

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()	13
4.1.3 Member Function Documentation	14
4.1.3.1 __checkInputs()	14
4.1.3.2 __writeSummary()	14
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()	18
4.1.3.11 writeResults()	19
4.1.4 Member Data Documentation	19
4.1.4.1 CH4_emissions_intensity_kgL	19
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	20
4.1.4.7 fuel_consumption_vec_L	20
4.1.4.8 fuel_cost_L	21
4.1.4.9 fuel_cost_vec	21
4.1.4.10 fuel_mode	21
4.1.4.11 fuel_mode_str	21
4.1.4.12 linear_fuel_intercept_LkWh	21
4.1.4.13 linear_fuel_slope_LkWh	21
4.1.4.14 nominal_fuel_escalation_annual	22

4.1.4.15 NOx_emissions_intensity_kgL	22
4.1.4.16 NOx_emissions_vec_kg	22
4.1.4.17 PM_emissions_intensity_kgL	22
4.1.4.18 PM_emissions_vec_kg	22
4.1.4.19 real_fuel_escalation_annual	22
4.1.4.20 SOx_emissions_intensity_kgL	23
4.1.4.21 SOx_emissions_vec_kg	23
4.1.4.22 total_emissions	23
4.1.4.23 total_fuel_consumed_L	23
4.1.4.24 type	23
4.2 CombustionInputs Struct Reference	24
4.2.1 Detailed Description	24
4.2.2 Member Data Documentation	24
4.2.2.1 fuel_mode	24
4.2.2.2 nominal_fuel_escalation_annual	25
4.2.2.3 path_2_fuel_interp_data	25
4.2.2.4 production_inputs	25
4.3 Controller Class Reference	25
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	27
4.3.3.1 __applyCycleChargingControl_CHARGING()	27
4.3.3.2 __applyCycleChargingControl_DISCHARGING()	28
4.3.3.3 __applyLoadFollowingControl_CHARGING()	29
4.3.3.4 __applyLoadFollowingControl_DISCHARGING()	30
4.3.3.5 __computeNetLoad()	32
4.3.3.6 __constructCombustionMap()	32
4.3.3.7 __getRenewableProduction()	34
4.3.3.8 __handleCombustionDispatch()	35
4.3.3.9 __handleNoncombustionDispatch()	36
4.3.3.10 __handleStorageCharging() [1/2]	37
4.3.3.11 __handleStorageCharging() [2/2]	38
4.3.3.12 __handleStorageDischarging()	40
4.3.3.13 applyDispatchControl()	41
4.3.3.14 clear()	42
4.3.3.15 init()	42
4.3.3.16 setControlMode()	43
4.3.4 Member Data Documentation	43
4.3.4.1 combustion_map	43
4.3.4.2 control_mode	44

4.3.4.3 control_string	44
4.3.4.4 missed_load_vec_kW	44
4.3.4.5 net_load_vec_kW	44
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()	48
4.4.3 Member Function Documentation	48
4.4.3.1 __checkInputs()	48
4.4.3.2 __getGenericCapitalCost()	50
4.4.3.3 __getGenericFuelIntercept()	50
4.4.3.4 __getGenericFuelSlope()	51
4.4.3.5 __getGenericOpMaintCost()	51
4.4.3.6 __handleStartStop()	51
4.4.3.7 __writeSummary()	52
4.4.3.8 __writeTimeSeries()	54
4.4.3.9 commit()	55
4.4.3.10 handleReplacement()	56
4.4.3.11 requestProductionkW()	56
4.4.4 Member Data Documentation	57
4.4.4.1 minimum_load_ratio	57
4.4.4.2 minimum_runtime_hrs	57
4.4.4.3 time_since_last_start_hrs	58
4.5 DieselInputs Struct Reference	58
4.5.1 Detailed Description	59
4.5.2 Member Data Documentation	59
4.5.2.1 capital_cost	60
4.5.2.2 CH4_emissions_intensity_kgL	60
4.5.2.3 CO2_emissions_intensity_kgL	60
4.5.2.4 CO_emissions_intensity_kgL	60
4.5.2.5 combustion_inputs	60
4.5.2.6 fuel_cost_L	60
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	61
4.5.2.10 minimum_runtime_hrs	61
4.5.2.11 NOx_emissions_intensity_kgL	61
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	62
4.6 ElectricalLoad Class Reference	62
4.6.1 Detailed Description	63
4.6.2 Constructor & Destructor Documentation	63
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()	64
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	66
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	67
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	68
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	69
4.8.1 Detailed Description	71
4.8.2 Constructor & Destructor Documentation	71
4.8.2.1 Hydro() [1/2]	72
4.8.2.2 Hydro() [2/2]	72
4.8.2.3 ~Hydro()	73
4.8.3 Member Function Documentation	73
4.8.3.1 __checkInputs()	73
4.8.3.2 __flowToPower()	74
4.8.3.3 __getAvailableFlow()	75
4.8.3.4 __getGenericCapitalCost()	75
4.8.3.5 __getGenericOpMaintCost()	76
4.8.3.6 __getMaximumFlowm3hr()	76

4.8.3.7	__getMinimumFlowm3hr()	77
4.8.3.8	__powerToFlow()	78
4.8.3.9	__updateState()	78
4.8.3.10	__writeSummary()	79
4.8.3.11	__writeTimeSeries()	81
4.8.3.12	commit()	81
4.8.3.13	handleReplacement()	82
4.8.3.14	requestProductionkW()	82
4.8.4	Member Data Documentation	83
4.8.4.1	fluid_density_kgm3	83
4.8.4.2	init_reservoir_state	84
4.8.4.3	maximum_flow_m3hr	84
4.8.4.4	minimum_flow_m3hr	84
4.8.4.5	net_head_m	84
4.8.4.6	reservoir_capacity_m3	84
4.8.4.7	stored_volume_m3	84
4.8.4.8	stored_volume_vec_m3	85
4.8.4.9	turbine_flow_vec_m3hr	85
4.8.4.10	turbine_type	85
4.9	HydroInputs Struct Reference	85
4.9.1	Detailed Description	86
4.9.2	Member Data Documentation	86
4.9.2.1	capital_cost	86
4.9.2.2	fluid_density_kgm3	86
4.9.2.3	init_reservoir_state	87
4.9.2.4	net_head_m	87
4.9.2.5	noncombustion_inputs	87
4.9.2.6	operation_maintenance_cost_kWh	87
4.9.2.7	reservoir_capacity_m3	87
4.9.2.8	resource_key	87
4.9.2.9	turbine_type	88
4.10	Interpolator Class Reference	88
4.10.1	Detailed Description	89
4.10.2	Constructor & Destructor Documentation	89
4.10.2.1	Interpolator()	89
4.10.2.2	~Interpolator()	90
4.10.3	Member Function Documentation	90
4.10.3.1	__checkBounds1D()	90
4.10.3.2	__checkBounds2D()	91
4.10.3.3	__checkDataKey1D()	92
4.10.3.4	__checkDataKey2D()	92
4.10.3.5	__getDataStringMatrix()	92

4.10.3.6	__getInterpolationIndex()	93
4.10.3.7	__isNonNumeric()	94
4.10.3.8	__readData1D()	94
4.10.3.9	__readData2D()	95
4.10.3.10	__splitCommaSeparatedString()	97
4.10.3.11	__throwReadError()	97
4.10.3.12	addData1D()	98
4.10.3.13	addData2D()	98
4.10.3.14	interp1D()	99
4.10.3.15	interp2D()	99
4.10.4	Member Data Documentation	100
4.10.4.1	interp_map_1D	100
4.10.4.2	interp_map_2D	101
4.10.4.3	path_map_1D	101
4.10.4.4	path_map_2D	101
4.11	InterpolatorStruct1D Struct Reference	101
4.11.1	Detailed Description	102
4.11.2	Member Data Documentation	102
4.11.2.1	max_x	102
4.11.2.2	min_x	102
4.11.2.3	n_points	102
4.11.2.4	x_vec	102
4.11.2.5	y_vec	102
4.12	InterpolatorStruct2D Struct Reference	103
4.12.1	Detailed Description	103
4.12.2	Member Data Documentation	103
4.12.2.1	max_x	103
4.12.2.2	max_y	104
4.12.2.3	min_x	104
4.12.2.4	min_y	104
4.12.2.5	n_cols	104
4.12.2.6	n_rows	104
4.12.2.7	x_vec	104
4.12.2.8	y_vec	105
4.12.2.9	z_matrix	105
4.13	Lilon Class Reference	105
4.13.1	Detailed Description	108
4.13.2	Constructor & Destructor Documentation	108
4.13.2.1	Lilon() <small>[1/2]</small>	108
4.13.2.2	Lilon() <small>[2/2]</small>	108
4.13.2.3	~Lilon()	109
4.13.3	Member Function Documentation	109

4.13.3.1	__checkInputs()	109
4.13.3.2	__getBcal()	112
4.13.3.3	__getEacal()	112
4.13.3.4	__getGenericCapitalCost()	113
4.13.3.5	__getGenericOpMaintCost()	113
4.13.3.6	__handleDegradation()	114
4.13.3.7	__modelDegradation()	114
4.13.3.8	__toggleDepleted()	115
4.13.3.9	__writeSummary()	115
4.13.3.10	__writeTimeSeries()	116
4.13.3.11	commitCharge()	117
4.13.3.12	commitDischarge()	118
4.13.3.13	getAcceptablekW()	119
4.13.3.14	getAvailablekW()	120
4.13.3.15	handleReplacement()	120
4.13.4	Member Data Documentation	121
4.13.4.1	charging_efficiency	121
4.13.4.2	degradation_a_cal	121
4.13.4.3	degradation_alpha	121
4.13.4.4	degradation_B_hat_cal_0	121
4.13.4.5	degradation_beta	121
4.13.4.6	degradation_Ea_cal_0	122
4.13.4.7	degradation_r_cal	122
4.13.4.8	degradation_s_cal	122
4.13.4.9	discharging_efficiency	122
4.13.4.10	dynamic_energy_capacity_kWh	122
4.13.4.11	gas_constant_JmolK	122
4.13.4.12	hysteresis_SOC	123
4.13.4.13	init_SOC	123
4.13.4.14	max_SOC	123
4.13.4.15	min_SOC	123
4.13.4.16	replace_SOH	123
4.13.4.17	SOH	123
4.13.4.18	SOH_vec	124
4.13.4.19	temperature_K	124
4.14	LilonInputs Struct Reference	124
4.14.1	Detailed Description	125
4.14.2	Member Data Documentation	126
4.14.2.1	capital_cost	126
4.14.2.2	charging_efficiency	126
4.14.2.3	degradation_a_cal	126
4.14.2.4	degradation_alpha	126

4.14.2.5 degradation_B_hat_cal_0	126
4.14.2.6 degradation_beta	127
4.14.2.7 degradation_Ea_cal_0	127
4.14.2.8 degradation_r_cal	127
4.14.2.9 degradation_s_cal	127
4.14.2.10 discharging_efficiency	127
4.14.2.11 gas_constant_JmolK	127
4.14.2.12 hysteresis_SOC	128
4.14.2.13 init_SOC	128
4.14.2.14 max_SOC	128
4.14.2.15 min_SOC	128
4.14.2.16 operation_maintenance_cost_kWh	128
4.14.2.17 replace_SOH	128
4.14.2.18 storage_inputs	129
4.14.2.19 temperature_K	129
4.15 Model Class Reference	129
4.15.1 Detailed Description	131
4.15.2 Constructor & Destructor Documentation	131
4.15.2.1 Model() [1/2]	131
4.15.2.2 Model() [2/2]	132
4.15.2.3 ~Model()	132
4.15.3 Member Function Documentation	132
4.15.3.1 __checkInputs()	132
4.15.3.2 __computeEconomics()	133
4.15.3.3 __computeFuelAndEmissions()	133
4.15.3.4 __computeLevellizedCostOfEnergy()	134
4.15.3.5 __computeNetPresentCost()	134
4.15.3.6 __writeSummary()	135
4.15.3.7 __writeTimeSeries()	138
4.15.3.8 addDiesel()	139
4.15.3.9 addHydro()	139
4.15.3.10 addLilon()	140
4.15.3.11 addResource() [1/2]	140
4.15.3.12 addResource() [2/2]	140
4.15.3.13 addSolar()	142
4.15.3.14 addTidal()	142
4.15.3.15 addWave()	143
4.15.3.16 addWind()	143
4.15.3.17 clear()	143
4.15.3.18 reset()	144
4.15.3.19 run()	144
4.15.3.20 writeResults()	145

4.15.4 Member Data Documentation	146
4.15.4.1 combustion_ptr_vec	146
4.15.4.2 controller	146
4.15.4.3 electrical_load	147
4.15.4.4 levlized_cost_of_energy_kWh	147
4.15.4.5 net_present_cost	147
4.15.4.6 noncombustion_ptr_vec	147
4.15.4.7 renewable_ptr_vec	147
4.15.4.8 resources	147
4.15.4.9 storage_ptr_vec	148
4.15.4.10 total_dispatch_discharge_kWh	148
4.15.4.11 total_emissions	148
4.15.4.12 total_fuel_consumed_L	148
4.16 ModelInputs Struct Reference	148
4.16.1 Detailed Description	149
4.16.2 Member Data Documentation	149
4.16.2.1 control_mode	149
4.16.2.2 path_2_electrical_load_time_series	149
4.17 Noncombustion Class Reference	149
4.17.1 Detailed Description	151
4.17.2 Constructor & Destructor Documentation	151
4.17.2.1 Noncombustion() [1/2]	151
4.17.2.2 Noncombustion() [2/2]	151
4.17.2.3 ~Noncombustion()	152
4.17.3 Member Function Documentation	152
4.17.3.1 __checkInputs()	152
4.17.3.2 __handleStartStop()	152
4.17.3.3 __writeSummary()	153
4.17.3.4 __writeTimeSeries()	153
4.17.3.5 commit() [1/2]	153
4.17.3.6 commit() [2/2]	154
4.17.3.7 computeEconomics()	154
4.17.3.8 handleReplacement()	155
4.17.3.9 requestProductionkW() [1/2]	155
4.17.3.10 requestProductionkW() [2/2]	155
4.17.3.11 writeResults()	156
4.17.4 Member Data Documentation	156
4.17.4.1 resource_key	156
4.17.4.2 type	157
4.18 NoncombustionInputs Struct Reference	157
4.18.1 Detailed Description	157
4.18.2 Member Data Documentation	157

4.18.2.1 production_inputs	158
4.19 Production Class Reference	158
4.19.1 Detailed Description	160
4.19.2 Constructor & Destructor Documentation	160
4.19.2.1 Production() [1/2]	161
4.19.2.2 Production() [2/2]	161
4.19.2.3 ~Production()	162
4.19.3 Member Function Documentation	162
4.19.3.1 __checkInputs()	162
4.19.3.2 commit()	163
4.19.3.3 computeEconomics()	164
4.19.3.4 computeRealDiscountAnnual()	165
4.19.3.5 handleReplacement()	165
4.19.4 Member Data Documentation	166
4.19.4.1 capacity_kW	166
4.19.4.2 capital_cost	166
4.19.4.3 capital_cost_vec	166
4.19.4.4 curtailment_vec_kW	166
4.19.4.5 dispatch_vec_kW	167
4.19.4.6 interpolator	167
4.19.4.7 is_running	167
4.19.4.8 is_running_vec	167
4.19.4.9 is_sunk	167
4.19.4.10 levlized_cost_of_energy_kWh	167
4.19.4.11 n_points	168
4.19.4.12 n_replacements	168
4.19.4.13 n_starts	168
4.19.4.14 n_years	168
4.19.4.15 net_present_cost	168
4.19.4.16 nominal_discount_annual	168
4.19.4.17 nominal_inflation_annual	169
4.19.4.18 operation_maintenance_cost_kWh	169
4.19.4.19 operation_maintenance_cost_vec	169
4.19.4.20 print_flag	169
4.19.4.21 production_vec_kW	169
4.19.4.22 real_discount_annual	169
4.19.4.23 replace_running_hrs	170
4.19.4.24 running_hours	170
4.19.4.25 storage_vec_kW	170
4.19.4.26 total_dispatch_kWh	170
4.19.4.27 type_str	170
4.20 ProductionInputs Struct Reference	170

4.20.1 Detailed Description	171
4.20.2 Member Data Documentation	171
4.20.2.1 capacity_kW	171
4.20.2.2 is_sunk	171
4.20.2.3 nominal_discount_annual	171
4.20.2.4 nominal_inflation_annual	172
4.20.2.5 print_flag	172
4.20.2.6 replace_running_hrs	172
4.21 Renewable Class Reference	172
4.21.1 Detailed Description	174
4.21.2 Constructor & Destructor Documentation	174
4.21.2.1 Renewable() [1/2]	174
4.21.2.2 Renewable() [2/2]	174
4.21.2.3 ~Renewable()	175
4.21.3 Member Function Documentation	175
4.21.3.1 __checkInputs()	175
4.21.3.2 __handleStartStop()	176
4.21.3.3 __writeSummary()	176
4.21.3.4 __writeTimeSeries()	176
4.21.3.5 commit()	176
4.21.3.6 computeEconomics()	177
4.21.3.7 computeProductionkW() [1/2]	177
4.21.3.8 computeProductionkW() [2/2]	178
4.21.3.9 handleReplacement()	178
4.21.3.10 writeResults()	178
4.21.4 Member Data Documentation	179
4.21.4.1 resource_key	179
4.21.4.2 type	180
4.22 RenewableInputs Struct Reference	180
4.22.1 Detailed Description	180
4.22.2 Member Data Documentation	180
4.22.2.1 production_inputs	181
4.23 Resources Class Reference	181
4.23.1 Detailed Description	182
4.23.2 Constructor & Destructor Documentation	182
4.23.2.1 Resources()	182
4.23.2.2 ~Resources()	182
4.23.3 Member Function Documentation	183
4.23.3.1 __checkResourceKey1D() [1/2]	183
4.23.3.2 __checkResourceKey1D() [2/2]	183
4.23.3.3 __checkResourceKey2D()	184
4.23.3.4 __checkTimePoint()	185

4.23.3.5 __readHydroResource()	185
4.23.3.6 __readSolarResource()	186
4.23.3.7 __readTidalResource()	187
4.23.3.8 __readWaveResource()	188
4.23.3.9 __readWindResource()	189
4.23.3.10 __throwLengthError()	190
4.23.3.11 addResource() [1/2]	191
4.23.3.12 addResource() [2/2]	192
4.23.3.13 clear()	193
4.23.4 Member Data Documentation	193
4.23.4.1 path_map_1D	193
4.23.4.2 path_map_2D	193
4.23.4.3 resource_map_1D	194
4.23.4.4 resource_map_2D	194
4.23.4.5 string_map_1D	194
4.23.4.6 string_map_2D	194
4.24 Solar Class Reference	195
4.24.1 Detailed Description	196
4.24.2 Constructor & Destructor Documentation	196
4.24.2.1 Solar() [1/2]	197
4.24.2.2 Solar() [2/2]	197
4.24.2.3 ~Solar()	198
4.24.3 Member Function Documentation	198
4.24.3.1 __checkInputs()	198
4.24.3.2 __getGenericCapitalCost()	198
4.24.3.3 __getGenericOpMaintCost()	199
4.24.3.4 __writeSummary()	199
4.24.3.5 __writeTimeSeries()	200
4.24.3.6 commit()	201
4.24.3.7 computeProductionkW()	202
4.24.3.8 handleReplacement()	202
4.24.4 Member Data Documentation	203
4.24.4.1 derating	203
4.25 SolarInputs Struct Reference	203
4.25.1 Detailed Description	204
4.25.2 Member Data Documentation	204
4.25.2.1 capital_cost	204
4.25.2.2 derating	204
4.25.2.3 operation_maintenance_cost_kWh	205
4.25.2.4 renewable_inputs	205
4.25.2.5 resource_key	205
4.26 Storage Class Reference	205

