
- stored_volume_m3
 - Hydro, [88](#)
- stored_volume_vec_m3
 - Hydro, [88](#)
- string_map_1D
 - Resources, [203](#)
- string_map_2D
 - Resources, [203](#)
- temperature_K
 - Lilon, [128](#)
 - LilonInputs, [133](#)
- test/source/Production/Combustion/test_Combustion.cpp, [381](#)
- test/source/Production/Combustion/test_Diesel.cpp, [383](#)
- test/source/Production/Noncombustion/test_Hydro.cpp, [394](#)
- test/source/Production/Noncombustion/test_Noncombustion.cpp, [400](#)
- test/source/Production/Renewable/test_Renewable.cpp, [402](#)
- test/source/Production/Renewable/test_Solar.cpp, [404](#)
- test/source/Production/Renewable/test_Tidal.cpp, [413](#)
- test/source/Production/Renewable/test_Wave.cpp, [419](#)
- test/source/Production/Renewable/test_Wind.cpp, [426](#)
- test/source/Production/test_Production.cpp, [432](#)
- test/source/Storage/test_Lilon.cpp, [435](#)
- test/source/Storage/test_Storage.cpp, [441](#)
- test/source/test_Controller.cpp, [443](#)
- test/source/test_ElectricalLoad.cpp, [445](#)
- test/source/test_Interpolator.cpp, [449](#)
- test/source/test_Model.cpp, [460](#)
- test/source/test_Resources.cpp, [483](#)
- test/utls/testing_utils.cpp, [494](#)
- test/utls/testing_utils.h, [500](#)
- test_Combustion.cpp
 - main, [381](#)
 - testConstruct_Combustion, [382](#)
- test_Controller.cpp
 - main, [444](#)
 - testConstruct_Controller, [445](#)
- test_Diesel.cpp
 - main, [385](#)
 - testBadConstruct_Diesel, [385](#)
 - testCapacityConstraint_Diesel, [386](#)
 - testCommit_Diesel, [386](#)
 - testConstruct_Diesel, [388](#)
 - testConstructLookup_Diesel, [389](#)
 - testEconomics_Diesel, [389](#)
 - testFuelConsumptionEmissions_Diesel, [390](#)
 - testFuelLookup_Diesel, [392](#)
 - testMinimumLoadRatioConstraint_Diesel, [393](#)
 - testMinimumRuntimeConstraint_Diesel, [393](#)
- test_ElectricalLoad.cpp
 - main, [446](#)
 - testConstruct_ElectricalLoad, [446](#)
 - testDataRead_ElectricalLoad, [447](#)
 - testPostConstructionAttributes_ElectricalLoad, [448](#)
- test_Hydro.cpp
 - main, [395](#)
 - testCommit_Hydro, [396](#)
 - testConstruct_Hydro, [397](#)
 - testEfficiencyInterpolation_Hydro, [398](#)
- test_Interpolator.cpp
 - main, [450](#)
 - testBadIndexing1D_Interpolator, [451](#)
 - testConstruct_Interpolator, [451](#)
 - testDataRead1D_Interpolator, [451](#)
 - testDataRead2D_Interpolator, [453](#)
 - testInterpolation1D_Interpolator, [455](#)
 - testInterpolation2D_Interpolator, [456](#)
 - testInvalidInterpolation1D_Interpolator, [457](#)
 - testInvalidInterpolation2D_Interpolator, [459](#)
- test_Lilon.cpp
 - main, [436](#)
 - testBadConstruct_Lilon, [437](#)
 - testCommitCharge_Lilon, [437](#)
 - testCommitDischarge_Lilon, [438](#)
 - testConstruct_Lilon, [439](#)
- test_Model.cpp
 - main, [462](#)
 - testAddDiesel_Model, [464](#)
 - testAddHydro_Model, [464](#)
 - testAddHydroResource_Model, [465](#)
 - testAddLilon_Model, [466](#)
 - testAddSolar_Model, [467](#)
 - testAddSolar_productionOverride_Model, [467](#)
 - testAddSolarResource_Model, [468](#)
 - testAddTidal_Model, [469](#)
 - testAddTidalResource_Model, [470](#)
 - testAddWave_Model, [471](#)