4.26.1 Detailed Description	208
4.26.2 Constructor & Destructor Documentation	208
4.26.2.1 Storage() [1/2]	208
4.26.2.2 Storage() [2/2]	208
4.26.2.3 ~Storage()	209
4.26.3 Member Function Documentation	209
4.26.3.1 __checkInputs()	209
4.26.3.2 __computeRealDiscountAnnual()	210
4.26.3.3 __writeSummary()	211
4.26.3.4 __writeTimeSeries()	211
4.26.3.5 commitCharge()	211
4.26.3.6 commitDischarge()	211
4.26.3.7 computeEconomics()	212
4.26.3.8 getAcceptablekW()	212
4.26.3.9 getAvailablekW()	213
4.26.3.10 handleReplacement()	213
4.26.3.11 writeResults()	213
4.26.4 Member Data Documentation	214
4.26.4.1 capital_cost	214
4.26.4.2 capital_cost_vec	214
4.26.4.3 charge_kWh	215
4.26.4.4 charge_vec_kWh	215
4.26.4.5 charging_power_vec_kW	215
4.26.4.6 discharging_power_vec_kW	215
4.26.4.7 energy_capacity_kWh	215
4.26.4.8 interpolator	215
4.26.4.9 is_depleted	216
4.26.4.10 is_sunk	216
4.26.4.11 levlized_cost_of_energy_kWh	216
4.26.4.12 n_points	216
4.26.4.13 n_replacements	216
4.26.4.14 n_years	216
4.26.4.15 net_present_cost	217
4.26.4.16 nominal_discount_annual	217
4.26.4.17 nominal_inflation_annual	217
4.26.4.18 operation_maintenance_cost_kWh	217
4.26.4.19 operation_maintenance_cost_vec	217
4.26.4.20 power_capacity_kW	217
4.26.4.21 power_kW	218
4.26.4.22 print_flag	218
4.26.4.23 real_discount_annual	218
4.26.4.24 total_discharge_kWh	218

4.26.4.25 type	218
4.26.4.26 type_str	218
4.27 StorageInputs Struct Reference	219
4.27.1 Detailed Description	219
4.27.2 Member Data Documentation	219
4.27.2.1 energy_capacity_kWh	219
4.27.2.2 is_sunk	219
4.27.2.3 nominal_discount_annual	220
4.27.2.4 nominal_inflation_annual	220
4.27.2.5 power_capacity_kW	220
4.27.2.6 print_flag	220
4.28 Tidal Class Reference	221
4.28.1 Detailed Description	222
4.28.2 Constructor & Destructor Documentation	223
4.28.2.1 Tidal() [1/2]	223
4.28.2.2 Tidal() [2/2]	223
4.28.2.3 ~Tidal()	224
4.28.3 Member Function Documentation	224
4.28.3.1 __checkInputs()	224
4.28.3.2 __computeCubicProductionkW()	225
4.28.3.3 __computeExponentialProductionkW()	226
4.28.3.4 __computeLookupProductionkW()	226
4.28.3.5 __getGenericCapitalCost()	227
4.28.3.6 __getGenericOpMaintCost()	227
4.28.3.7 __writeSummary()	227
4.28.3.8 __writeTimeSeries()	229
4.28.3.9 commit()	229
4.28.3.10 computeProductionkW()	230
4.28.3.11 handleReplacement()	231
4.28.4 Member Data Documentation	232
4.28.4.1 design_speed_ms	232
4.28.4.2 power_model	232
4.28.4.3 power_model_string	232
4.29 TidalInputs Struct Reference	232
4.29.1 Detailed Description	233
4.29.2 Member Data Documentation	233
4.29.2.1 capital_cost	233
4.29.2.2 design_speed_ms	233
4.29.2.3 operation_maintenance_cost_kWh	234
4.29.2.4 power_model	234
4.29.2.5 renewable_inputs	234
4.29.2.6 resource_key	234

4.30 Wave Class Reference	235
4.30.1 Detailed Description	237
4.30.2 Constructor & Destructor Documentation	237
4.30.2.1 Wave() [1/2]	237
4.30.2.2 Wave() [2/2]	237
4.30.2.3 ~Wave()	238
4.30.3 Member Function Documentation	238
4.30.3.1 __checkInputs()	239
4.30.3.2 __computeGaussianProductionkW()	239
4.30.3.3 __computeLookupProductionkW()	240
4.30.3.4 __computeParaboloidProductionkW()	241
4.30.3.5 __getGenericCapitalCost()	241
4.30.3.6 __getGenericOpMaintCost()	242
4.30.3.7 __writeSummary()	242
4.30.3.8 __writeTimeSeries()	244
4.30.3.9 commit()	245
4.30.3.10 computeProductionkW()	245
4.30.3.11 handleReplacement()	246
4.30.4 Member Data Documentation	247
4.30.4.1 design_energy_period_s	247
4.30.4.2 design_significant_wave_height_m	247
4.30.4.3 power_model	247
4.30.4.4 power_model_string	247
4.31 WaveInputs Struct Reference	248
4.31.1 Detailed Description	249
4.31.2 Member Data Documentation	249
4.31.2.1 capital_cost	249
4.31.2.2 design_energy_period_s	249
4.31.2.3 design_significant_wave_height_m	249
4.31.2.4 operation_maintenance_cost_kWh	249
4.31.2.5 path_2_normalized_performance_matrix	250
4.31.2.6 power_model	250
4.31.2.7 renewable_inputs	250
4.31.2.8 resource_key	250
4.32 Wind Class Reference	251
4.32.1 Detailed Description	252
4.32.2 Constructor & Destructor Documentation	253
4.32.2.1 Wind() [1/2]	253
4.32.2.2 Wind() [2/2]	253
4.32.2.3 ~Wind()	254
4.32.3 Member Function Documentation	254
4.32.3.1 __checkInputs()	254

4.32.3.2	__computeExponentialProductionkW()	255
4.32.3.3	__computeLookupProductionkW()	255
4.32.3.4	__getGenericCapitalCost()	256
4.32.3.5	__getGenericOpMaintCost()	256
4.32.3.6	__writeSummary()	257
4.32.3.7	__writeTimeSeries()	258
4.32.3.8	commit()	259
4.32.3.9	computeProductionkW()	259
4.32.3.10	handleReplacement()	261
4.32.4	Member Data Documentation	262
4.32.4.1	design_speed_ms	262
4.32.4.2	power_model	262
4.32.4.3	power_model_string	262
4.33	WindInputs Struct Reference	263
4.33.1	Detailed Description	263
4.33.2	Member Data Documentation	264
4.33.2.1	capital_cost	264
4.33.2.2	design_speed_ms	264
4.33.2.3	operation_maintenance_cost_kWh	264
4.33.2.4	power_model	264
4.33.2.5	renewable_inputs	264
4.33.2.6	resource_key	264
5	File Documentation	265
5.1	header/Controller.h File Reference	265
5.1.1	Detailed Description	266
5.1.2	Enumeration Type Documentation	266
5.1.2.1	ControlMode	266
5.2	header/doxygen_cite.h File Reference	266
5.2.1	Detailed Description	266
5.3	header/ElectricalLoad.h File Reference	267
5.3.1	Detailed Description	267
5.4	header/Interpolator.h File Reference	267
5.4.1	Detailed Description	268
5.5	header/Model.h File Reference	268
5.5.1	Detailed Description	269
5.6	header/Production/Combustion/Combustion.h File Reference	269
5.6.1	Detailed Description	270
5.6.2	Enumeration Type Documentation	270
5.6.2.1	CombustionType	270
5.6.2.2	FuelMode	270
5.7	header/Production/Combustion/Diesel.h File Reference	272

5.7.1 Detailed Description	273
5.8 header/Production/Noncombustion/Hydro.h File Reference	273
5.8.1 Detailed Description	274
5.8.2 Enumeration Type Documentation	274
5.8.2.1 HydroTurbineType	274
5.9 header/Production/Noncombustion/Noncombustion.h File Reference	274
5.9.1 Enumeration Type Documentation	275
5.9.1.1 NoncombustionType	275
5.10 header/Production/Production.h File Reference	275
5.10.1 Detailed Description	276
5.11 header/Production/Renewable/Renewable.h File Reference	276
5.11.1 Detailed Description	277
5.11.2 Enumeration Type Documentation	277
5.11.2.1 RenewableType	277
5.12 header/Production/Renewable/Solar.h File Reference	278
5.12.1 Detailed Description	278
5.13 header/Production/Renewable/Tidal.h File Reference	279
5.13.1 Detailed Description	279
5.13.2 Enumeration Type Documentation	280
5.13.2.1 TidalPowerProductionModel	280
5.14 header/Production/Renewable/Wave.h File Reference	280
5.14.1 Detailed Description	281
5.14.2 Enumeration Type Documentation	281
5.14.2.1 WavePowerProductionModel	281
5.15 header/Production/Renewable/Wind.h File Reference	282
5.15.1 Detailed Description	283
5.15.2 Enumeration Type Documentation	283
5.15.2.1 WindPowerProductionModel	283
5.16 header/Resources.h File Reference	283
5.16.1 Detailed Description	284
5.17 header/std_includes.h File Reference	284
5.17.1 Detailed Description	285
5.18 header/Storage/Lilon.h File Reference	285
5.18.1 Detailed Description	285
5.19 header/Storage/Storage.h File Reference	286
5.19.1 Detailed Description	286
5.19.2 Enumeration Type Documentation	287
5.19.2.1 StorageType	287
5.20 projects/example.cpp File Reference	287
5.20.1 Function Documentation	287
5.20.1.1 main()	288
5.21 pybindings/PYBIND11_PGM.cpp File Reference	289

5.21.1 Detailed Description	290
5.21.2 Function Documentation	290
5.21.2.1 PYBIND11_MODULE()	290
5.22 pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp File Reference	291
5.22.1 Detailed Description	291
5.22.2 Function Documentation	291
5.22.2.1 def_readwrite()	292
5.22.2.2 value() [1/2]	292
5.22.2.3 value() [2/2]	292
5.22.3 Variable Documentation	292
5.22.3.1 def_readwrite	292
5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference	292
5.23.1 Detailed Description	293
5.23.2 Function Documentation	294
5.23.2.1 def()	294
5.23.2.2 def_readwrite() [1/8]	294
5.23.2.3 def_readwrite() [2/8]	294
5.23.2.4 def_readwrite() [3/8]	294
5.23.2.5 def_readwrite() [4/8]	294
5.23.2.6 def_readwrite() [5/8]	295
5.23.2.7 def_readwrite() [6/8]	295
5.23.2.8 def_readwrite() [7/8]	295
5.23.2.9 def_readwrite() [8/8]	295
5.24 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference	295
5.24.1 Detailed Description	296
5.24.2 Function Documentation	296
5.24.2.1 def_readwrite() [1/2]	296
5.24.2.2 def_readwrite() [2/2]	296
5.24.3 Variable Documentation	296
5.24.3.1 def_readwrite	297
5.25 pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp File Reference	297
5.25.1 Detailed Description	297
5.25.2 Function Documentation	298
5.25.2.1 def()	298
5.25.2.2 value() [1/2]	298
5.25.2.3 value() [2/2]	298
5.26 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference	298
5.26.1 Detailed Description	299
5.26.2 Function Documentation	299
5.26.2.1 def()	299
5.26.2.2 def_readwrite() [1/2]	299
5.26.2.3 def_readwrite() [2/2]	299

5.27 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference	300
5.27.1 Detailed Description	300
5.27.2 Function Documentation	301
5.27.2.1 def_readwrite() [1/2]	301
5.27.2.2 def_readwrite() [2/2]	301
5.27.2.3 value() [1/2]	301
5.27.2.4 value() [2/2]	301
5.27.3 Variable Documentation	301
5.27.3.1 def_readwrite	301
5.28 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference	302
5.28.1 Detailed Description	302
5.28.2 Function Documentation	303
5.28.2.1 def_readwrite() [1/3]	303
5.28.2.2 def_readwrite() [2/3]	303
5.28.2.3 def_readwrite() [3/3]	303
5.28.2.4 value() [1/2]	303
5.28.2.5 value() [2/2]	303
5.28.3 Variable Documentation	303
5.28.3.1 def_readwrite	304
5.29 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference	304
5.29.1 Detailed Description	305
5.29.2 Function Documentation	305
5.29.2.1 def_readwrite() [1/2]	305
5.29.2.2 def_readwrite() [2/2]	305
5.29.2.3 value()	305
5.29.3 Variable Documentation	305
5.29.3.1 def_readwrite	305
5.30 pybindings/snippets/PYBIND11_Controller.cpp File Reference	306
5.30.1 Detailed Description	306
5.30.2 Function Documentation	306
5.30.2.1 def() [1/3]	307
5.30.2.2 def() [2/3]	307
5.30.2.3 def() [3/3]	307
5.30.2.4 def_readwrite() [1/2]	307
5.30.2.5 def_readwrite() [2/2]	307
5.30.2.6 value()	307
5.31 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference	308
5.31.1 Detailed Description	308
5.31.2 Function Documentation	308
5.31.2.1 def_readwrite() [1/4]	308
5.31.2.2 def_readwrite() [2/4]	309
5.31.2.3 def_readwrite() [3/4]	309

5.31.2.4 def_readwrite() [4/4]	309
5.32 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference	309
5.32.1 Detailed Description	310
5.32.2 Function Documentation	310
5.32.2.1 def()	310
5.32.2.2 def_readwrite() [1/7]	310
5.32.2.3 def_readwrite() [2/7]	311
5.32.2.4 def_readwrite() [3/7]	311
5.32.2.5 def_readwrite() [4/7]	311
5.32.2.6 def_readwrite() [5/7]	311
5.32.2.7 def_readwrite() [6/7]	311
5.32.2.8 def_readwrite() [7/7]	311
5.33 pybindings/snippets/PYBIND11_Model.cpp File Reference	312
5.33.1 Detailed Description	312
5.33.2 Variable Documentation	312
5.33.2.1 def_readwrite	312
5.34 pybindings/snippets/PYBIND11_Resources.cpp File Reference	313
5.34.1 Detailed Description	313
5.34.2 Function Documentation	313
5.34.2.1 def_readwrite() [1/2]	313
5.34.2.2 def_readwrite() [2/2]	314
5.35 pybindings/snippets/Storage/PYBIND11_Lilon.cpp File Reference	314
5.35.1 Detailed Description	315
5.35.2 Function Documentation	315
5.35.2.1 def()	315
5.35.2.2 def_readwrite() [1/18]	316
5.35.2.3 def_readwrite() [2/18]	316
5.35.2.4 def_readwrite() [3/18]	316
5.35.2.5 def_readwrite() [4/18]	316
5.35.2.6 def_readwrite() [5/18]	316
5.35.2.7 def_readwrite() [6/18]	316
5.35.2.8 def_readwrite() [7/18]	317
5.35.2.9 def_readwrite() [8/18]	317
5.35.2.10 def_readwrite() [9/18]	317
5.35.2.11 def_readwrite() [10/18]	317
5.35.2.12 def_readwrite() [11/18]	317
5.35.2.13 def_readwrite() [12/18]	318
5.35.2.14 def_readwrite() [13/18]	318
5.35.2.15 def_readwrite() [14/18]	318
5.35.2.16 def_readwrite() [15/18]	318
5.35.2.17 def_readwrite() [16/18]	318
5.35.2.18 def_readwrite() [17/18]	318

5.35.2.19 def_readwrite() [18/18]	319
5.36 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference	319
5.36.1 Detailed Description	319
5.36.2 Function Documentation	320
5.36.2.1 def_readwrite() [1/2]	320
5.36.2.2 def_readwrite() [2/2]	320
5.36.2.3 value()	320
5.36.3 Variable Documentation	320
5.36.3.1 def_readwrite	320
5.37 source/Controller.cpp File Reference	321
5.37.1 Detailed Description	321
5.38 source/ElectricalLoad.cpp File Reference	321
5.38.1 Detailed Description	321
5.39 source/Interpolator.cpp File Reference	322
5.39.1 Detailed Description	322
5.40 source/Model.cpp File Reference	322
5.40.1 Detailed Description	322
5.41 source/Production/Combustion/Combustion.cpp File Reference	323
5.41.1 Detailed Description	323
5.42 source/Production/Combustion/Diesel.cpp File Reference	323
5.42.1 Detailed Description	324
5.43 source/Production/Noncombustion/Hydro.cpp File Reference	324
5.43.1 Detailed Description	324
5.44 source/Production/Noncombustion/Noncombustion.cpp File Reference	324
5.44.1 Detailed Description	325
5.45 source/Production/Production.cpp File Reference	325
5.45.1 Detailed Description	325
5.46 source/Production/Renewable/Renewable.cpp File Reference	326
5.46.1 Detailed Description	326
5.47 source/Production/Renewable/Solar.cpp File Reference	326
5.47.1 Detailed Description	327
5.48 source/Production/Renewable/Tidal.cpp File Reference	327
5.48.1 Detailed Description	327
5.49 source/Production/Renewable/Wave.cpp File Reference	327
5.49.1 Detailed Description	328
5.50 source/Production/Renewable/Wind.cpp File Reference	328
5.50.1 Detailed Description	328
5.51 source/Resources.cpp File Reference	329
5.51.1 Detailed Description	329
5.52 source/Storage/Lilon.cpp File Reference	329
5.52.1 Detailed Description	330
5.53 source/Storage/Storage.cpp File Reference	330

5.53.1 Detailed Description	330
5.54 test/source/Production/Combustion/test_Combustion.cpp File Reference	330
5.54.1 Detailed Description	331
5.54.2 Function Documentation	331
5.54.2.1 main()	331
5.55 test/source/Production/Combustion/test_Diesel.cpp File Reference	332
5.55.1 Detailed Description	333
5.55.2 Function Documentation	333
5.55.2.1 main()	333
5.56 test/source/Production/Noncombustion/test_Hydro.cpp File Reference	339
5.56.1 Detailed Description	339
5.56.2 Function Documentation	339
5.56.2.1 main()	340
5.57 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference	342
5.57.1 Detailed Description	342
5.57.2 Function Documentation	342
5.57.2.1 main()	343
5.58 test/source/Production/Renewable/test_Renewable.cpp File Reference	343
5.58.1 Detailed Description	344
5.58.2 Function Documentation	344
5.58.2.1 main()	344
5.59 test/source/Production/Renewable/test_Solar.cpp File Reference	345
5.59.1 Detailed Description	345
5.59.2 Function Documentation	346
5.59.2.1 main()	346
5.60 test/source/Production/Renewable/test_Tidal.cpp File Reference	349
5.60.1 Detailed Description	349
5.60.2 Function Documentation	349
5.60.2.1 main()	350
5.61 test/source/Production/Renewable/test_Wave.cpp File Reference	352
5.61.1 Detailed Description	353
5.61.2 Function Documentation	353
5.61.2.1 main()	353
5.62 test/source/Production/Renewable/test_Wind.cpp File Reference	357
5.62.1 Detailed Description	358
5.62.2 Function Documentation	358
5.62.2.1 main()	358
5.63 test/source/Production/test_Production.cpp File Reference	361
5.63.1 Detailed Description	362
5.63.2 Function Documentation	362
5.63.2.1 main()	362
5.64 test/source/Storage/test_Lilon.cpp File Reference	364

5.64.1 Detailed Description	364
5.64.2 Function Documentation	364
5.64.2.1 main()	365
5.65 test/source/Storage/test_Storage.cpp File Reference	367
5.65.1 Detailed Description	367
5.65.2 Function Documentation	367
5.65.2.1 main()	368
5.66 test/source/test_Controller.cpp File Reference	369
5.66.1 Detailed Description	370
5.66.2 Function Documentation	370
5.66.2.1 main()	370
5.67 test/source/test_ElectricalLoad.cpp File Reference	371
5.67.1 Detailed Description	371
5.67.2 Function Documentation	371
5.67.2.1 main()	371
5.68 test/source/test_Interpolator.cpp File Reference	373
5.68.1 Detailed Description	374
5.68.2 Function Documentation	374
5.68.2.1 main()	374
5.69 test/source/test_Model.cpp File Reference	379
5.69.1 Detailed Description	380
5.69.2 Function Documentation	380
5.69.2.1 main()	380
5.70 test/source/test_Resources.cpp File Reference	388
5.70.1 Detailed Description	389
5.70.2 Function Documentation	389
5.70.2.1 main()	389
5.71 test/utlis/testing_utils.cpp File Reference	396
5.71.1 Detailed Description	397
5.71.2 Function Documentation	397
5.71.2.1 expectedErrorNotDetected()	397
5.71.2.2 printGold()	397
5.71.2.3 printGreen()	398
5.71.2.4 printRed()	398
5.71.2.5 testFloatEquals()	398
5.71.2.6 testGreaterThan()	399
5.71.2.7 testGreaterThanOrEqualTo()	399
5.71.2.8 testLessThan()	400
5.71.2.9 testLessThanOrEqualTo()	401
5.71.2.10 testTruth()	401
5.72 test/utlis/testing_utils.h File Reference	402
5.72.1 Detailed Description	403

5.72.2 Macro Definition Documentation	403
5.72.2.1 FLOAT_TOLERANCE	403
5.72.3 Function Documentation	403
5.72.3.1 expectedErrorNotDetected()	403
5.72.3.2 printGold()	404
5.72.3.3 printGreen()	404
5.72.3.4 printRed()	404
5.72.3.5 testFloatEquals()	405
5.72.3.6 testGreaterThanOr()	405
5.72.3.7 testGreaterThanOrEqualTo()	406
5.72.3.8 testLessThan()	407
5.72.3.9 testLessThanOrEqualTo()	407
5.72.3.10 testTruth()	408
 Bibliography	 410
 Index	 411

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

CombustionInputs	24
Controller	25
DieselInputs	58
ElectricalLoad	62
Emissions	67
HydroInputs	85
Interpolator	88
InterpolatorStruct1D	101
InterpolatorStruct2D	103
LilonInputs	124
Model	129
ModelInputs	148
NoncombustionInputs	157
Production	158
Combustion	9
Diesel	45
Noncombustion	149
Hydro	69
Renewable	172
Solar	195
Tidal	221
Wave	235
Wind	251
ProductionInputs	170
RenewableInputs	180
Resources	181
SolarInputs	203
Storage	205
Lilon	105
StorageInputs	219
TidalInputs	232
WaveInputs	248
WindInputs	263

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	25
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 . . .	58
ElectricalLoad	A class which contains time and electrical load data. Intended to serve as a component class of Model	62
Emissions	A structure which bundles the emitted masses of various emissions chemistries	67
Hydro	A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not)	69
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	85
Interpolator	A class which contains interpolation data and functionality. Intended to serve as a component of the Production and Storage hierarchies	88
InterpolatorStruct1D	A struct which holds two parallel vectors for use in 1D interpolation	101
InterpolatorStruct2D	A struct which holds two parallel vectors and a matrix for use in 2D interpolation	103
Lilon	A derived class of Storage which models energy storage by way of lithium-ion batteries	105

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	124
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	129
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)	148
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	149
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	157
Production	The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise	158
ProductionInputs	A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input	170
Renewable	The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy	172
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	180
Resources	A class which contains renewable resource data. Intended to serve as a component class of Model	181
Solar	A derived class of the Renewable branch of Production which models solar production	195
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	203
Storage	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	205
StorageInputs	A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input	219
Tidal	A derived class of the Renewable branch of Production which models tidal production	221
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	232
Wave	A derived class of the Renewable branch of Production which models wave production	235
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	248
Wind	A derived class of the Renewable branch of Production which models wind production	251
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	263

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	265
header/ doxygen_cite.h	
Header file which simply cites the doxygen tool	266
header/ ElectricalLoad.h	
Header file for the ElectricalLoad class	267
header/ Interpolator.h	
Header file for the Interpolator class	267
header/ Model.h	
Header file for the Model class	268
header/ Resources.h	
Header file for the Resources class	283
header/ std_includes.h	
Header file which simply batches together some standard includes	284
header/Production/ Production.h	
Header file for the Production class	275
header/Production/Combustion/ Combustion.h	
Header file for the Combustion class	269
header/Production/Combustion/ Diesel.h	
Header file for the Diesel class	272
header/Production/Noncombustion/ Hydro.h	
Header file for the Hydro class	273
header/Production/Noncombustion/ Noncombustion.h	
Header file for the Noncombustion class	274
header/Production/Renewable/ Renewable.h	
Header file for the Renewable class	276
header/Production/Renewable/ Solar.h	
Header file for the Solar class	278
header/Production/Renewable/ Tidal.h	
Header file for the Tidal class	279
header/Production/Renewable/ Wave.h	
Header file for the Wave class	280
header/Production/Renewable/ Wind.h	
Header file for the Wind class	282
header/Storage/ Lilon.h	
Header file for the Lilon class	285

header/Storage/Storage.h	
Header file for the Storage class	286
projects/example.cpp	287
pybindings/PYBIND11_PGM.cpp	
Bindings file for PGMcpp	289
pybindings/snippets/PYBIND11_Controller.cpp	
Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp	306
pybindings/snippets/PYBIND11_ElectricalLoad.cpp	
Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp	308
pybindings/snippets/PYBIND11_Interpolator.cpp	
Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp	309
pybindings/snippets/PYBIND11_Model.cpp	
Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp	312
pybindings/snippets/PYBIND11_Resources.cpp	
Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp	313
pybindings/snippets/Production/PYBIND11_Production.cpp	
Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp	295
pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp	
Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp	291
pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp	
Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp	292
pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp	
Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp	297
pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp	
Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp	298
pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp	
Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp	300
pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp	
Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp	302
pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp	
Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp	304
pybindings/snippets/Storage/PYBIND11_Lilon.cpp	
Bindings file for the Lilon class. Intended to be #include'd in PYBIND11_PGM.cpp	314
pybindings/snippets/Storage/PYBIND11_Storage.cpp	
Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp	319
source/Controller.cpp	
Implementation file for the Controller class	321
source/ElectricalLoad.cpp	
Implementation file for the ElectricalLoad class	321
source/Interpolator.cpp	
Implementation file for the Interpolator class	322
source/Model.cpp	
Implementation file for the Model class	322
source/Resources.cpp	
Implementation file for the Resources class	329
source/Production/Production.cpp	
Implementation file for the Production class	325
source/Production/Combustion/Combustion.cpp	
Implementation file for the Combustion class	323
source/Production/Combustion/Diesel.cpp	
Implementation file for the Diesel class	323
source/Production/Noncombustion/Hydro.cpp	
Implementation file for the Hydro class	324
source/Production/Noncombustion/Noncombustion.cpp	
Implementation file for the Noncombustion class	324
source/Production/Renewable/Renewable.cpp	
Implementation file for the Renewable class	326

source/Production/Renewable/ Solar.cpp	
Implementation file for the Solar class	326
source/Production/Renewable/ Tidal.cpp	
Implementation file for the Tidal class	327
source/Production/Renewable/ Wave.cpp	
Implementation file for the Wave class	327
source/Production/Renewable/ Wind.cpp	
Implementation file for the Wind class	328
source/Storage/ Lilon.cpp	
Implementation file for the Lilon class	329
source/Storage/ Storage.cpp	
Implementation file for the Storage class	330
test/source/ test_Controller.cpp	
Testing suite for Controller class	369
test/source/ test_ElectricalLoad.cpp	
Testing suite for ElectricalLoad class	371
test/source/ test_Interpolator.cpp	
Testing suite for Interpolator class	373
test/source/ test_Model.cpp	
Testing suite for Model class	379
test/source/ test_Resources.cpp	
Testing suite for Resources class	388
test/source/Production/ test_Production.cpp	
Testing suite for Production class	361
test/source/Production/Combustion/ test_Combustion.cpp	
Testing suite for Combustion class	330
test/source/Production/Combustion/ test_Diesel.cpp	
Testing suite for Diesel class	332
test/source/Production/Noncombustion/ test_Hydro.cpp	
Testing suite for Hydro class	339
test/source/Production/Noncombustion/ test_Noncombustion.cpp	
Testing suite for Noncombustion class	342
test/source/Production/Renewable/ test_Renewable.cpp	
Testing suite for Renewable class	343
test/source/Production/Renewable/ test_Solar.cpp	
Testing suite for Solar class	345
test/source/Production/Renewable/ test_Tidal.cpp	
Testing suite for Tidal class	349
test/source/Production/Renewable/ test_Wave.cpp	
Testing suite for Wave class	352
test/source/Production/Renewable/ test_Wind.cpp	
Testing suite for Wind class	357
test/source/Storage/ test_Lilon.cpp	
Testing suite for Lilon class	364
test/source/Storage/ test_Storage.cpp	
Testing suite for Storage class	367
test/utls/ testing_utils.cpp	
Header file for various PGMcpp testing utilities	396
test/utls/ testing_utils.h	
Header file for various PGMcpp testing utilities	402

Chapter 4

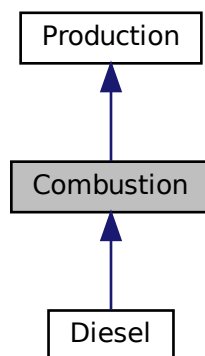
Class Documentation

4.1 Combustion Class Reference

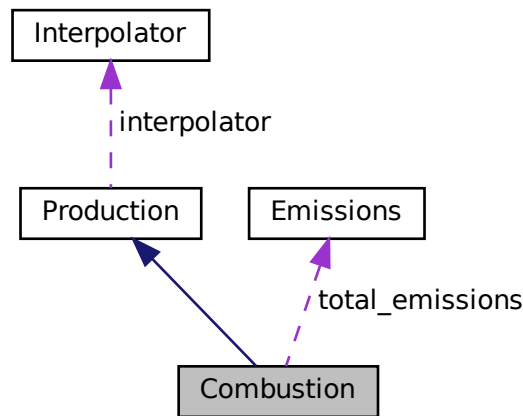
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

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

Public Attributes

- [CombustionType](#) type
The type (CombustionType) of the asset.
- [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 [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.

```
77 {
78     return;
79 } /* Combustion() */
```

4.1.2.2 `Combustion()` [2/2]

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

Constructor (intended) for the `Combustion` class.

Parameters

<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 <code>Combustion</code> constructor inputs.

```
107 :
108 Production(
109     n_points,
110     n_years,
111     combustion_inputs.production_inputs
112 )
113 {
114     // 1. check inputs
115     this->__checkInputs(combustion_inputs);
116 }
```

```

117 // 2. set attributes
118 this->fuel_mode = combustion_inputs.fuel_mode;
119
120 switch (this->fuel_mode) {
121     case (FuelMode :: FUEL_MODE_LINEAR): {
122         this->fuel_mode_str = "FUEL_MODE_LINEAR";
123
124         break;
125     }
126
127     case (FuelMode :: FUEL_MODE_LOOKUP): {
128         this->fuel_mode_str = "FUEL_MODE_LOOKUP";
129
130         this->interpolator.addData1D(
131             0,
132             combustion_inputs.path_2_fuel_interp_data
133         );
134
135         break;
136     }
137
138     default: {
139         std::string error_str = "ERROR: Combustion(): ";
140         error_str += "fuel mode ";
141         error_str += std::to_string(this->fuel_mode);
142         error_str += " not recognized";
143
144         #ifdef _WIN32
145             std::cout << error_str << std::endl;
146         #endif
147
148         throw std::runtime_error(error_str);
149
150         break;
151     }
152 }
153
154 this->fuel_cost_L = 0;
155 this->nominal_fuel_escalation_annual =
156     combustion_inputs.nominal_fuel_escalation_annual;
157
158 this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
159     combustion_inputs.nominal_fuel_escalation_annual,
160     combustion_inputs.production_inputs.nominal_discount_annual
161 );
162
163 this->linear_fuel_slope_LkWh = 0;
164 this->linear_fuel_intercept_LkWh = 0;
165
166 this->CO2_emissions_intensity_kgL = 0;
167 this->CO_emissions_intensity_kgL = 0;
168 this->NOx_emissions_intensity_kgL = 0;
169 this->SOx_emissions_intensity_kgL = 0;
170 this->CH4_emissions_intensity_kgL = 0;
171 this->PM_emissions_intensity_kgL = 0;
172
173 this->total_fuel_consumed_L = 0;
174
175 this->fuel_consumption_vec_L.resize(this->n_points, 0);
176 this->fuel_cost_vec.resize(this->n_points, 0);
177
178 this->CO2_emissions_vec_kg.resize(this->n_points, 0);
179 this->CO_emissions_vec_kg.resize(this->n_points, 0);
180 this->NOx_emissions_vec_kg.resize(this->n_points, 0);
181 this->SOx_emissions_vec_kg.resize(this->n_points, 0);
182 this->CH4_emissions_vec_kg.resize(this->n_points, 0);
183 this->PM_emissions_vec_kg.resize(this->n_points, 0);
184
185 // 3. construction print
186 if (this->print_flag) {
187     std::cout << "Combustion object constructed at " << this << std::endl;
188 }
189
190 return;
191 } /* Combustion() */

```

4.1.2.3 ~Combustion()

```

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

```

Destructor for the [Combustion](#) class.

```

529 {
530     // 1. destruction print
531     if (this->print_flag) {
532         std::cout << "Combustion object at " << this << " destroyed" << std::endl;
533     }
534     return;
535 }
536 } /* ~Combustion() */

```

4.1.3 Member Function Documentation

4.1.3.1 __checkInputs()

```

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

```

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

Parameters

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

```

40 {
41     // 1. if FUEL_MODE_LOOKUP, check that path is given
42     if (
43         combustion_inputs.fuel_mode == FuelMode :: FUEL_MODE_LOOKUP and
44         combustion_inputs.path_2_fuel_interp_data.empty()
45     ) {
46         std::string error_str = "ERROR: Combustion() fuel mode was set to ";
47         error_str += "FuelMode::FUEL_MODE_LOOKUP, but no path to fuel interpolation ";
48         error_str += "data was given";
49
50         #ifdef _WIN32
51             std::cout << error_str << std::endl;
52         #endif
53
54         throw std::invalid_argument(error_str);
55     }
56
57     return;
58 } /* __checkInputs() */

```

4.1.3.2 __writeSummary()

```

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

```

Reimplemented in [Diesel](#).

```

105 {return;}

```


4.1.3.3 __writeTimeSeries()

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

Reimplemented in [Diesel](#).

```
110         {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](#).

```
321 {
322     // 1. invoke base class method
323     load_kW = Production::commit(
324         timestep,
325         dt_hrs,
326         production_kW,
327         load_kW
328     );
329
330
331     if (this->is_running) {
332         // 2. compute and record fuel consumption
333         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
334         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
335
336         // 3. compute and record emissions
337         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
338         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
339         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
340         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
341         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
342         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
343         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
344
345         // 4. incur fuel costs

```

```

346         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
347     }
348
349     return load_kW;
350 } /* 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 time_vec_hrs attribute of the ElectricalLoad .
-------------------------	---

Reimplemented from [Production](#).

```

265 {
266     // 1. account for fuel costs in net present cost
267     double t_hrs = 0;
268     double real_fuel_escalation_scalar = 0;
269
270     for (int i = 0; i < this->n_points; i++) {
271         t_hrs = time_vec_hrs_ptr->at(i);
272
273         real_fuel_escalation_scalar = 1.0 / pow(
274             1 + this->real_fuel_escalation_annual,
275             t_hrs / 8760
276         );
277
278         this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
279     }
280
281     // 2. invoke base class method
282     Production :: computeEconomics(time_vec_hrs_ptr);
283
284     return;
285 } /* computeEconomics() */

```

4.1.3.6 computeFuelAndEmissions()

```

void Combustion::computeFuelAndEmissions (
    void )

```

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

```

233 {
234     for (int i = 0; i < n_points; i++) {
235         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
236
237         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
238         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
239         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
240         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
241         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
242         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
243     }
244
245     return;
246 } /* 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.

```
429                                     {
430     Emissions emissions;
431
432     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
433     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
434     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
435     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
436     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
437     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
438
439     return emissions;
440 } /* 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].

```
372 {
373     double fuel_consumed_L = 0;
374
375     switch (this->fuel_mode) {
376     case (FuelMode :: FUEL_MODE_LINEAR): {
377         fuel_consumed_L = (
378             this->linear_fuel_slope_LkWh * production_kW +
379             this->linear_fuel_intercept_LkWh * this->capacity_kW
380         ) * dt_hrs;
381
382         break;
383     }
384
385     case (FuelMode :: FUEL_MODE_LOOKUP): {
```

```

386         double load_ratio = production_kW / this->capacity_kW;
387
388         fuel_consumed_L = this->interpolator.interp1D(0, load_ratio) * dt_hrs;
389
390         break;
391     }
392
393     default: {
394         std::string error_str = "ERROR: Combustion::getFuelConsumptionL(): ";
395         error_str += "fuel mode ";
396         error_str += std::to_string(this->fuel_mode);
397         error_str += " not recognized";
398
399         #ifdef _WIN32
400             std::cout << error_str << std::endl;
401         #endif
402
403         throw std::runtime_error(error_str);
404
405         break;
406     }
407 }
408
409 return fuel_consumed_L;
410 } /* 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](#).

```

209 {
210     // 1. reset attributes
211     //...
212
213     // 2. invoke base class method
214     Production::handleReplacement(timestep);
215
216     return;
217 } /* __handleReplacement() */

```

4.1.3.10 requestProductionkW()

```

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

```

Reimplemented in [Diesel](#).