 - testAddWaveResource_Model, [472](#)
 - testAddWind_Model, [473](#)
 - testAddWindResource_Model, [474](#)

- testBadConstruct_Model, 475
- testConstruct_Model, 476
- testEconomics_Model, 476
- testElectricalLoadData_Model, 477
- testFuelConsumptionEmissions_Model, 478
- testLoadBalance_Model, 479
- testOperatingReserve_Model, 481
- testPostConstructionAttributes_Model, 483
- test_Noncombustion.cpp
 - main, 400
 - testConstruct_Noncombustion, 401
- test_Production.cpp
 - main, 433
 - testBadConstruct_Production, 433
 - testConstruct_Production, 434
- test_Renewable.cpp
 - main, 403
 - testConstruct_Renewable, 403
- test_Resources.cpp
 - main, 484
 - testAddHydroResource_Resources, 486
 - testAddSolarResource_Resources, 487
 - testAddTidalResource_Resources, 488
 - testAddWaveResource_Resources, 489
 - testAddWindResource_Resources, 491
 - testBadAdd_Resources, 492
 - testConstruct_Resources, 494
- test_Solar.cpp
 - main, 405
 - testBadConstruct_Solar, 406
 - testCommit_Solar, 406
 - testConstruct_Solar, 408
 - testDetailed_Solar, 409
 - testEconomics_Solar, 411
 - testProductionConstraint_Solar, 411
 - testProductionOverride_Solar, 412
- test_Storage.cpp
 - main, 441
 - testBadConstruct_Storage, 442
 - testConstruct_Storage, 442
- test_Tidal.cpp
 - main, 414
 - testBadConstruct_Tidal, 415
 - testCommit_Tidal, 416
 - testConstruct_Tidal, 417
 - testEconomics_Tidal, 418
 - testProductionConstraint_Tidal, 418
- test_Wave.cpp
 - main, 420
 - testBadConstruct_Wave, 421
 - testCommit_Wave, 421
 - testConstruct_Wave, 423
 - testConstructLookup_Wave, 424
 - testEconomics_Wave, 424
 - testProductionConstraint_Wave, 425
 - testProductionLookup_Wave, 425
- test_Wind.cpp
 - main, 427
 - testBadConstruct_Wind, 428
 - testCommit_Wind, 429
 - testConstruct_Wind, 430
 - testEconomics_Wind, 431
 - testProductionConstraint_Wind, 431
- testAddDiesel_Model
 - test_Model.cpp, 464
- testAddHydro_Model
 - test_Model.cpp, 464
- testAddHydroResource_Model
 - test_Model.cpp, 465
- testAddHydroResource_Resources
 - test_Resources.cpp, 486
- testAddLilon_Model
 - test_Model.cpp, 466
- testAddSolar_Model
 - test_Model.cpp, 467
- testAddSolar_productionOverride_Model
 - test_Model.cpp, 467
- testAddSolarResource_Model
 - test_Model.cpp, 468
- testAddSolarResource_Resources
 - test_Resources.cpp, 487
- testAddTidal_Model
 - test_Model.cpp, 469
- testAddTidalResource_Model
 - test_Model.cpp, 470
- testAddTidalResource_Resources
 - test_Resources.cpp, 488
- testAddWave_Model
 - test_Model.cpp, 471
- testAddWaveResource_Model
 - test_Model.cpp, 472
- testAddWaveResource_Resources
 - test_Resources.cpp, 489
- testAddWind_Model
 - test_Model.cpp, 473
- testAddWindResource_Model
 - test_Model.cpp, 474
- testAddWindResource_Resources
 - test_Resources.cpp, 491
- testBadAdd_Resources
 - test_Resources.cpp, 492
- testBadConstruct_Diesel
 - test_Diesel.cpp, 385
- testBadConstruct_Lilon
 - test_Lilon.cpp, 437
- testBadConstruct_Model
 - test_Model.cpp, 475
- testBadConstruct_Production
 - test_Production.cpp, 433
- testBadConstruct_Solar
 - test_Solar.cpp, 406
- testBadConstruct_Storage
 - test_Storage.cpp, 442