```

156 {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.

```
476 {
477     // 1. handle sentinel
478     if (max_lines < 0) {
479         max_lines = this->n_points;
480     }
481
482     // 2. create subdirectories
483     write_path += "Production/";
484     if (not std::filesystem::is_directory(write_path)) {
485         std::filesystem::create_directory(write_path);
486     }
487
488     write_path += "Combustion/";
489     if (not std::filesystem::is_directory(write_path)) {
490         std::filesystem::create_directory(write_path);
491     }
492
493     write_path += this->type_str;
494     write_path += "_";
495     write_path += std::to_string(int(ceil(this->capacity_kW)));
496     write_path += "kW_idx";
497     write_path += std::to_string(combustion_index);
498     write_path += "/";
499     std::filesystem::create_directory(write_path);
500
501     // 3. write summary
502     this->__writeSummary(write_path);
503
504     // 4. write time series
505     if (max_lines > this->n_points) {
506         max_lines = this->n_points;
507     }
508
509     if (max_lines > 0) {
510         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
511     }
512
513     return;
514 } /* writeResults() */
```

4.1.4 Member Data Documentation

4.1.4.1 CH4_emissions_intensity_kgL

```
double Combustion::CH4_emissions_intensity_kgL
```

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

4.1.4.2 CH4_emissions_vec_kg

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

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

4.1.4.3 CO2_emissions_intensity_kgL

```
double Combustion::CO2_emissions_intensity_kgL
```

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

4.1.4.4 CO2_emissions_vec_kg

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

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

4.1.4.5 CO_emissions_intensity_kgL

```
double Combustion::CO_emissions_intensity_kgL
```

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

4.1.4.6 CO_emissions_vec_kg

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

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

4.1.4.7 fuel_consumption_vec_L

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

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

4.1.4.8 fuel_cost_L

```
double Combustion::fuel_cost_L
```

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

4.1.4.9 fuel_cost_vec

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

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

4.1.4.10 fuel_mode

```
FuelMode Combustion::fuel_mode
```

The fuel mode to use in modelling fuel consumption.

4.1.4.11 fuel_mode_str

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

A string describing the fuel mode of the asset.

4.1.4.12 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.

4.1.4.13 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.

4.1.4.14 nominal_fuel_escalation_annual

```
double Combustion::nominal_fuel_escalation_annual
```

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

4.1.4.15 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.16 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.

4.1.4.17 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.18 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.

4.1.4.19 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.

4.1.4.20 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.21 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.

4.1.4.22 total_emissions

```
Emissions Combustion::total_emissions
```

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

4.1.4.23 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

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

4.1.4.24 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

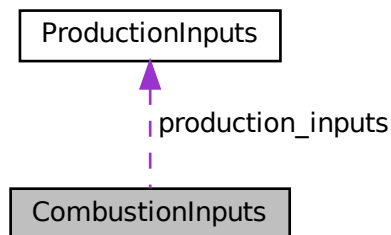
- [header/Production/Combustion/Combustion.h](#)
- [source/Production/Combustion/Combustion.cpp](#)

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.
- 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 fuel_mode

```
FuelMode CombustionInputs::fuel_mode = FuelMode :: FUEL\_MODE\_LINEAR
```

The fuel mode to use in modelling fuel consumption.

4.2.2.2 nominal_fuel_escalation_annual

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

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

4.2.2.3 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.

4.2.2.4 production_inputs

```
ProductionInputs CombustionInputs::production_inputs
```

An encapsulated [ProductionInputs](#) instance.

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

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

4.3 Controller Class Reference

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

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

Public Member Functions

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

Public Attributes

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

Private Member Functions

- `void` [__computeNetLoad](#) (*ElectricalLoad* *, `std::vector< Renewable * >` *, *Resources* *)
Helper method to compute and populate the net load vector.
- `void` [__constructCombustionMap](#) (`std::vector< Combustion * >` *)
Helper method to construct a *Combustion* map, for use in determining.
- `void` [__applyLoadFollowingControl_CHARGING](#) (int, *ElectricalLoad* *, *Resources* *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply load following control action for given timestep of the *Model* run when net load ≤ 0 .
- `void` [__applyLoadFollowingControl_DISCHARGING](#) (int, *ElectricalLoad* *, *Resources* *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply load following control action for given timestep of the *Model* run when net load > 0 .
- `void` [__applyCycleChargingControl_CHARGING](#) (int, *ElectricalLoad* *, *Resources* *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply cycle charging control action for given timestep of the *Model* run when net load ≤ 0 . Simply defaults to load following control.
- `void` [__applyCycleChargingControl_DISCHARGING](#) (int, *ElectricalLoad* *, *Resources* *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *, `std::vector< Storage * >` *)
Helper method to apply cycle charging control action for given timestep of the *Model* run when net load > 0 . Defaults to load following control if no depleted storage assets.
- `void` [__handleStorageCharging](#) (int, double, `std::list< Storage * >` *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *)
Helper method to handle the charging of the given *Storage* assets.
- `void` [__handleStorageCharging](#) (int, double, `std::vector< Storage * >` *, `std::vector< Combustion * >` *, `std::vector< Noncombustion * >` *, `std::vector< Renewable * >` *)
Helper method to handle the charging of the given *Storage* assets.
- `double` [__getRenewableProduction](#) (int, double, *Renewable* *, *Resources* *)
Helper method to compute the production from the given *Renewable* asset at the given point in time.
- `double` [__handleCombustionDispatch](#) (int, double, double, `std::vector< Combustion * >` *, bool)
bool is_cycle_charging)
- `double` [__handleNoncombustionDispatch](#) (int, double, double, `std::vector< Noncombustion * >` *, *Resources* *)
- `double` [__handleStorageDischarging](#) (int, double, double, `std::list< Storage * >` *)
Helper method to handle the discharging of the given *Storage* assets.

4.3.1 Detailed Description

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

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Controller()

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

Constructor for the [Controller](#) class.

```
1209 {
1210     return;
1211 } /* Controller() */
```

4.3.2.2 ~Controller()

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

Destructor for the [Controller](#) class.

```
1455 {
1456     this->clear();
1457
1458     return;
1459 } /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

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

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

Parameters

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

```

450 {
451     // 1. default to load following
452     this->__applyLoadFollowingControl_CHARGING(
453         timestep,
454         electrical_load_ptr,
455         resources_ptr,
456         combustion_ptr_vec_ptr,
457         noncombustion_ptr_vec_ptr,
458         renewable_ptr_vec_ptr,
459         storage_ptr_vec_ptr
460     );
461
462     return;
463 } /* __applyCycleChargingControl_CHARGING() */

```

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

```

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

```

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

Parameters

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

curtailment

```

511 {
512     // 1. get dt_hrs, net load
513     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
514     double net_load_kW = this->net_load_vec_kW[timestep];
515
516     // 2. partition Storage assets into depleted and non-depleted
517     std::list<Storage*> depleted_storage_ptr_list;

```

```

518     std::list<Storage*> nondepleted_storage_ptr_list;
519
520     Storage* storage_ptr;
521     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
522         storage_ptr = storage_ptr_vec_ptr->at(i);
523
524         if (storage_ptr->is_depleted) {
525             depleted_storage_ptr_list.push_back(storage_ptr);
526         }
527
528         else {
529             nondepleted_storage_ptr_list.push_back(storage_ptr);
530         }
531     }
532
533     // 3. discharge non-depleted storage assets
534     net_load_kW = this->__handleStorageDischarging(
535         timestep,
536         dt_hrs,
537         net_load_kW,
538         nondepleted_storage_ptr_list
539     );
540
541     // 4. request optimal production from all Noncombustion assets
542     net_load_kW = this->__handleNoncombustionDispatch(
543         timestep,
544         dt_hrs,
545         net_load_kW,
546         noncombustion_ptr_vec_ptr,
547         resources_ptr
548     );
549
550     // 5. request optimal production from all Combustion assets
551     // default to load following if no depleted storage
552     if (depleted_storage_ptr_list.empty()) {
553         net_load_kW = this->__handleCombustionDispatch(
554             timestep,
555             dt_hrs,
556             net_load_kW,
557             combustion_ptr_vec_ptr,
558             false // is_cycle_charging
559         );
560     }
561
562     else {
563         net_load_kW = this->__handleCombustionDispatch(
564             timestep,
565             dt_hrs,
566             net_load_kW,
567             combustion_ptr_vec_ptr,
568             true // is_cycle_charging
569         );
570     }
571
572     // 6. attempt to charge depleted Storage assets using any and all available
573     // charge priority is Combustion, then Renewable
574     this->__handleStorageCharging(
575         timestep,
576         dt_hrs,
577         depleted_storage_ptr_list,
578         combustion_ptr_vec_ptr,
579         noncombustion_ptr_vec_ptr,
580         renewable_ptr_vec_ptr
581     );
582
583
584     // 7. record any missed load
585     if (net_load_kW > 1e-6) {
586         this->missed_load_vec_kW[timestep] = net_load_kW;
587     }
588
589     return;
590 } /* __applyCycleChargingControl_DISCHARGING() */

```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```

void Controller::__applyLoadFollowingControl_CHARGING (
    int timestep,
    ElectricalLoad * electrical_load_ptr,

```

```

Resources * resources_ptr,
std::vector< Combustion * > * combustion_ptr_vec_ptr,
std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
std::vector< Renewable * > * renewable_ptr_vec_ptr,
std::vector< Storage * > * storage_ptr_vec_ptr ) [private]

```

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

Parameters

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

```

255 {
256     // 1. get dt_hrs, set net load
257     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
258     double net_load_kW = 0;
259
260     // 2. request zero production from all Combustion assets
261     this->__handleCombustionDispatch(
262         timestep,
263         dt_hrs,
264         net_load_kW,
265         combustion_ptr_vec_ptr,
266         false // is_cycle_charging
267     );
268
269     // 3. request zero production from all Noncombustion assets
270     this->__handleNoncombustionDispatch(
271         timestep,
272         dt_hrs,
273         net_load_kW,
274         noncombustion_ptr_vec_ptr,
275         resources_ptr
276     );
277
278     // 4. attempt to charge all Storage assets using any and all available curtailment
279     // charge priority is Combustion, then Renewable
280     this->__handleStorageCharging(
281         timestep,
282         dt_hrs,
283         storage_ptr_vec_ptr,
284         combustion_ptr_vec_ptr,
285         noncombustion_ptr_vec_ptr,
286         renewable_ptr_vec_ptr
287     );
288
289     return;
290 } /* __applyLoadFollowingControl_CHARGING() */

```

4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

```

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

```

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

Parameters

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

curtailment

```

337 {
338     // 1. get dt_hrs, net load
339     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
340     double net_load_kW = this->net_load_vec_kW[timestep];
341
342     // 2. partition Storage assets into depleted and non-depleted
343     std::list<Storage*> depleted_storage_ptr_list;
344     std::list<Storage*> nondepleted_storage_ptr_list;
345
346     Storage* storage_ptr;
347     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
348         storage_ptr = storage_ptr_vec_ptr->at(i);
349
350         if (storage_ptr->is_depleted) {
351             depleted_storage_ptr_list.push_back(storage_ptr);
352         }
353
354         else {
355             nondepleted_storage_ptr_list.push_back(storage_ptr);
356         }
357     }
358
359     // 3. discharge non-depleted storage assets
360     net_load_kW = this->__handleStorageDischarging(
361         timestep,
362         dt_hrs,
363         net_load_kW,
364         nondepleted_storage_ptr_list
365     );
366
367     // 4. request optimal production from all Noncombustion assets
368     net_load_kW = this->__handleNoncombustionDispatch(
369         timestep,
370         dt_hrs,
371         net_load_kW,
372         noncombustion_ptr_vec_ptr,
373         resources_ptr
374     );
375
376     // 5. request optimal production from all Combustion assets
377     net_load_kW = this->__handleCombustionDispatch(
378         timestep,
379         dt_hrs,
380         net_load_kW,
381         combustion_ptr_vec_ptr,
382         false // is_cycle_charging
383     );
384
385     // 6. attempt to charge depleted Storage assets using any and all available
386     // charge priority is Combustion, then Renewable
387     this->__handleStorageCharging(
388         timestep,
389         dt_hrs,
390         depleted_storage_ptr_list,
391         combustion_ptr_vec_ptr,
392         noncombustion_ptr_vec_ptr,
393         renewable_ptr_vec_ptr
394     );
395
396     // 7. record any missed load
397     if (net_load_kW > 1e-6) {
398         this->missed_load_vec_kW[timestep] = net_load_kW;
399     }
400 }
401
402 return;
403 } /* __applyLoadFollowingControl_DISCHARGING() */

```

4.3.3.5 __computeNetLoad()

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

Helper method to compute and populate the net load vector.

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

Parameters

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

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

4.3.3.6 __constructCombustionMap()

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

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

Parameters

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

```

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

```

```

203     }
204     // ==== END TEST PRINT ==== //
205     /**/
206
207     return;
208 } /* __constructCombustionTable() */

```

4.3.3.7 __getRenewableProduction()

```

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

```

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

Parameters

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

Returns

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

```

879 {
880     double production_kW = 0;
881
882     switch (renewable_ptr->type) {
883         case (RenewableType :: SOLAR): {
884             production_kW = renewable_ptr->computeProductionkW(
885                 timestep,
886                 dt_hrs,
887                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
888             );
889
890             break;
891         }
892
893         case (RenewableType :: TIDAL): {
894             production_kW = renewable_ptr->computeProductionkW(
895                 timestep,
896                 dt_hrs,
897                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
898             );
899
900             break;
901         }
902
903         case (RenewableType :: WAVE): {
904             production_kW = renewable_ptr->computeProductionkW(
905                 timestep,
906                 dt_hrs,
907                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0],
908                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1]
909             );
910
911             break;
912         }
913
914         case (RenewableType :: WIND): {
915             production_kW = renewable_ptr->computeProductionkW(
916                 timestep,
917                 dt_hrs,
918                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]

```

```

919         );
920
921         break;
922     }
923
924     default: {
925         std::string error_str = "ERROR: Controller::__getRenewableProduction(): ";
926         error_str += "renewable type ";
927         error_str += std::to_string(renewable_ptr->type);
928         error_str += " not recognized";
929
930         #ifdef _WIN32
931             std::cout << error_str << std::endl;
932         #endif
933
934         throw std::runtime_error(error_str);
935
936         break;
937     }
938 }
939
940 return production_kW;
941 } /* __getRenewableProduction() */

```

4.3.3.8 __handleCombustionDispatch()

```

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

```

bool is_cycle_charging)

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

Parameters

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

Returns

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

```

984 {
985     // 1. get minimal Combustion dispatch
986     double target_production_kW = 1.2 * net_load_kW;
987     double total_capacity_kW = 0;
988
989     std::map<double, std::vector<bool>::iterator> iter = this->combustion_map.begin();
990     while (iter != std::prev(this->combustion_map.end(), 1)) {
991         if (target_production_kW <= total_capacity_kW) {
992             break;
993         }
994
995         iter++;
996         total_capacity_kW = iter->first;

```

```

997     }
998
999     // 2. share load proportionally (by rated capacity) over active diesels
1000     Combustion* combustion_ptr;
1001     double production_kW = 0;
1002     double request_kW = 0;
1003     double _net_load_kW = net_load_kW;
1004
1005     for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
1006         combustion_ptr = combustion_ptr_vec_ptr->at(i);
1007
1008         if (total_capacity_kW > 0) {
1009             request_kW =
1010                 int(this->combustion_map[total_capacity_kW][i]) *
1011                 net_load_kW *
1012                 (combustion_ptr->capacity_kW / total_capacity_kW);
1013         }
1014
1015         else {
1016             request_kW = 0;
1017         }
1018
1019         if (is_cycle_charging and request_kW > 0) {
1020             if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
1021                 request_kW = 0.85 * combustion_ptr->capacity_kW;
1022             }
1023         }
1024
1025         production_kW = combustion_ptr->requestProductionkW(
1026             timestep,
1027             dt_hrs,
1028             request_kW
1029         );
1030
1031         _net_load_kW = combustion_ptr->commit(
1032             timestep,
1033             dt_hrs,
1034             production_kW,
1035             _net_load_kW
1036         );
1037     }
1038
1039     return _net_load_kW;
1040 } /* __handleCombustionDispatch() */

```

4.3.3.9 __handleNoncombustionDispatch()

```

double Controller::__handleNoncombustionDispatch (
    int timestep,
    double dt_hrs,
    double net_load_kW,
    std::vector< Noncombustion * > * noncombustion_ptr_vec_ptr,
    Resources * resources_ptr ) [private]
1081 {
1082     Noncombustion* noncombustion_ptr;
1083     double production_kW = 0;
1084
1085     for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
1086         noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
1087
1088         switch (noncombustion_ptr->type) {
1089             case (NoncombustionType :: HYDRO): {
1090                 production_kW = noncombustion_ptr->requestProductionkW(
1091                     timestep,
1092                     dt_hrs,
1093                     net_load_kW,
1094                     resources_ptr->resource_map_1D[noncombustion_ptr->resource_key][timestep]
1095                 );
1096
1097                 net_load_kW = noncombustion_ptr->commit(
1098                     timestep,
1099                     dt_hrs,
1100                     production_kW,
1101                     net_load_kW,
1102                     resources_ptr->resource_map_1D[noncombustion_ptr->resource_key][timestep]
1103                 );
1104

```

```

1105         break;
1106     }
1107
1108     default: {
1109         production_kW = noncombustion_ptr->requestProductionkW(
1110             timestep,
1111             dt_hrs,
1112             net_load_kW
1113         );
1114
1115         net_load_kW = noncombustion_ptr->commit(
1116             timestep,
1117             dt_hrs,
1118             production_kW,
1119             net_load_kW
1120         );
1121
1122         break;
1123     }
1124 }
1125 }
1126
1127 return net_load_kW;
1128 } /* __handleNoncombustionDispatch() */

```

4.3.3.10 __handleStorageCharging() [1/2]

```

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

```

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

Parameters

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

```

633 {
634     double acceptable_kW = 0;
635     double curtailment_kW = 0;
636
637     Storage* storage_ptr;
638     Combustion* combustion_ptr;
639     Noncombustion* noncombustion_ptr;
640     Renewable* renewable_ptr;
641
642     std::list<Storage*>::iterator iter;
643     for (
644         iter = storage_ptr_list.begin();
645         iter != storage_ptr_list.end();
646         iter++
647     ){
648         storage_ptr = (*iter);
649
650         // 1. attempt to charge from Combustion curtailment first
651         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
652             combustion_ptr = combustion_ptr_vec_ptr->at(i);
653             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
654

```

```

655         if (curtailment_kW <= 0) {
656             continue;
657         }
658
659         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
660
661         if (acceptable_kW > curtailment_kW) {
662             acceptable_kW = curtailment_kW;
663         }
664
665         combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
666         combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
667         storage_ptr->power_kW += acceptable_kW;
668     }
669
670     // 2. attempt to charge from Noncombustion curtailment second
671     for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
672         noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
673         curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
674
675         if (curtailment_kW <= 0) {
676             continue;
677         }
678
679         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
680
681         if (acceptable_kW > curtailment_kW) {
682             acceptable_kW = curtailment_kW;
683         }
684
685         noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
686         noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
687         storage_ptr->power_kW += acceptable_kW;
688     }
689
690     // 3. attempt to charge from Renewable curtailment third
691     for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
692         renewable_ptr = renewable_ptr_vec_ptr->at(i);
693         curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
694
695         if (curtailment_kW <= 0) {
696             continue;
697         }
698
699         acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
700
701         if (acceptable_kW > curtailment_kW) {
702             acceptable_kW = curtailment_kW;
703         }
704
705         renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
706         renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
707         storage_ptr->power_kW += acceptable_kW;
708     }
709
710     // 4. commit charge
711     storage_ptr->commitCharge(
712         timestep,
713         dt_hrs,
714         storage_ptr->power_kW
715     );
716 }
717
718 return;
719 } /* __handleStorageCharging() */

```

4.3.3.11 __handleStorageCharging() [2/2]

```

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

```

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

Parameters

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

```

762 {
763     double acceptable_kW = 0;
764     double curtailment_kW = 0;
765
766     Storage* storage_ptr;
767     Combustion* combustion_ptr;
768     Noncombustion* noncombustion_ptr;
769     Renewable* renewable_ptr;
770
771     for (size_t j = 0; j < storage_ptr_vec_ptr->size(); j++) {
772         storage_ptr = storage_ptr_vec_ptr->at(j);
773
774         // 1. attempt to charge from Combustion curtailment first
775         for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
776             combustion_ptr = combustion_ptr_vec_ptr->at(i);
777             curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];
778
779             if (curtailment_kW <= 0) {
780                 continue;
781             }
782
783             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
784
785             if (acceptable_kW > curtailment_kW) {
786                 acceptable_kW = curtailment_kW;
787             }
788
789             combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
790             combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
791             storage_ptr->power_kW += acceptable_kW;
792         }
793
794         // 2. attempt to charge from Noncombustion curtailment second
795         for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
796             noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
797             curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];
798
799             if (curtailment_kW <= 0) {
800                 continue;
801             }
802
803             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
804
805             if (acceptable_kW > curtailment_kW) {
806                 acceptable_kW = curtailment_kW;
807             }
808
809             noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
810             noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
811             storage_ptr->power_kW += acceptable_kW;
812         }
813
814         // 3. attempt to charge from Renewable curtailment third
815         for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
816             renewable_ptr = renewable_ptr_vec_ptr->at(i);
817             curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];
818
819             if (curtailment_kW <= 0) {
820                 continue;
821             }
822
823             acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);
824
825             if (acceptable_kW > curtailment_kW) {
826                 acceptable_kW = curtailment_kW;
827             }
828
829             renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
830             renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
831             storage_ptr->power_kW += acceptable_kW;
832         }
833     }

```

```

834         // 4. commit charge
835         storage_ptr->commitCharge(
836             timestep,
837             dt_hrs,
838             storage_ptr->power_kW
839         );
840     }
841
842     return;
843 } /* __handleStorageCharging() */

```

4.3.3.12 __handleStorageDischarging()

```

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

```

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

Parameters

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

Returns

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

```

1162 {
1163     double discharging_kW = 0;
1164
1165     Storage* storage_ptr;
1166
1167     std::list<Storage*>::iterator iter;
1168     for (
1169         iter = storage_ptr_list.begin();
1170         iter != storage_ptr_list.end();
1171         iter++
1172     ){
1173         storage_ptr = (*iter);
1174
1175         discharging_kW = storage_ptr->getAvailablekW(dt_hrs);
1176
1177         if (discharging_kW > net_load_kW) {
1178             discharging_kW = net_load_kW;
1179         }
1180
1181         net_load_kW = storage_ptr->commitDischarge(
1182             timestep,
1183             dt_hrs,
1184             discharging_kW,
1185             net_load_kW
1186         );
1187     }
1188
1189     return net_load_kW;
1190 } /* __handleStorageDischarging() */

```

4.3.3.13 applyDispatchControl()

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

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

Parameters

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

```
1342 {
1343     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
1344         switch (this->control_mode) {
1345             case (ControlMode :: LOAD_FOLLOWING): {
1346                 if (this->net_load_vec_kW[i] <= 0) {
1347                     this->__applyLoadFollowingControl_CHARGING(
1348                         i,
1349                         electrical_load_ptr,
1350                         resources_ptr,
1351                         combustion_ptr_vec_ptr,
1352                         noncombustion_ptr_vec_ptr,
1353                         renewable_ptr_vec_ptr,
1354                         storage_ptr_vec_ptr
1355                     );
1356                 }
1357             }
1358             else {
1359                 this->__applyLoadFollowingControl_DISCHARGING(
1360                     i,
1361                     electrical_load_ptr,
1362                     resources_ptr,
1363                     combustion_ptr_vec_ptr,
1364                     noncombustion_ptr_vec_ptr,
1365                     renewable_ptr_vec_ptr,
1366                     storage_ptr_vec_ptr
1367                 );
1368             }
1369             break;
1370         }
1371     }
1372
1373     case (ControlMode :: CYCLE_CHARGING): {
1374         if (this->net_load_vec_kW[i] <= 0) {
1375             this->__applyCycleChargingControl_CHARGING(
1376                 i,
1377                 electrical_load_ptr,
1378                 resources_ptr,
1379                 combustion_ptr_vec_ptr,
1380                 noncombustion_ptr_vec_ptr,
1381                 renewable_ptr_vec_ptr,
1382                 storage_ptr_vec_ptr
1383             );
1384         }
1385     }
1386     else {
1387         this->__applyCycleChargingControl_DISCHARGING(
1388             i,
1389             electrical_load_ptr,
1390             resources_ptr,
1391             combustion_ptr_vec_ptr,
1392             noncombustion_ptr_vec_ptr,
1393             renewable_ptr_vec_ptr,
1394             storage_ptr_vec_ptr
```

```

1395         );
1396     }
1397
1398     break;
1399 }
1400
1401     default: {
1402         std::string error_str = "ERROR: Controller :: applyDispatchControl(): ";
1403         error_str += "control mode ";
1404         error_str += std::to_string(this->control_mode);
1405         error_str += " not recognized";
1406
1407         #ifdef _WIN32
1408             std::cout << error_str << std::endl;
1409         #endif
1410
1411         throw std::runtime_error(error_str);
1412
1413         break;
1414     }
1415 }
1416 }
1417
1418     return;
1419 } /* applyDispatchControl() */

```

4.3.3.14 clear()

```

void Controller::clear (
    void )

```

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

```

1434 {
1435     this->net_load_vec_kW.clear();
1436     this->missed_load_vec_kW.clear();
1437     this->combustion_map.clear();
1438
1439     return;
1440 } /* clear() */

```

4.3.3.15 init()

```

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

```

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

Parameters

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

```

1292 {
1293     // 1. compute net load
1294     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);
1295 }

```

```

1296 // 2. construct Combustion table
1297 this->__constructCombustionMap(combustion_ptr_vec_ptr);
1298
1299 return;
1300 } /* init() */

```

4.3.3.16 setControlMode()

```

void Controller::setControlMode (
    ControlMode control_mode )

```

Parameters

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

```

1226 {
1227     this->control_mode = control_mode;
1228
1229     switch(control_mode) {
1230         case (ControlMode :: LOAD_FOLLOWING): {
1231             this->control_string = "LOAD_FOLLOWING";
1232
1233             break;
1234         }
1235
1236         case (ControlMode :: CYCLE_CHARGING): {
1237             this->control_string = "CYCLE_CHARGING";
1238
1239             break;
1240         }
1241
1242         default: {
1243             std::string error_str = "ERROR: Controller :: setControlMode(): ";
1244             error_str += "control mode ";
1245             error_str += std::to_string(control_mode);
1246             error_str += " not recognized";
1247
1248             #ifdef _WIN32
1249                 std::cout << error_str << std::endl;
1250             #endif
1251
1252             throw std::runtime_error(error_str);
1253
1254             break;
1255         }
1256     }
1257
1258     return;
1259 } /* setControlMode() */

```

4.3.4 Member Data Documentation

4.3.4.1 combustion_map

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

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

4.3.4.2 control_mode

`ControlMode Controller::control_mode`

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

4.3.4.3 control_string

`std::string Controller::control_string`

A string describing the active ControlMode.

4.3.4.4 missed_load_vec_kW

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

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

4.3.4.5 net_load_vec_kW

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

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

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

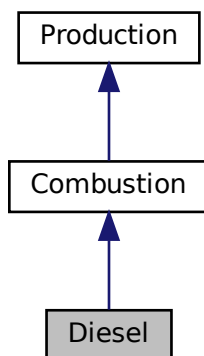
- header/[Controller.h](#)
- source/[Controller.cpp](#)

4.4 Diesel Class Reference

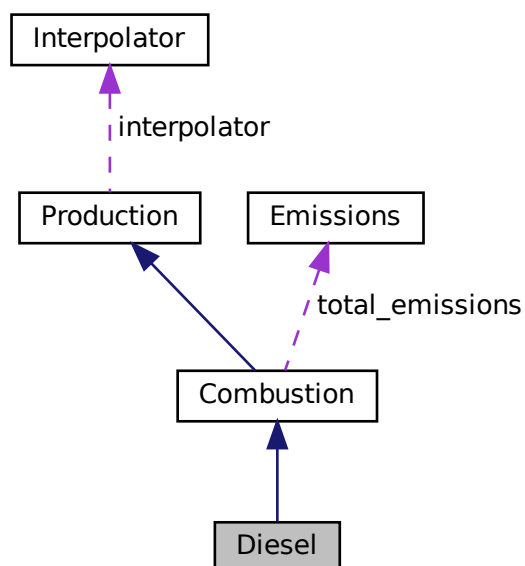
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

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

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

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

Public Attributes

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

Private Member Functions

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

4.4.1 Detailed Description

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

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Diesel() [1/2]

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

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

```
596 {
597     return;
598 } /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

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

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

Parameters

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

```
626 :
627 Combustion(
628     n_points,
629     n_years,
630     diesel_inputs.combustion_inputs
631 )
632 {
633     // 1. check inputs
634     this->__checkInputs(diesel_inputs);
635
636     // 2. set attributes
637     this->type = CombustionType :: DIESEL;
638     this->type_str = "DIESEL";
639
640     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
641
642     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
643
644     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
645     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
646     this->time_since_last_start_hrs = 0;
647
648     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
649     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
650     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
```

```

651     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
652     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
653     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
654
655     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
656         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
657     }
658
659     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
660         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
661     }
662
663     if (diesel_inputs.capital_cost < 0) {
664         this->capital_cost = this->__getGenericCapitalCost();
665     }
666
667     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
668         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
669     }
670
671     if (not this->is_sunk) {
672         this->capital_cost_vec[0] = this->capital_cost;
673     }
674
675     // 3. construction print
676     if (this->print_flag) {
677         std::cout << "Diesel object constructed at " << this << std::endl;
678     }
679
680     return;
681 } /* Diesel() */

```

4.4.2.3 ~Diesel()

```

Diesel::~Diesel (
    void )

```

Destructor for the [Diesel](#) class.

```

836 {
837     // 1. destruction print
838     if (this->print_flag) {
839         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
840     }
841
842     return;
843 } /* ~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.
----------------------	---

```

39 {

```

```

40 // 1. check fuel_cost_L
41 if (diesel_inputs.fuel_cost_L < 0) {
42     std::string error_str = "ERROR: Diesel(): ";
43     error_str += "DieselInputs::fuel_cost_L must be >= 0";
44
45     #ifdef _WIN32
46         std::cout << error_str << std::endl;
47     #endif
48
49     throw std::invalid_argument(error_str);
50 }
51
52 // 2. check CO2_emissions_intensity_kgL
53 if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
54     std::string error_str = "ERROR: Diesel(): ";
55     error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";
56
57     #ifdef _WIN32
58         std::cout << error_str << std::endl;
59     #endif
60
61     throw std::invalid_argument(error_str);
62 }
63
64 // 3. check CO_emissions_intensity_kgL
65 if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
66     std::string error_str = "ERROR: Diesel(): ";
67     error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";
68
69     #ifdef _WIN32
70         std::cout << error_str << std::endl;
71     #endif
72
73     throw std::invalid_argument(error_str);
74 }
75
76 // 4. check NOx_emissions_intensity_kgL
77 if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
78     std::string error_str = "ERROR: Diesel(): ";
79     error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";
80
81     #ifdef _WIN32
82         std::cout << error_str << std::endl;
83     #endif
84
85     throw std::invalid_argument(error_str);
86 }
87
88 // 5. check SOx_emissions_intensity_kgL
89 if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
90     std::string error_str = "ERROR: Diesel(): ";
91     error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";
92
93     #ifdef _WIN32
94         std::cout << error_str << std::endl;
95     #endif
96
97     throw std::invalid_argument(error_str);
98 }
99
100 // 6. check CH4_emissions_intensity_kgL
101 if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
102     std::string error_str = "ERROR: Diesel(): ";
103     error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";
104
105     #ifdef _WIN32
106         std::cout << error_str << std::endl;
107     #endif
108
109     throw std::invalid_argument(error_str);
110 }
111
112 // 7. check PM_emissions_intensity_kgL
113 if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
114     std::string error_str = "ERROR: Diesel(): ";
115     error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";
116
117     #ifdef _WIN32
118         std::cout << error_str << std::endl;
119     #endif
120
121     throw std::invalid_argument(error_str);
122 }
123
124 // 8. check minimum_load_ratio
125 if (diesel_inputs.minimum_load_ratio < 0) {
126     std::string error_str = "ERROR: Diesel(): ";

```

```

127         error_str += "DieselInputs::minimum_load_ratio must be >= 0";
128
129         #ifdef _WIN32
130             std::cout << error_str << std::endl;
131         #endif
132
133         throw std::invalid_argument(error_str);
134     }
135
136     // 9. check minimum_runtime_hrs
137     if (diesel_inputs.minimum_runtime_hrs < 0) {
138         std::string error_str = "ERROR: Diesel(): ";
139         error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";
140
141         #ifdef _WIN32
142             std::cout << error_str << std::endl;
143         #endif
144
145         throw std::invalid_argument(error_str);
146     }
147
148     // 10. check replace_running_hrs
149     if (diesel_inputs.replace_running_hrs <= 0) {
150         std::string error_str = "ERROR: Diesel(): ";
151         error_str += "DieselInputs::replace_running_hrs must be > 0";
152
153         #ifdef _WIN32
154             std::cout << error_str << std::endl;
155         #endif
156
157         throw std::invalid_argument(error_str);
158     }
159
160     return;
161 } /* __checkInputs() */

```

4.4.3.2 __getGenericCapitalCost()

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

Helper method to generate a generic diesel generator capital cost.

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

Returns

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

```

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

```

4.4.3.3 __getGenericFuelIntercept()

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

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

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

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

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

Returns

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

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

4.4.3.4 __getGenericFuelSlope()

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

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

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

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

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

Returns

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

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

4.4.3.5 __getGenericOpMaintCost()

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

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

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

Returns

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

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

4.4.3.6 __handleStartStop()

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

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

Parameters

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

```

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

```

4.4.3.7 __writeSummary()

```

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

```

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

Parameters

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

Reimplemented from [Combustion](#).

```

345 {
346     // 1. create filestream
347     write_path += "summary_results.md";
348     std::ofstream ofs;
349     ofs.open(write_path, std::ofstream::out);
350
351     // 2. write to summary results (markdown)
352     ofs << "# ";
353     ofs << std::to_string(int(ceil(this->capacity_kW)));
354     ofs << " kW DIESEL Summary Results\n";
355     ofs << "\n-----\n\n";
356
357     // 2.1. Production attributes
358     ofs << "## Production Attributes\n";
359     ofs << "\n";
360
361     ofs << "Capacity: " << this->capacity_kW << " kW \n";
362     ofs << "\n";
363
364     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
365     ofs << "Capital Cost: " << this->capital_cost << " \n";

```

```

366 ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
367     « " per kWh produced \n";
368 ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
369     « " \n";
370 ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
371     « " \n";
372 ofs « "Real Discount Rate (annual): " « this->real_discount_annual « " \n";
373 ofs « "\n";
374
375 ofs « "Replacement Running Hours: " « this->replace_running_hrs « " \n";
376 ofs « "\n-----\n\n";
377
378 // 2.2. Combustion attributes
379 ofs « "## Combustion Attributes\n";
380 ofs « "\n";
381
382 ofs « "Fuel Cost: " « this->fuel_cost_L « " per L \n";
383 ofs « "Nominal Fuel Escalation Rate (annual): "
384     « this->nominal_fuel_escalation_annual « " \n";
385 ofs « "Real Fuel Escalation Rate (annual): "
386     « this->real_fuel_escalation_annual « " \n";
387 ofs « "\n";
388
389 ofs « "Fuel Mode: " « this->fuel_mode_str « " \n";
390 switch (this->fuel_mode) {
391     case (FuelMode :: FUEL_MODE_LINEAR): {
392         ofs « "Linear Fuel Slope: " « this->linear_fuel_slope_LkWh
393             « " L/kWh \n";
394         ofs « "Linear Fuel Intercept Coefficient: "
395             « this->linear_fuel_intercept_LkWh « " L/kWh \n";
396         ofs « "\n";
397         break;
398     }
399
400     case (FuelMode :: FUEL_MODE_LOOKUP): {
401         ofs « "Fuel Consumption Data: " « this->interpolator.path_map_1D[0]
402             « " \n";
403         break;
404     }
405
406     default: {
407         // write nothing!
408         break;
409     }
410 }
411
412 ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
413     « this->CO2_emissions_intensity_kgL « " kg/L \n";
414
415 ofs « "Carbon Monoxide (CO) Emissions Intensity: "
416     « this->CO_emissions_intensity_kgL « " kg/L \n";
417
418 ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
419     « this->NOx_emissions_intensity_kgL « " kg/L \n";
420
421 ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
422     « this->SOx_emissions_intensity_kgL « " kg/L \n";
423
424 ofs « "Methane (CH4) Emissions Intensity: "
425     « this->CH4_emissions_intensity_kgL « " kg/L \n";
426
427 ofs « "Particulate Matter (PM) Emissions Intensity: "
428     « this->PM_emissions_intensity_kgL « " kg/L \n";
429
430 ofs « "\n-----\n\n";
431
432 // 2.3. Diesel attributes
433 ofs « "## Diesel Attributes\n";
434 ofs « "\n";
435
436 ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « " \n";
437 ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs \n";
438
439 ofs « "\n-----\n\n";
440
441 // 2.4. Diesel Results
442 ofs « "## Results\n";
443 ofs « "\n";
444
445 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
446 ofs « "\n";
447
448 ofs « "Total Dispatch: " « this->total_dispatch_kWh
449     « " kWh \n";
450
451 ofs « "Total Dispatch: " « this->total_dispatch_kWh
452     « " kWh \n";