- testBadConstruct_Tidal
 - test_Tidal.cpp, 415
- testBadConstruct_Wave

- test_Wave.cpp, 421
- testBadConstruct_Wind
 - test_Wind.cpp, 428
- testBadIndexing1D_Interpolator
 - test_Interpolator.cpp, 451
- testCapacityConstraint_Diesel
 - test_Diesel.cpp, 386
- testCommit_Diesel
 - test_Diesel.cpp, 386
- testCommit_Hydro
 - test_Hydro.cpp, 396
- testCommit_Solar
 - test_Solar.cpp, 406
- testCommit_Tidal
 - test_Tidal.cpp, 416
- testCommit_Wave
 - test_Wave.cpp, 421
- testCommit_Wind
 - test_Wind.cpp, 429
- testCommitCharge_Lilon
 - test_Lilon.cpp, 437
- testCommitDischarge_Lilon
 - test_Lilon.cpp, 438
- testConstruct_Combustion
 - test_Combustion.cpp, 382
- testConstruct_Controller
 - test_Controller.cpp, 445
- testConstruct_Diesel
 - test_Diesel.cpp, 388
- testConstruct_ElectricalLoad
 - test_ElectricalLoad.cpp, 446
- testConstruct_Hydro
 - test_Hydro.cpp, 397
- testConstruct_Interpolator
 - test_Interpolator.cpp, 451
- testConstruct_Lilon
 - test_Lilon.cpp, 439
- testConstruct_Model
 - test_Model.cpp, 476
- testConstruct_Noncombustion
 - test_Noncombustion.cpp, 401
- testConstruct_Production
 - test_Production.cpp, 434
- testConstruct_Renewable
 - test_Renewable.cpp, 403
- testConstruct_Resources
 - test_Resources.cpp, 494
- testConstruct_Solar
 - test_Solar.cpp, 408
- testConstruct_Storage
 - test_Storage.cpp, 442
- testConstruct_Tidal
 - test_Tidal.cpp, 417
- testConstruct_Wave
 - test_Wave.cpp, 423
- testConstruct_Wind
 - test_Wind.cpp, 430
- testConstructLookup_Diesel
 - test_Diesel.cpp, 389
- testConstructLookup_Wave
 - test_Wave.cpp, 424
- testDataRead1D_Interpolator
 - test_Interpolator.cpp, 451
- testDataRead2D_Interpolator
 - test_Interpolator.cpp, 453
- testDataRead_ElectricalLoad
 - test_ElectricalLoad.cpp, 447
- testDetailed_Solar
 - test_Solar.cpp, 409
- testEconomics_Diesel
 - test_Diesel.cpp, 389
- testEconomics_Model
 - test_Model.cpp, 476
- testEconomics_Solar
 - test_Solar.cpp, 411
- testEconomics_Tidal
 - test_Tidal.cpp, 418
- testEconomics_Wave
 - test_Wave.cpp, 424
- testEconomics_Wind
 - test_Wind.cpp, 431
- testEfficiencyInterpolation_Hydro
 - test_Hydro.cpp, 398
- testElectricalLoadData_Model
 - test_Model.cpp, 477
- testFloatEquals
 - testing_utils.cpp, 497
 - testing_utils.h, 503
- testFuelConsumptionEmissions_Diesel
 - test_Diesel.cpp, 390
- testFuelConsumptionEmissions_Model
 - test_Model.cpp, 478
- testFuelLookup_Diesel
 - test_Diesel.cpp, 392
- testGreaterThan
 - testing_utils.cpp, 497
 - testing_utils.h, 504
- testGreaterThanOrEqualTo
 - testing_utils.cpp, 498
 - testing_utils.h, 505
- testing_utils.cpp
 - expectedErrorNotDetected, 495
 - printGold, 496
 - printGreen, 496
 - printRed, 496
 - testFloatEquals, 497
 - testGreaterThan, 497
 - testGreaterThanOrEqualTo, 498
 - testLessThan, 499
 - testLessThanOrEqualTo, 499
 - testTruth, 500
- testing_utils.h
 - expectedErrorNotDetected, 502
 - FLOAT_TOLERANCE, 502
 - printGold, 502
 - printGreen, 503

- printRed, [503](#)
- testFloatEquals, [503](#)
- testGreaterThan, [504](#)
- testGreaterThanOrEqualTo, [505](#)
- testLessThan, [505](#)
- testLessThanOrEqualTo, [506](#)
- testTruth, [506](#)