```

```

453
454 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
455     << " per kWh dispatched \n";
456 ofs << "\n";
457
458 ofs << "Running Hours: " << this->running_hours << " \n";
459 ofs << "Starts: " << this->n_starts << " \n";
460 ofs << "Replacements: " << this->n_replacements << " \n";
461
462 ofs << "Total Fuel Consumed: " << this->total_fuel_consumed_L << " L "
463     << "(Annual Average: " << this->total_fuel_consumed_L / this->n_years
464     << " L/yr) \n";
465 ofs << "\n";
466
467 ofs << "Total Carbon Dioxide (CO2) Emissions: " <<
468     this->total_emissions.CO2_kg << " kg "
469     << "(Annual Average: " << this->total_emissions.CO2_kg / this->n_years
470     << " kg/yr) \n";
471
472 ofs << "Total Carbon Monoxide (CO) Emissions: " <<
473     this->total_emissions.CO_kg << " kg "
474     << "(Annual Average: " << this->total_emissions.CO_kg / this->n_years
475     << " kg/yr) \n";
476
477 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
478     this->total_emissions.NOx_kg << " kg "
479     << "(Annual Average: " << this->total_emissions.NOx_kg / this->n_years
480     << " kg/yr) \n";
481
482 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
483     this->total_emissions.SOx_kg << " kg "
484     << "(Annual Average: " << this->total_emissions.SOx_kg / this->n_years
485     << " kg/yr) \n";
486
487 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
488     << "(Annual Average: " << this->total_emissions.CH4_kg / this->n_years
489     << " kg/yr) \n";
490
491 ofs << "Total Particulate Matter (PM) Emissions: " <<
492     this->total_emissions.PM_kg << " kg "
493     << "(Annual Average: " << this->total_emissions.PM_kg / this->n_years
494     << " kg/yr) \n";
495
496 ofs << "\n-----\n\n";
497
498 ofs.close();
499 return;
500 } /* __writeSummary() */

```

4.4.3.8 __writeTimeSeries()

```

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

```

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

Parameters

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

Reimplemented from [Combustion](#).

```

530 {
531     // 1. create filestream
532     write_path += "time_series_results.csv";
533     std::ofstream ofs;

```



```

534     ofs.open(write_path, std::ofstream::out);
535
536     // 2. write time series results (comma separated value)
537     ofs << "Time (since start of data) [hrs],";
538     ofs << "Production [kW],";
539     ofs << "Dispatch [kW],";
540     ofs << "Storage [kW],";
541     ofs << "Curtailement [kW],";
542     ofs << "Is Running (N = 0 / Y = 1),";
543     ofs << "Fuel Consumption [L],";
544     ofs << "Fuel Cost (actual),";
545     ofs << "Carbon Dioxide (CO2) Emissions [kg],";
546     ofs << "Carbon Monoxide (CO) Emissions [kg],";
547     ofs << "Nitrogen Oxides (NOx) Emissions [kg],";
548     ofs << "Sulfur Oxides (SOx) Emissions [kg],";
549     ofs << "Methane (CH4) Emissions [kg],";
550     ofs << "Particulate Matter (PM) Emissions [kg],";
551     ofs << "Capital Cost (actual),";
552     ofs << "Operation and Maintenance Cost (actual),";
553     ofs << "\n";
554
555     for (int i = 0; i < max_lines; i++) {
556         ofs << time_vec_hrs_ptr->at(i) << ",";
557         ofs << this->production_vec_kW[i] << ",";
558         ofs << this->dispatch_vec_kW[i] << ",";
559         ofs << this->storage_vec_kW[i] << ",";
560         ofs << this->curtailment_vec_kW[i] << ",";
561         ofs << this->is_running_vec[i] << ",";
562         ofs << this->fuel_consumption_vec_L[i] << ",";
563         ofs << this->fuel_cost_vec[i] << ",";
564         ofs << this->CO2_emissions_vec_kg[i] << ",";
565         ofs << this->CO_emissions_vec_kg[i] << ",";
566         ofs << this->NOx_emissions_vec_kg[i] << ",";
567         ofs << this->SOx_emissions_vec_kg[i] << ",";
568         ofs << this->CH4_emissions_vec_kg[i] << ",";
569         ofs << this->PM_emissions_vec_kg[i] << ",";
570         ofs << this->capital_cost_vec[i] << ",";
571         ofs << this->operation_maintenance_cost_vec[i] << ",";
572         ofs << "\n";
573     }
574
575     ofs.close();
576     return;
577 } /* __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](#).

```

794 {
795     // 1. handle start/stop, enforce minimum runtime constraint
796     this->__handleStartStop(timestep, dt_hrs, production_kW);
797
798     // 2. invoke base class method
799     load_kW = Combustion::commit(
800         timestep,
801         dt_hrs,
802         production_kW,
803         load_kW
804     );
805
806     if (this->is_running) {
807         // 3. log time since last start
808         this->time_since_last_start_hrs += dt_hrs;
809
810         // 4. correct operation and maintenance costs (should be non-zero if idling)
811         if (production_kW <= 0) {
812             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
813
814             double operation_maintenance_cost =
815                 this->operation_maintenance_cost_kWh * produced_kWh;
816             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
817         }
818     }
819
820     return load_kW;
821 } /* 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](#).

```

699 {
700     // 1. reset attributes
701     this->time_since_last_start_hrs = 0;
702
703     // 2. invoke base class method
704     Combustion::handleReplacement(timestep);
705
706     return;
707 } /* __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](#).

```
739 {  
740     // 1. return on request of zero  
741     if (request_kW <= 0) {  
742         return 0;  
743     }  
744  
745     double deliver_kW = request_kW;  
746  
747     // 2. enforce capacity constraint  
748     if (deliver_kW > this->capacity_kW) {  
749         deliver_kW = this->capacity_kW;  
750     }  
751  
752     // 3. enforce minimum load ratio  
753     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {  
754         deliver_kW = this->minimum_load_ratio * this->capacity_kW;  
755     }  
756  
757     return deliver_kW;  
758 } /* 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:

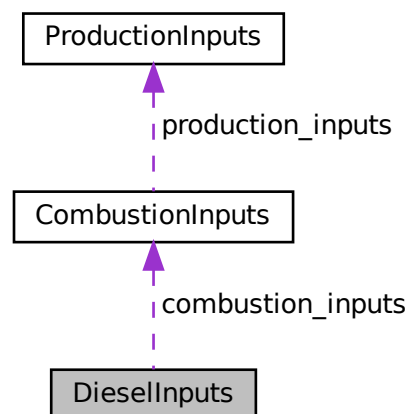
- [header/Production/Combustion/Diesel.h](#)
- [source/Production/Combustion/Diesel.cpp](#)

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 (NO_x) emissions intensity [kg/L].

- double `SOx_emissions_intensity_kgL` = 0.0042

Sulfur oxide (SO_x) 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.

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

4.6.2.2 ElectricalLoad() [2/2]

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

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

Parameters

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

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

4.6.2.3 ~ElectricalLoad()

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

Destructor for the [ElectricalLoad](#) class.

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

4.6.3 Member Function Documentation

4.6.3.1 clear()

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

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

```
157 {
158     this->n_points = 0;
```

```

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

```

4.6.3.2 readLoadData()

```

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

```

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

Parameters

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

```

79 {
80     // 1. clear
81     this->clear();
82
83     // 2. init CSV reader, record path
84     io::CSVReader<2> CSV(path_2_electrical_load_time_series);
85
86     CSV.read_header(
87         io::ignore_extra_column,
88         "Time (since start of data) [hrs]",
89         "Electrical Load [kW]"
90     );
91
92     this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;
93
94     // 3. read in time and load data, increment n_points, track min and max load
95     double time_hrs = 0;
96     double load_kW = 0;
97     double load_sum_kW = 0;
98
99     this->n_points = 0;
100
101     this->min_load_kW = std::numeric_limits<double>::infinity();
102     this->max_load_kW = -1 * std::numeric_limits<double>::infinity();
103
104     while (CSV.read_row(time_hrs, load_kW)) {
105         this->time_vec_hrs.push_back(time_hrs);
106         this->load_vec_kW.push_back(load_kW);
107
108         load_sum_kW += load_kW;
109
110         this->n_points++;
111
112         if (this->min_load_kW > load_kW) {
113             this->min_load_kW = load_kW;
114         }
115
116         if (this->max_load_kW < load_kW) {
117             this->max_load_kW = load_kW;
118         }
119     }
120
121     // 4. compute mean load
122     this->mean_load_kW = load_sum_kW / this->n_points;
123
124     // 5. set number of years (assuming 8,760 hours per year)
125     this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;
126

```

```

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

```

4.6.4 Member Data Documentation

4.6.4.1 dt_vec_hrs

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

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

4.6.4.2 load_vec_kW

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

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

4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

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

4.6.4.4 mean_load_kW

`double ElectricalLoad::mean_load_kW`

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

4.6.4.5 min_load_kW

```
double ElectricalLoad::min_load_kW
```

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

4.6.4.6 n_points

```
int ElectricalLoad::n_points
```

The number of points in the modelling time series.

4.6.4.7 n_years

```
double ElectricalLoad::n_years
```

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

4.6.4.8 path_2_electrical_load_time_series

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

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

4.6.4.9 time_vec_hrs

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

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

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

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

4.7 Emissions Struct Reference

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

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

Public Attributes

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

4.7.1 Detailed Description

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

4.7.2 Member Data Documentation

4.7.2.1 CH4_kg

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

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

4.7.2.2 CO2_kg

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

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

4.7.2.3 CO_kg

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

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

4.7.2.4 NOx_kg

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

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

4.7.2.5 PM_kg

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

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

4.7.2.6 SOx_kg

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

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

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

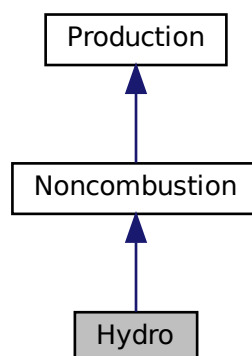
- [header/Production/Combustion/Combustion.h](#)

4.8 Hydro Class Reference

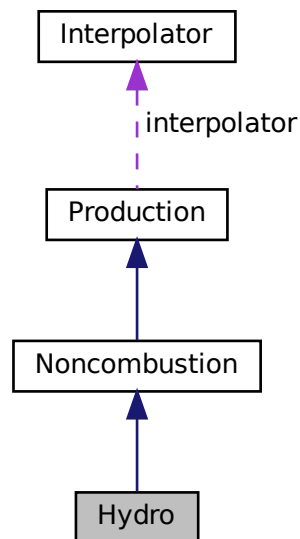
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

```
#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](#))
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_flow_m3hr](#)
The minimum required flow [m3/hr] for the asset to produce.
- 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 > [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.
- 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 [__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 through the turbine.
- 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.

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

4.8.2.2 Hydro() [2/2]

```
Hydro::Hydro (
    int n_points,
    double n_years,
    HydroInputs hydro_inputs )
```

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.

```
704 :
705 Noncombustion(
706     n_points,
707     n_years,
708     hydro_inputs.noncombustion_inputs
709 )
710 {
711     // 1. check inputs
712     this->__checkInputs(hydro_inputs);
713
714     // 2. set attributes
715     this->type = NoncombustionType :: HYDRO;
716     this->type_str = "HYDRO";
717
718     this->resource_key = hydro_inputs.resource_key;
719
720     this->turbine_type = hydro_inputs.turbine_type;
721
722     this->fluid_density_kgm3 = hydro_inputs.fluid_density_kgm3;
723     this->net_head_m = hydro_inputs.net_head_m;
724
725     this->reservoir_capacity_m3 = hydro_inputs.reservoir_capacity_m3;
726     this->init_reservoir_state = hydro_inputs.init_reservoir_state;
727     this->stored_volume_m3 =
728         hydro_inputs.init_reservoir_state * hydro_inputs.reservoir_capacity_m3;
729
730     this->minimum_flow_m3hr = this->__getMinimumFlowm3hr();
731     this->maximum_flow_m3hr = this->__getMaximumFlowm3hr();
732
733     this->turbine_flow_vec_m3hr.resize(this->n_points, 0);
734     this->stored_volume_vec_m3.resize(this->n_points, 0);
735
736     if (hydro_inputs.capital_cost < 0) {
737         this->capital_cost = this->__getGenericCapitalCost();
738     }
739
740     if (hydro_inputs.operation_maintenance_cost_kWh < 0) {
741         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
742     }
743
744     if (not this->is_sunk) {
745         this->capital_cost_vec[0] = this->capital_cost;
746     }
747 }
```

```

748     return;
749 } /* Hydro() */

```

4.8.2.3 ~Hydro()

```

Hydro::~~Hydro (
    void )

```

Destructor for the [Hydro](#) class.

```

914 {
915     // 1. destruction print
916     if (this->print_flag) {
917         std::cout << "Hydro object at " << this << " destroyed" << std::endl;
918     }
919
920     return;
921 } /* ~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.
---------------------	--

```

39 {
40     // 1. check fluid_density_kgm3
41     if (hydro_inputs.fluid_density_kgm3 <= 0) {
42         std::string error_str = "ERROR: Hydro(): fluid_density_kgm3 must be > 0";
43
44         #ifdef _WIN32
45             std::cout << error_str << std::endl;
46         #endif
47
48         throw std::invalid_argument(error_str);
49     }
50
51     // 2. check net_head_m
52     if (hydro_inputs.net_head_m <= 0) {
53         std::string error_str = "ERROR: Hydro(): net_head_m must be > 0";
54
55         #ifdef _WIN32
56             std::cout << error_str << std::endl;
57         #endif
58
59         throw std::invalid_argument(error_str);
60     }
61
62     // 3. check reservoir_capacity_m3
63     if (hydro_inputs.reservoir_capacity_m3 < 0) {
64         std::string error_str = "ERROR: Hydro(): reservoir_capacity_m3 must be >= 0";
65
66         #ifdef _WIN32
67             std::cout << error_str << std::endl;
68         #endif

```

```

69
70     throw std::invalid_argument(error_str);
71 }
72
73 // 4. check init_reservoir_state
74 if (
75     hydro_inputs.init_reservoir_state < 0 or
76     hydro_inputs.init_reservoir_state > 1
77 ) {
78     std::string error_str = "ERROR: Hydro(): init_reservoir_state must be in ";
79     error_str += "the closed interval [0, 1]";
80
81     #ifdef _WIN32
82         std::cout << error_str << std::endl;
83     #endif
84
85     throw std::invalid_argument(error_str);
86 }
87
88 return;
89 } /* __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.

This model was obtained by way of surveying an assortment of published hydroelectric operational data, and then constructing a best fit model.

Ref: [Marks'](#)

Returns

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

```

259 {
260     if (flow_m3hr <= 0) {
261         return 0;
262     }
263
264     // 1. compute power ratio
265     double power_ratio =
266         this->fluid_density_kgm3 * 9.81 * this->net_head_m * (flow_m3hr / 3600);
267
268     power_ratio /= 1000 * this->capacity_kW;
269
270     // 2. get normalized power
271     double normalized_power = 0;
272
273     switch (this->turbine_type) {
274         case (HydroTurbineType :: HYDRO_TURBINE_PELTON): {
275             if (power_ratio <= 0.023529) {
276                 normalized_power = 0;
277             }
278             else if (power_ratio >= 1.166301) {
279                 normalized_power = 1;
280             }
281             else {
282                 normalized_power = 0.87448308 * power_ratio - 0.02108607;
283             }
284
285             break;
286         }
287
288         case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
289             if (power_ratio <= 0.2164706) {
290                 normalized_power = 0;
291             }
292             else if (power_ratio >= 1.1952933) {

```

```

293         normalized_power = 1;
294     }
295     else {
296         normalized_power = (
297             1.61681669 * pow(power_ratio, 0.49508545) - 0.76355563
298         );
299     }
300
301     break;
302 }
303
304 default: {
305     //..
306     break;
307 }
308 }
309 }
310
311 if (normalized_power < 0) {
312     normalized_power = 0;
313 }
314 else if (normalized_power > 1) {
315     normalized_power = 1;
316 }
317
318 return normalized_power * this->capacity_kW;
319 } /* __flowToPower() */

```

4.8.3.3 __getAvailableFlow()

```

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

```

Helper method to determine what flow is currently available through 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.

```

399 {
400     double flow_m3hr = 0;
401
402     // 1. add flow available from reservoir
403     flow_m3hr += this->stored_volume_m3 / dt_hrs;
404
405     // 2. add flow available from resource
406     flow_m3hr += hydro_resource_m3hr;
407
408     // 3. cap at maximum flow
409     if (flow_m3hr > this->maximum_flow_m3hr) {
410         flow_m3hr = this->maximum_flow_m3hr;
411     }
412
413     return flow_m3hr;
414 } /* __getAvailableFlow() */

```

4.8.3.4 __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].

```
108 {
109     double capital_cost_per_kW = 0; //<-- WIP: need something better here
110
111     return capital_cost_per_kW * this->capacity_kW;
112 } /* __getGenericCapitalCost() */
```

4.8.3.5 __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].

```
133 {
134     double operation_maintenance_cost_kWh = 0; //<-- WIP: need something better here
135
136     return operation_maintenance_cost_kWh;
137 } /* __getGenericOpMaintCost() */
```

4.8.3.6 __getMaximumFlowm3hr()

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

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

This model was obtained by way of surveying an assortment of published hydroelectric operational data, and then constructing a best fit model.

Ref: [Marks'](#)

Returns

The maximum productive flow [m3/hr].

```

210 {
211     double coefficient = 0;
212
213     switch (this->turbine_type) {
214         case (HydroTurbineType :: HYDRO_TURBINE_PELTON): {
215             coefficient = 1.166301;
216
217             break;
218         }
219
220         case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
221             coefficient = 1.1952933;
222
223             break;
224         }
225
226         default: {
227             //..
228
229             break;
230         }
231     }
232
233     double maximum_flow_m3hr = (1000 * 3600 * coefficient * this->capacity_kW) /
234         (this->fluid_density_kgm3 * 9.81 * this->net_head_m);
235
236     return maximum_flow_m3hr;
237 } /* __getMaximumFlowm3hr() */

```

4.8.3.7 __getMinimumFlowm3hr()

```

double Hydro::__getMinimumFlowm3hr (
    void ) [private]

```

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

This model was obtained by way of surveying an assortment of published hydroelectric operational data, and then constructing a best fit model.

Ref: [Marks'](#)

Returns

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

```

160 {
161     double coefficient = 0;
162
163     switch (this->turbine_type) {
164         case (HydroTurbineType :: HYDRO_TURBINE_PELTON): {
165             coefficient = 0.023529;
166
167             break;
168         }
169
170         case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
171             coefficient = 0.2164706;
172
173             break;
174         }
175
176         default: {
177             //..
178
179             break;
180         }
181     }
182
183     double minimum_flow_m3hr = (1000 * 3600 * coefficient * this->capacity_kW) /
184         (this->fluid_density_kgm3 * 9.81 * this->net_head_m);
185
186     return minimum_flow_m3hr;
187 } /* __getMinimumFlowm3hr() */

```

4.8.3.8 __powerToFlow()

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

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

This model was obtained by way of surveying an assortment of published hydroelectric operational data, and then constructing a best fit model.

Ref: [Marks'](#)

Returns

```
341 {
342     if (power_kW <= 0) {
343         return 0;
344     }
345     double flow_m3hr = 0;
346     switch (this->turbine_type) {
347     case (HydroTurbineType :: HYDRO_TURBINE_PELTON): {
348         flow_m3hr = 3600.0 / 0.87448308;
349         flow_m3hr *= (power_kW / this->capacity_kW) + 0.02108607;
350         flow_m3hr *= 1000 * this->capacity_kW;
351         flow_m3hr /= this->fluid_density_kgm3 * 9.81 * this->net_head_m;
352         break;
353     }
354     case (HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
355         flow_m3hr = pow(
356             (1.0 / 1.61681669) * ((power_kW / this->capacity_kW) + 0.76355563),
357             1.0 / 0.49508545
358         );
359         flow_m3hr *= 3600 * 1000 * this->capacity_kW;
360         flow_m3hr /= this->fluid_density_kgm3 * 9.81 * this->net_head_m;
361         break;
362     }
363     default: {
364         //..
365         break;
366     }
367     }
368     return flow_m3hr;
369 }
370 /* __powerToFlow() */
```

4.8.3.9 __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].