- testInterpolation1D_Interpolator
 - test_Interpolator.cpp, [455](#)
- testInterpolation2D_Interpolator
 - test_Interpolator.cpp, [456](#)
- testInvalidInterpolation1D_Interpolator
 - test_Interpolator.cpp, [457](#)
- testInvalidInterpolation2D_Interpolator
 - test_Interpolator.cpp, [459](#)
- testLessThan
 - testing_utils.cpp, [499](#)
 - testing_utils.h, [505](#)
- testLessThanOrEqualTo
 - testing_utils.cpp, [499](#)
 - testing_utils.h, [506](#)
- testLoadBalance_Model
 - test_Model.cpp, [479](#)
- testMinimumLoadRatioConstraint_Diesel
 - test_Diesel.cpp, [393](#)
- testMinimumRuntimeConstraint_Diesel
 - test_Diesel.cpp, [393](#)
- testOperatingReserve_Model
 - test_Model.cpp, [481](#)
- testPostConstructionAttributes_ElectricalLoad
 - test_ElectricalLoad.cpp, [448](#)
- testPostConstructionAttributes_Model
 - test_Model.cpp, [483](#)
- testProductionConstraint_Solar
 - test_Solar.cpp, [411](#)
- testProductionConstraint_Tidal
 - test_Tidal.cpp, [418](#)
- testProductionConstraint_Wave
 - test_Wave.cpp, [425](#)
- testProductionConstraint_Wind
 - test_Wind.cpp, [431](#)
- testProductionLookup_Wave
 - test_Wave.cpp, [425](#)
- testProductionOverride_Solar
 - test_Solar.cpp, [412](#)
- testTruth
 - testing_utils.cpp, [500](#)
 - testing_utils.h, [506](#)
- TIDAL
 - Renewable.h, [314](#)
- Tidal, [251](#)
 - __checkInputs, [255](#)
 - __computeCubicProductionkW, [256](#)
 - __computeExponentialProductionkW, [256](#)
 - __computeLookupProductionkW, [257](#)
 - __getGenericCapitalCost, [258](#)
 - __getGenericOpMaintCost, [258](#)
 - __writeSummary, [258](#)
 - __writeTimeSeries, [260](#)
 - ~Tidal, [255](#)
 - commit, [260](#)
 - computeProductionkW, [261](#)
 - design_speed_ms, [263](#)
 - handleReplacement, [262](#)
 - power_model, [263](#)
 - power_model_string, [263](#)
 - Tidal, [253](#)
- Tidal.h
 - N_TIDAL_POWER_PRODUCTION_MODELS, [317](#)
 - TIDAL_POWER_CUBIC, [317](#)
 - TIDAL_POWER_EXPONENTIAL, [317](#)
 - TIDAL_POWER_LOOKUP, [317](#)
 - TidalPowerProductionModel, [317](#)
- TIDAL_POWER_CUBIC
 - Tidal.h, [317](#)
- TIDAL_POWER_EXPONENTIAL
 - Tidal.h, [317](#)
- TIDAL_POWER_LOOKUP
 - Tidal.h, [317](#)
- TidalInputs, [264](#)
 - capital_cost, [265](#)
 - design_speed_ms, [265](#)
 - firmness_factor, [265](#)
 - operation_maintenance_cost_kWh, [265](#)
 - power_model, [265](#)
 - renewable_inputs, [266](#)
 - resource_key, [266](#)
- TidalPowerProductionModel
 - Tidal.h, [317](#)
- time_since_last_start_hrs
 - Diesel, [58](#)
- time_vec_hrs
 - ElectricalLoad, [67](#)
- total_discharge_kWh
 - Storage, [248](#)
- total_dispatch_discharge_kWh
 - Model, [151](#)
- total_dispatch_kWh
 - Production, [178](#)
- total_emissions
 - Combustion, [23](#)
 - Model, [152](#)
- total_fuel_consumed_L
 - Combustion, [23](#)
 - Model, [152](#)
- total_renewable_dispatch_kWh
 - Model, [152](#)
- TURBINE_EFFICIENCY_INTERP_KEY
 - Hydro.h, [310](#)
- turbine_flow_vec_m3hr
 - Hydro, [88](#)
- turbine_type
 - Hydro, [88](#)
 - HydroInputs, [91](#)
- type

- Combustion, [24](#)
- Noncombustion, [161](#)
- Renewable, [189](#)
- Storage, [248](#)
- type_str
 - Production, [179](#)
 - Storage, [249](#)
- value
 - PYBIND11_Combustion.cpp, [332](#)
 - PYBIND11_Controller.cpp, [359](#)
 - PYBIND11_Hydro.cpp, [338](#), [339](#)
 - PYBIND11_Noncombustion.cpp, [340](#)
 - PYBIND11_Renewable.cpp, [347](#)
 - PYBIND11_Solar.cpp, [349](#)
 - PYBIND11_Storage.cpp, [370](#)
 - PYBIND11_Tidal.cpp, [352](#)
 - PYBIND11_Wave.cpp, [354](#)
 - PYBIND11_Wind.cpp, [356](#), [357](#)
- WAVE
 - Renewable.h, [314](#)
- Wave, [266](#)
 - __checkInputs, [270](#)
 - __computeGaussianProductionkW, [271](#)
 - __computeLookupProductionkW, [272](#)
 - __computeParaboloidProductionkW, [272](#)
 - __getGenericCapitalCost, [274](#)
 - __getGenericOpMaintCost, [274](#)
 - __writeSummary, [275](#)
 - __writeTimeSeries, [276](#)
 - ~Wave, [270](#)
 - commit, [277](#)
 - computeProductionkW, [278](#)
 - design_energy_period_s, [280](#)
 - design_significant_wave_height_m, [280](#)
 - handleReplacement, [279](#)
 - power_model, [280](#)
 - power_model_string, [280](#)
 - Wave, [268](#)
- Wave.h
 - N_WAVE_POWER_PRODUCTION_MODELS, [318](#)
 - WAVE_POWER_GAUSSIAN, [318](#)
 - WAVE_POWER_LOOKUP, [318](#)
 - WAVE_POWER_PARABOLOID, [318](#)
 - WavePowerProductionModel, [318](#)
- WAVE_POWER_GAUSSIAN
 - Wave.h, [318](#)
- WAVE_POWER_LOOKUP
 - Wave.h, [318](#)
- WAVE_POWER_PARABOLOID
 - Wave.h, [318](#)
- WaveInputs, [281](#)
 - capital_cost, [282](#)
 - design_energy_period_s, [282](#)
 - design_significant_wave_height_m, [282](#)
 - firmness_factor, [282](#)
 - operation_maintenance_cost_kWh, [282](#)
 - path_2_normalized_performance_matrix, [283](#)
 - power_model, [283](#)
 - renewable_inputs, [283](#)
 - resource_key, [283](#)
- WavePowerProductionModel
 - Wave.h, [318](#)
- WIND
 - Renewable.h, [314](#)
- Wind, [284](#)
 - __checkInputs, [287](#)
 - __computeCubicProductionkW, [288](#)
 - __computeExponentialProductionkW, [289](#)
 - __computeLookupProductionkW, [290](#)
 - __getGenericCapitalCost, [290](#)
 - __getGenericOpMaintCost, [290](#)
 - __writeSummary, [291](#)
 - __writeTimeSeries, [292](#)
 - ~Wind, [287](#)
 - commit, [293](#)
 - computeProductionkW, [294](#)
 - design_speed_ms, [296](#)
 - handleReplacement, [295](#)
 - power_model, [296](#)
 - power_model_string, [296](#)
 - Wind, [286](#)
- Wind.h
 - N_WIND_POWER_PRODUCTION_MODELS, [320](#)
 - WIND_POWER_CUBIC, [320](#)
 - WIND_POWER_EXPONENTIAL, [320](#)
 - WIND_POWER_LOOKUP, [320](#)
 - WindPowerProductionModel, [320](#)
- WIND_POWER_CUBIC
 - Wind.h, [320](#)
- WIND_POWER_EXPONENTIAL
 - Wind.h, [320](#)
- WIND_POWER_LOOKUP
 - Wind.h, [320](#)
- WindInputs, [297](#)
 - capital_cost, [298](#)
 - design_speed_ms, [298](#)
 - firmness_factor, [298](#)
 - operation_maintenance_cost_kWh, [298](#)
 - power_model, [298](#)
 - renewable_inputs, [299](#)
 - resource_key, [299](#)
- WindPowerProductionModel
 - Wind.h, [320](#)
- writeResults
 - Combustion, [19](#)
 - Model, [149](#)
 - Noncombustion, [160](#)
 - Renewable, [187](#)
 - Storage, [243](#)
- x_vec
 - InterpolatorStruct1D, [106](#)
 - InterpolatorStruct2D, [108](#)
- y_vec

InterpolatorStruct1D, [106](#)

InterpolatorStruct2D, [108](#)

z_matrix

InterpolatorStruct2D, [108](#)