```

447 {
448     // 1. get flow, log
449     double flow_m3hr = this->__powerToFlow(production_kW);
450     this->turbine_flow_vec_m3hr[timestep] = flow_m3hr;
451
452     // 2. update reservoir state, log
453     if (this->reservoir_capacity_m3 > 0) {
454         this->stored_volume_m3 += hydro_resource_m3hr * dt_hrs;
455         this->stored_volume_m3 -= flow_m3hr * dt_hrs;
456
457         if (this->stored_volume_m3 < 0) {
458             this->stored_volume_m3 = 0;
459         }
460
461         else if (this->stored_volume_m3 > this->reservoir_capacity_m3) {
462             this->stored_volume_m3 = this->reservoir_capacity_m3;
463         }
464
465         this->stored_volume_vec_m3[timestep] = this->stored_volume_m3;
466     }
467
468     return;
469 } /* __updateState() */

```

4.8.3.10 __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](#).

```

487 {
488     // 1. create filestream
489     write_path += "summary_results.md";
490     std::ofstream ofs;
491     ofs.open(write_path, std::ofstream::out);
492
493     // 2. write to summary results (markdown)
494     ofs << "# ";
495     ofs << std::to_string(int(ceil(this->capacity_kW)));
496     ofs << " kW HYDRO Summary Results\n";
497     ofs << "\n-----\n\n";
498
499     // 2.1. Production attributes
500     ofs << "## Production Attributes\n";
501     ofs << "\n";
502
503     ofs << "Capacity: " << this->capacity_kW << " kW \n";
504     ofs << "\n";
505
506     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
507     ofs << "Capital Cost: " << this->capital_cost << " \n";
508     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
509         << " per kWh produced \n";
510     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual

```

```

511         « " \n";
512 ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
513     « " \n";
514 ofs « "Real Discount Rate (annual): " « this->real_discount_annual « " \n";
515 ofs « "\n";
516
517 ofs « "Replacement Running Hours: " « this->replace_running_hrs « " \n";
518 ofs « "\n-----\n\n";
519
520 // 2.2. Noncombustion attributes
521 ofs « "## Noncombustion Attributes\n";
522 ofs « "\n";
523
524 //...
525
526 ofs « "\n-----\n\n";
527
528 // 2.3. Hydro attributes
529 ofs « "## Hydro Attributes\n";
530 ofs « "\n";
531
532 ofs « "Fluid Density: " « this->fluid_density_kgm3 « " kg/m3 \n";
533 ofs « "Net Head: " « this->net_head_m « " m \n";
534 ofs « "\n";
535
536 ofs « "Reservoir Volume: " « this->reservoir_capacity_m3 « " m3 \n";
537 ofs « "Reservoir Initial State: " « this->init_reservoir_state « " \n";
538 ofs « "\n";
539
540 ofs « "Turbine Type: ";
541 switch(this->turbine_type) {
542     case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
543         ofs « "PELTON";
544
545         break;
546     }
547
548     case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
549         ofs « "FRANCIS";
550
551         break;
552     }
553
554     default: {
555         // write nothing!
556
557         break;
558     }
559 }
560 ofs « " \n";
561 ofs « "Minimum Flow: " « this->minimum_flow_m3hr « " m3/hr \n";
562 ofs « "Maximum Flow: " « this->maximum_flow_m3hr « " m3/hr \n";
563 ofs « "\n";
564
565 ofs « "\n-----\n\n";
566
567 // 2.4. Hydro Results
568 ofs « "## Results\n";
569 ofs « "\n";
570
571 ofs « "Net Present Cost: " « this->net_present_cost « " \n";
572 ofs « "\n";
573
574 ofs « "Total Dispatch: " « this->total_dispatch_kWh
575     « " kWh \n";
576
577 ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
578     « " per kWh dispatched \n";
579 ofs « "\n";
580
581 ofs « "Running Hours: " « this->running_hours « " \n";
582 ofs « "Replacements: " « this->n_replacements « " \n";
583
584 //...
585
586 ofs « "\n-----\n\n";
587
588 ofs.close();
589 return;
590 } /* __writeSummary() */

```

4.8.3.11 `__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 <code>time_vec_hrs</code> attribute of the ElectricalLoad .
<i>max_lines</i>	The maximum number of lines of output to write.

Reimplemented from [Noncombustion](#).

```
620 {
621     // 1. create filestream
622     write_path += "time_series_results.csv";
623     std::ofstream ofs;
624     ofs.open(write_path, std::ofstream::out);
625
626     // 2. write time series results (comma separated value)
627     ofs << "Time (since start of data) [hrs],";
628     ofs << "Production [kW],";
629     ofs << "Dispatch [kW],";
630     ofs << "Storage [kW],";
631     ofs << "Curtailment [kW],";
632     ofs << "Is Running (N = 0 / Y = 1),";
633     ofs << "Turbine Flow [m3/hr],";
634     ofs << "Stored Volume [m3],";
635     ofs << "Capital Cost (actual),";
636     ofs << "Operation and Maintenance Cost (actual),";
637     ofs << "\n";
638
639     for (int i = 0; i < max_lines; i++) {
640         ofs << time_vec_hrs_ptr->at(i) << ", ";
641         ofs << this->production_vec_kW[i] << ", ";
642         ofs << this->dispatch_vec_kW[i] << ", ";
643         ofs << this->storage_vec_kW[i] << ", ";
644         ofs << this->curtailment_vec_kW[i] << ", ";
645         ofs << this->is_running_vec[i] << ", ";
646         ofs << this->turbine_flow_vec_m3hr[i] << ", ";
647         ofs << this->stored_volume_vec_m3[i] << ", ";
648         ofs << this->capital_cost_vec[i] << ", ";
649         ofs << this->operation_maintenance_cost_vec[i] << ", ";
650         ofs << "\n";
651     }
652     ofs.close();
653     return;
654 } /* __writeTimeSeries() */
```

4.8.3.12 `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](#).

```

881 {
882     // 1. invoke base class method
883     load_kW = Noncombustion :: commit(
884         timestep,
885         dt_hrs,
886         production_kW,
887         load_kW
888     );
889
890     // 2. update state and record
891     this->__updateState(
892         timestep,
893         dt_hrs,
894         production_kW,
895         hydro_resource_m3hr
896     );
897
898     return load_kW;
899 } /* commit() */

```

4.8.3.13 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](#).

```

767 {
768     // 1. reset attributes
769     //...
770
771     // 2. invoke base class method
772     Noncombustion :: handleReplacement(timestep);
773
774     return;
775 } /* __handleReplacement() */

```

4.8.3.14 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](#).

```
811 {
812     // 1. return on request of zero
813     if (request_kW <= 0) {
814         return 0;
815     }
816     // 2. set flow to available
817     double flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);
818     if (flow_m3hr < this->minimum_flow_m3hr) {
819         return 0;
820     }
821     // 3. limit flow to request (and max)
822     double request_m3hr = this->__powerToFlow(request_kW);
823     if (flow_m3hr > request_m3hr) {
824         flow_m3hr = request_m3hr;
825     }
826     if (flow_m3hr > this->maximum_flow_m3hr) {
827         flow_m3hr = this->maximum_flow_m3hr;
828     }
829     // 4. map flow to production
830     double production_kW = this->__flowToPower(flow_m3hr);
831     // 5. limit production to capacity
832     if (production_kW > this->capacity_kW) {
833         production_kW = this->capacity_kW;
834     }
835     return production_kW;
836 } /* 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.

4.8.4.5 net_head_m

```
double Hydro::net_head_m
```

The net head [m] of the asset.

4.8.4.6 reservoir_capacity_m3

```
double Hydro::reservoir_capacity_m3
```

The capacity [m3] of the hydro reservoir.

4.8.4.7 stored_volume_m3

```
double Hydro::stored_volume_m3
```

The volume [m3] of stored fluid.

4.8.4.8 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.

4.8.4.9 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.

4.8.4.10 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:

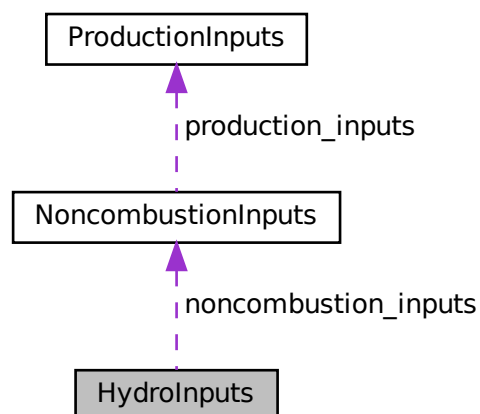
- header/Production/Noncombustion/[Hydro.h](#)
- source/Production/Noncombustion/[Hydro.cpp](#)

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 = 10`
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 = 10
```

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.

```
670 {
671     //...
672
673     return;
674 } /* Interpolator() */
```

4.10.2.2 ~Interpolator()

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

Destructor for the [Interpolator](#) class.

```
856 {
857     ///  
858     ///  
859     return;
860 } /* ~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_x</i>	The query value to be interpolated.

```
108 {
109     ///  
110     // 1. key error
111     if (this->interp_map_1D.count(data_key) == 0) {
112         std::string error_str = "ERROR: Interpolator::interp1D() ";
113         error_str += "data key ";
114         error_str += std::to_string(data_key);
115         error_str += " has not been registered";
116         #ifdef _WIN32
117             std::cout << error_str << std::endl;
118         #endif
119         throw std::invalid_argument(error_str);
120     }
121 }
122
123     // 2. bounds error
124     if (
125         interp_x < this->interp_map_1D[data_key].min_x or
126         interp_x > this->interp_map_1D[data_key].max_x
127     ) {
128         std::string error_str = "ERROR: Interpolator::interp1D() ";
129         error_str += "interpolation value ";
130         error_str += std::to_string(interp_x);
131         error_str += " is outside of the given interpolation data domain";
132         #ifdef _WIN32
133             std::cout << error_str << std::endl;
134         #endif
135         throw std::invalid_argument(error_str);
136     }
137 }
138
139     return;
140 } /* __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.

```
164 {
165     // 1. key error
166     if (this->interp_map_2D.count(data_key) == 0) {
167         std::string error_str = "ERROR: Interpolator::interp2D() ";
168         error_str += "data key ";
169         error_str += std::to_string(data_key);
170         error_str += " has not been registered";
171
172         #ifdef _WIN32
173             std::cout << error_str << std::endl;
174         #endif
175
176         throw std::invalid_argument(error_str);
177     }
178
179     // 2. bounds error (x_interp)
180     if (
181         interp_x < this->interp_map_2D[data_key].min_x or
182         interp_x > this->interp_map_2D[data_key].max_x
183     ) {
184         std::string error_str = "ERROR: Interpolator::interp2D() ";
185         error_str += "interpolation value interp_x = ";
186         error_str += std::to_string(interp_x);
187         error_str += " is outside of the given interpolation data domain";
188
189         #ifdef _WIN32
190             std::cout << error_str << std::endl;
191         #endif
192
193         throw std::invalid_argument(error_str);
194     }
195
196     // 2. bounds error (y_interp)
197     if (
198         interp_y < this->interp_map_2D[data_key].min_y or
199         interp_y > this->interp_map_2D[data_key].max_y
200     ) {
201         std::string error_str = "ERROR: Interpolator::interp2D() ";
202         error_str += "interpolation value interp_y = ";
203         error_str += std::to_string(interp_y);
204         error_str += " is outside of the given interpolation data domain";
205
206         #ifdef _WIN32
207             std::cout << error_str << std::endl;
208         #endif
209
210         throw std::invalid_argument(error_str);
211     }
212
213     return;
214 } /* __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.
-----------------	--

```
40 {
41     if (this->interp_map_1D.count(data_key) > 0) {
42         std::string error_str = "ERROR: Interpolator::addData1D() ";
43         error_str += "data key (1D) ";
44         error_str += std::to_string(data_key);
45         error_str += " is already in use";
46
47         #ifdef _WIN32
48             std::cout << error_str << std::endl;
49         #endif
50
51         throw std::invalid_argument(error_str);
52     }
53
54     return;
55 } /* __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.
-----------------	--

```
72 {
73     if (this->interp_map_2D.count(data_key) > 0) {
74         std::string error_str = "ERROR: Interpolator::addData2D() ";
75         error_str += "data key (2D) ";
76         error_str += std::to_string(data_key);
77         error_str += " is already in use";
78
79         #ifdef _WIN32
80             std::cout << error_str << std::endl;
81         #endif
82
83         throw std::invalid_argument(error_str);
84     }
85
86     return;
87 } /* __checkDataKey2D() */
```

4.10.3.5 `__getDataStringMatrix()`

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

```

389 {
390     // 1. create input file stream
391     std::ifstream ifs;
392     ifs.open(path_2_data);
393
394     // 2. check that open() worked
395     if (not ifs.is_open()) {
396         std::string error_str = "ERROR: Interpolator::__getDataStringMatrix() ";
397         error_str += " failed to open ";
398         error_str += path_2_data;
399
400         #ifdef _WIN32
401             std::cout << error_str << std::endl;
402         #endif
403
404         throw std::invalid_argument(error_str);
405     }
406
407     // 3. read file line by line
408     bool is_header = true;
409     std::string line;
410     std::vector<std::string> line_split_vec;
411     std::vector<std::vector<std::string>> string_matrix;
412
413     while (not ifs.eof()) {
414         std::getline(ifs, line);
415
416         if (is_header) {
417             is_header = false;
418             continue;
419         }
420
421         line_split_vec = this->__splitCommaSeparatedString(line);
422
423         if (not line_split_vec.empty()) {
424             string_matrix.push_back(line_split_vec);
425         }
426     }
427
428     ifs.close();
429     return string_matrix;
430 } /* __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.

```

306 {
307     int idx = 0;
308     while (
309         not (interp_x >= x_vec_ptr->at(idx) and interp_x <= x_vec_ptr->at(idx + 1))
310     ) {
311         idx++;
312     }
313
314     return idx;
315 } /* __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.

```
271 {
272     for (size_t i = 0; i < str.size(); i++) {;
273         if (isalpha(str[i])) {
274             return true;
275         }
276     }
277     return false;
278 } /* __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.

```
450 {
451     // 1. get string matrix
452     std::vector<std::vector<std::string>> string_matrix =
453         this->__getDataStringMatrix(path_2_data);
454
455     // 2. read string matrix contents into 1D interpolation struct
456     InterpolatorStruct1D interp_struct_1D;
457
458     interp_struct_1D.n_points = string_matrix.size();
459     interp_struct_1D.x_vec.resize(interp_struct_1D.n_points, 0);
460     interp_struct_1D.y_vec.resize(interp_struct_1D.n_points, 0);
461
462     for (int i = 0; i < interp_struct_1D.n_points; i++) {
463         try {
464             interp_struct_1D.x_vec[i] = std::stod(string_matrix[i][0]);
465             interp_struct_1D.y_vec[i] = std::stod(string_matrix[i][1]);
466         }
467
468         catch (...) {
469             this->__throwReadError(path_2_data, 1);
470         }
471     }
472
473     interp_struct_1D.min_x = interp_struct_1D.x_vec[0];
474     interp_struct_1D.max_x = interp_struct_1D.x_vec[interp_struct_1D.n_points - 1];
```



```

475
476 // 3. write struct to map
477 this->interp_map_1D.insert(
478     std::pair<int, InterpolatorStruct1D>(data_key, interp_struct_1D)
479 );
480
481 /*
482 // ==== TEST PRINT ==== //
483 std::cout << std::endl;
484 std::cout << path_2_data << std::endl;
485 std::cout << "-----" << std::endl;
486
487 std::cout << "n_points: " << this->interp_map_1D[data_key].n_points << std::endl;
488
489 std::cout << "x_vec: [";
490 for (
491     int i = 0;
492     i < this->interp_map_1D[data_key].n_points;
493     i++
494 ) {
495     std::cout << this->interp_map_1D[data_key].x_vec[i] << ", ";
496 }
497 std::cout << "]" << std::endl;
498
499 std::cout << "y_vec: [";
500 for (
501     int i = 0;
502     i < this->interp_map_1D[data_key].n_points;
503     i++
504 ) {
505     std::cout << this->interp_map_1D[data_key].y_vec[i] << ", ";
506 }
507 std::cout << "]" << std::endl;
508
509 std::cout << std::endl;
510 // ==== END TEST PRINT ==== //
511 /**/
512
513 return;
514 } /* __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.

```

534 {
535     // 1. get string matrix
536     std::vector<std::vector<std::string>> string_matrix =
537         this->__getDataStringMatrix(path_2_data);
538
539     // 2. read string matrix contents into 2D interpolation map
540     InterpolatorStruct2D interp_struct_2D;
541
542     interp_struct_2D.n_rows = string_matrix.size() - 1;
543     interp_struct_2D.n_cols = string_matrix[0].size() - 1;
544
545     interp_struct_2D.x_vec.resize(interp_struct_2D.n_cols, 0);
546     interp_struct_2D.y_vec.resize(interp_struct_2D.n_rows, 0);
547
548     interp_struct_2D.z_matrix.resize(interp_struct_2D.n_rows, {});
549
550     for (int i = 0; i < interp_struct_2D.n_rows; i++) {
551         interp_struct_2D.z_matrix[i].resize(interp_struct_2D.n_cols, 0);
552     }

```

```

553
554     for (size_t i = 1; i < string_matrix[0].size(); i++) {
555         try {
556             interp_struct_2D.x_vec[i - 1] = std::stod(string_matrix[0][i]);
557         }
558
559         catch (...) {
560             this->__throwReadError(path_2_data, 2);
561         }
562     }
563
564     interp_struct_2D.min_x = interp_struct_2D.x_vec[0];
565     interp_struct_2D.max_x = interp_struct_2D.x_vec[interp_struct_2D.n_cols - 1];
566
567     for (size_t i = 1; i < string_matrix.size(); i++) {
568         try {
569             interp_struct_2D.y_vec[i - 1] = std::stod(string_matrix[i][0]);
570         }
571
572         catch (...) {
573             this->__throwReadError(path_2_data, 2);
574         }
575     }
576
577     interp_struct_2D.min_y = interp_struct_2D.y_vec[0];
578     interp_struct_2D.max_y = interp_struct_2D.y_vec[interp_struct_2D.n_rows - 1];
579
580     for (size_t i = 1; i < string_matrix.size(); i++) {
581         for (size_t j = 1; j < string_matrix[0].size(); j++) {
582             try {
583                 interp_struct_2D.z_matrix[i - 1][j - 1] = std::stod(string_matrix[i][j]);
584             }
585
586             catch (...) {
587                 this->__throwReadError(path_2_data, 2);
588             }
589         }
590     }
591
592     // 3. write struct to map
593     this->interp_map_2D.insert(
594         std::pair<int, InterpolatorStruct2D>(data_key, interp_struct_2D)
595     );
596
597     /*
598     // ==== TEST PRINT ==== //
599     std::cout << std::endl;
600     std::cout << path_2_data << std::endl;
601     std::cout << "-----" << std::endl;
602
603     std::cout << "n_rows: " << this->interp_map_2D[data_key].n_rows << std::endl;
604     std::cout << "n_cols: " << this->interp_map_2D[data_key].n_cols << std::endl;
605
606     std::cout << "x_vec: [";
607     for (
608         int i = 0;
609         i < this->interp_map_2D[data_key].n_cols;
610         i++
611     ) {
612         std::cout << this->interp_map_2D[data_key].x_vec[i] << ", ";
613     }
614     std::cout << "]" << std::endl;
615
616     std::cout << "y_vec: [";
617     for (
618         int i = 0;
619         i < this->interp_map_2D[data_key].n_rows;
620         i++
621     ) {
622         std::cout << this->interp_map_2D[data_key].y_vec[i] << ", ";
623     }
624     std::cout << "]" << std::endl;
625
626     std::cout << "z_matrix:" << std::endl;
627     for (
628         int i = 0;
629         i < this->interp_map_2D[data_key].n_rows;
630         i++
631     ) {
632         std::cout << "\t[";
633
634         for (
635             int j = 0;
636             j < this->interp_map_2D[data_key].n_cols;
637             j++
638         ) {
639             std::cout << this->interp_map_2D[data_key].z_matrix[i][j] << ", ";

```

```

640         }
641
642         std::cout << "]" << std::endl;
643     }
644     std::cout << std::endl;
645
646     std::cout << std::endl;
647     // ==== END TEST PRINT ==== //
648     /* */
649
650     return;
651 } /* __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.

```

344 {
345     std::vector<std::string> str_split_vec;
346
347     size_t idx = 0;
348     std::string substr;
349
350     while ((idx = str.find(',')) != std::string::npos) {
351         substr = str.substr(0, idx);
352
353         if (substr == break_str) {
354             break;
355         }
356
357         str_split_vec.push_back(substr);
358
359         str.erase(0, idx + 1);
360     }
361
362     return str_split_vec;
363 } /* __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.

```

235 {
236     std::string error_str = "ERROR:  Interpolator::addData";
237     error_str += std::to_string(dimensions);
238     error_str += "D()  ";
239     error_str += " failed to read ";
240     error_str += path_2_data;
241     error_str += " (this is probably a std::stod() error; is there non-numeric ";
242     error_str += "data where only numeric data should be?)";
243
244     #ifdef _WIN32
245         std::cout << error_str << std::endl;
246     #endif
247
248     throw std::runtime_error(error_str);
249
250     return;
251 } /* __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.

```

694 {
695     // 1. check key
696     this->__checkDataKey1D(data_key);
697
698     // 2. read data into map
699     this->__readData1D(data_key, path_2_data);
700
701     // 3. record path
702     this->path_map_1D.insert(std::pair<int, std::string>(data_key, path_2_data));
703
704     return;
705 } /* 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.

```

725 {
726     // 1. check key
727     this->__checkDataKey2D(data_key);
728
729     // 2. read data into map
730     this->__readData2D(data_key, path_2_data);
731
732     // 3. record path
733     this->path_map_2D.insert(std::pair<int, std::string>(data_key, path_2_data));
734
735     return;
736 } /* 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.

```

758 {
759     // 1. check bounds
760     this->__checkBounds1D(data_key, interp_x);
761
762     // 2. get interpolation index
763     int idx = this->__getInterpolationIndex(
764         interp_x,
765         &(this->interp_map_1D[data_key].x_vec)
766     );
767
768     // 3. perform interpolation
769     double x_0 = this->interp_map_1D[data_key].x_vec[idx];
770     double x_1 = this->interp_map_1D[data_key].x_vec[idx + 1];
771
772     double y_0 = this->interp_map_1D[data_key].y_vec[idx];
773     double y_1 = this->interp_map_1D[data_key].y_vec[idx + 1];
774
775     double interp_y = ((y_1 - y_0) / (x_1 - x_0)) * (interp_x - x_0) + y_0;
776
777     return interp_y;
778 } /* 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.

```
803 {
804     // 1. check bounds
805     this->__checkBounds2D(data_key, interp_x, interp_y);
806
807     // 2. get interpolation indices
808     int idx_x = this->__getInterpolationIndex(
809         interp_x,
810         &(this->interp_map_2D[data_key].x_vec)
811     );
812
813     int idx_y = this->__getInterpolationIndex(
814         interp_y,
815         &(this->interp_map_2D[data_key].y_vec)
816     );
817
818     // 3. perform first horizontal interpolation
819     double x_0 = this->interp_map_2D[data_key].x_vec[idx_x];
820     double x_1 = this->interp_map_2D[data_key].x_vec[idx_x + 1];
821
822     double z_0 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x];
823     double z_1 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x + 1];
824
825     double interp_z_0 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
826
827     // 4. perform second horizontal interpolation
828     z_0 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x];
829     z_1 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x + 1];
830
831     double interp_z_1 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;
832
833     // 5. perform vertical interpolation
834     double y_0 = this->interp_map_2D[data_key].y_vec[idx_y];
835     double y_1 = this->interp_map_2D[data_key].y_vec[idx_y + 1];
836
837     double interp_z =
838         ((interp_z_1 - interp_z_0) / (y_1 - y_0)) * (interp_y - y_0) + interp_z_0;
839
840     return interp_z;
841 } /* 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:

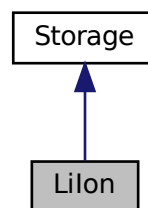
- header/[Interpolator.h](#)

4.13 Lilon Class Reference

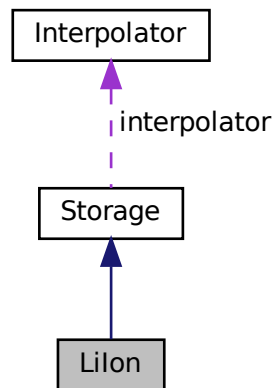
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

```
#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

- double [dynamic_energy_capacity_kWh](#)
The dynamic (i.e. degrading) energy capacity [kWh] of the asset.
- double [SOH](#)
The state of health of the asset.
- double [replace_SOH](#)
The state of health at which the asset is considered "dead" and must be replaced.
- double [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(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.

```
646 {
647     return;
648 } /* LiIon() */
```

4.13.2.2 Lilon() [2/2]

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

Constructor (intended) for the [Lilon](#) 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 Lilon constructor inputs.

```
676 :
677 Storage(
678     n_points,
679     n_years,
680     liion_inputs.storage_inputs
681 )
682 {
683     // 1. check inputs
684     this->__checkInputs(liion_inputs);
685
686     // 2. set attributes
687     this->type = StorageType :: LIION;
688     this->type_str = "LIION";
```

```

689
690     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
691     this->SOH = 1;
692     this->replace_SOH = liion_inputs.replace_SOH;
693
694     this->degradation_alpha = liion_inputs.degradation_alpha;
695     this->degradation_beta = liion_inputs.degradation_beta;
696     this->degradation_B_hat_cal_0 = liion_inputs.degradation_B_hat_cal_0;
697     this->degradation_r_cal = liion_inputs.degradation_r_cal;
698     this->degradation_Ea_cal_0 = liion_inputs.degradation_Ea_cal_0;
699     this->degradation_a_cal = liion_inputs.degradation_a_cal;
700     this->degradation_s_cal = liion_inputs.degradation_s_cal;
701     this->gas_constant_JmolK = liion_inputs.gas_constant_JmolK;
702     this->temperature_K = liion_inputs.temperature_K;
703
704     this->init_SOC = liion_inputs.init_SOC;
705     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
706
707     this->min_SOC = liion_inputs.min_SOC;
708     this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
709     this->max_SOC = liion_inputs.max_SOC;
710
711     this->charging_efficiency = liion_inputs.charging_efficiency;
712     this->discharging_efficiency = liion_inputs.discharging_efficiency;
713
714     if (liion_inputs.capital_cost < 0) {
715         this->capital_cost = this->__getGenericCapitalCost();
716     }
717
718     if (liion_inputs.operation_maintenance_cost_kWh < 0) {
719         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
720     }
721
722     if (not this->is_sunk) {
723         this->capital_cost_vec[0] = this->capital_cost;
724     }
725
726     this->SOH_vec.resize(this->n_points, 0);
727
728     // 3. construction print
729     if (this->print_flag) {
730         std::cout << "LiIon object constructed at " << this << std::endl;
731     }
732
733     return;
734 } /* LiIon() */

```

4.13.2.3 ~LiIon()

```

LiIon::~~LiIon (
    void )

```

Destructor for the [LiIon](#) class.

```

990 {
991     // 1. destruction print
992     if (this->print_flag) {
993         std::cout << "LiIon object at " << this << " destroyed" << std::endl;
994     }
995
996     return;
997 } /* ~LiIon() */

```

4.13.3 Member Function Documentation

4.13.3.1 __checkInputs()

```

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

```

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

Parameters

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

```

39 {
40     // 1. check replace_SOH
41     if (liion_inputs.replace_SOH < 0 or liion_inputs.replace_SOH > 1) {
42         std::string error_str = "ERROR: LiIon(): replace_SOH must be in the closed ";
43         error_str += "interval [0, 1]";
44
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check init_SOC
53     if (liion_inputs.init_SOC < 0 or liion_inputs.init_SOC > 1) {
54         std::string error_str = "ERROR: LiIon(): init_SOC must be in the closed ";
55         error_str += "interval [0, 1]";
56
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     // 3. check min_SOC
65     if (liion_inputs.min_SOC < 0 or liion_inputs.min_SOC > 1) {
66         std::string error_str = "ERROR: LiIon(): min_SOC must be in the closed ";
67         error_str += "interval [0, 1]";
68
69         #ifdef _WIN32
70             std::cout << error_str << std::endl;
71         #endif
72
73         throw std::invalid_argument(error_str);
74     }
75
76     // 4. check hysteresis_SOC
77     if (liion_inputs.hysteresis_SOC < 0 or liion_inputs.hysteresis_SOC > 1) {
78         std::string error_str = "ERROR: LiIon(): hysteresis_SOC must be in the closed ";
79         error_str += "interval [0, 1]";
80
81         #ifdef _WIN32
82             std::cout << error_str << std::endl;
83         #endif
84
85         throw std::invalid_argument(error_str);
86     }
87
88     // 5. check max_SOC
89     if (liion_inputs.max_SOC < 0 or liion_inputs.max_SOC > 1) {
90         std::string error_str = "ERROR: LiIon(): max_SOC must be in the closed ";
91         error_str += "interval [0, 1]";
92
93         #ifdef _WIN32
94             std::cout << error_str << std::endl;
95         #endif
96
97         throw std::invalid_argument(error_str);
98     }
99
100    // 6. check charging_efficiency
101    if (liion_inputs.charging_efficiency <= 0 or liion_inputs.charging_efficiency > 1) {
102        std::string error_str = "ERROR: LiIon(): charging_efficiency must be in the ";
103        error_str += "half-open interval (0, 1]";
104
105        #ifdef _WIN32
106            std::cout << error_str << std::endl;
107        #endif
108
109        throw std::invalid_argument(error_str);
110    }
111
112    // 7. check discharging_efficiency
113    if (
114        liion_inputs.discharging_efficiency <= 0 or
115        liion_inputs.discharging_efficiency > 1
116    ) {
117        std::string error_str = "ERROR: LiIon(): discharging_efficiency must be in the ";
118        error_str += "half-open interval (0, 1]";
119
120        #ifdef _WIN32

```



```

121         std::cout << error_str << std::endl;
122     #endif
123
124     throw std::invalid_argument(error_str);
125 }
126
127 // 8. check degradation_alpha
128 if (lilion_inputs.degradation_alpha <= 0) {
129     std::string error_str = "ERROR: LiIon(): degradation_alpha must be > 0";
130
131     #ifdef _WIN32
132         std::cout << error_str << std::endl;
133     #endif
134
135     throw std::invalid_argument(error_str);
136 }
137
138 // 9. check degradation_beta
139 if (lilion_inputs.degradation_beta <= 0) {
140     std::string error_str = "ERROR: LiIon(): degradation_beta 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 // 10. check degradation_B_hat_cal_0
150 if (lilion_inputs.degradation_B_hat_cal_0 <= 0) {
151     std::string error_str = "ERROR: LiIon(): degradation_B_hat_cal_0 must be > 0";
152
153     #ifdef _WIN32
154         std::cout << error_str << std::endl;
155     #endif
156
157     throw std::invalid_argument(error_str);
158 }
159
160 // 11. check degradation_r_cal
161 if (lilion_inputs.degradation_r_cal < 0) {
162     std::string error_str = "ERROR: LiIon(): degradation_r_cal must be >= 0";
163
164     #ifdef _WIN32
165         std::cout << error_str << std::endl;
166     #endif
167
168     throw std::invalid_argument(error_str);
169 }
170
171 // 12. check degradation_Ea_cal_0
172 if (lilion_inputs.degradation_Ea_cal_0 <= 0) {
173     std::string error_str = "ERROR: LiIon(): degradation_Ea_cal_0 must be > 0";
174
175     #ifdef _WIN32
176         std::cout << error_str << std::endl;
177     #endif
178
179     throw std::invalid_argument(error_str);
180 }
181
182 // 13. check degradation_a_cal
183 if (lilion_inputs.degradation_a_cal < 0) {
184     std::string error_str = "ERROR: LiIon(): degradation_a_cal must be >= 0";
185
186     #ifdef _WIN32
187         std::cout << error_str << std::endl;
188     #endif
189
190     throw std::invalid_argument(error_str);
191 }
192
193 // 14. check degradation_s_cal
194 if (lilion_inputs.degradation_s_cal < 0) {
195     std::string error_str = "ERROR: LiIon(): degradation_s_cal must be >= 0";
196
197     #ifdef _WIN32
198         std::cout << error_str << std::endl;
199     #endif
200
201     throw std::invalid_argument(error_str);
202 }
203
204 // 15. check gas_constant_JmolK
205 if (lilion_inputs.gas_constant_JmolK <= 0) {
206     std::string error_str = "ERROR: LiIon(): gas_constant_JmolK must be > 0";
207

```

```

208         #ifdef _WIN32
209             std::cout << error_str << std::endl;
210         #endif
211
212         throw std::invalid_argument(error_str);
213     }
214
215     // 16. check temperature_K
216     if (liion_inputs.temperature_K < 0) {
217         std::string error_str = "ERROR: LiIon(): temperature_K must be >= 0";
218
219         #ifdef _WIN32
220             std::cout << error_str << std::endl;
221         #endif
222
223         throw std::invalid_argument(error_str);
224     }
225
226     return;
227 } /* __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 \[2023\]](#)

Parameters

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

Returns

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

```

427 {
428     double B_cal = this->degradation_B_hat_cal_0 *
429         exp(this->degradation_r_cal * SOC);
430
431     return B_cal;
432 } /* __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 \[2023\]](#)

Parameters

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

Returns

The activation energy value for the given state of charge.

```

454 {
455     double Ea_cal = this->degradation_Ea_cal_0;
456
457     Ea_cal -= this->degradation_a_cal *
458         (exp(this->degradation_s_cal * SOC) - 1);
459
460     return Ea_cal;
461 } /* __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].

```

250 {
251     double capital_cost_per_kWh = 250 * pow(this->energy_capacity_kWh, -0.15) + 650;
252
253     return capital_cost_per_kWh * this->energy_capacity_kWh;
254 } /* __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].

```

278 {
279     return 0.01;
280 } /* __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.

```
348 {
349     // 1. model degradation
350     this->__modelDegradation(dt_hrs, charging_discharging_kW);
351
352     // 2. update and record
353     this->SOH_vec[timestep] = this->SOH;
354     this->dynamic_energy_capacity_kWh = this->SOH * this->energy_capacity_kWh;
355
356     return;
357 } /* __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 \[2023\]](#)

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.

```
380 {
381     // 1. compute SOC
382     double SOC = this->charge_kWh / this->energy_capacity_kWh;
383
384     // 2. compute C-rate and corresponding acceleration factor
385     double C_rate = charging_discharging_kW / this->power_capacity_kW;
386
387     double C_acceleration_factor =
388         1 + this->degradation_alpha * pow(C_rate, this->degradation_beta);
389
390     // 3. compute dSOH / dt
391     double B_cal = __getBcal(SOC);
392     double Ea_cal = __getEacal(SOC);
393
394     double dSOH_dt = B_cal *
395         exp((-1 * Ea_cal) / (this->gas_constant_JmolK * this->temperature_K));
396
397     dSOH_dt *= dSOH_dt;
398     dSOH_dt *= 1 / (2 * this->SOH);
399     dSOH_dt *= C_acceleration_factor;
400 }
```

```

401     // 4. update state of health
402     this->SOH -= dSOH_dt * dt_hrs;
403
404     return;
405 } /* __modelDegradation() */

```

4.13.3.8 __toggleDepleted()

```

void LiIon::__toggleDepleted (
    void ) [private]

```

Helper method to toggle the `is_depleted` attribute of [Lilon](#).

```

295 {
296     if (this->is_depleted) {
297         double hysteresis_charge_kWh = this->hysteresis_SOC * this->energy_capacity_kWh;
298
299         if (hysteresis_charge_kWh > this->dynamic_energy_capacity_kWh) {
300             hysteresis_charge_kWh = this->dynamic_energy_capacity_kWh;
301         }
302
303         if (this->charge_kWh >= hysteresis_charge_kWh) {
304             this->is_depleted = false;
305         }
306     }
307
308     else {
309         double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
310
311         if (this->charge_kWh <= min_charge_kWh) {
312             this->is_depleted = true;
313         }
314     }
315
316     return;
317 } /* __toggleDepleted() */

```

4.13.3.9 __writeSummary()

```

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

```

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

Parameters

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

Reimplemented from [Storage](#).

```

479 {
480     // 1. create filestream
481     write_path += "summary_results.md";
482     std::ofstream ofs;
483     ofs.open(write_path, std::ofstream::out);
484
485     // 2. write summary results (markdown)
486     ofs << "# ";
487     ofs << std::to_string(int(ceil(this->power_capacity_kW)));
488     ofs << " kW ";
489     ofs << std::to_string(int(ceil(this->energy_capacity_kWh)));
490     ofs << " kWh LIION Summary Results\n";
491     ofs << "\n-----\n\n";

```

```

492
493 // 2.1. Storage attributes
494 ofs << "## Storage Attributes\n";
495 ofs << "\n";
496 ofs << "Power Capacity: " << this->power_capacity_kW << "kW \n";
497 ofs << "Energy Capacity: " << this->energy_capacity_kWh << "kWh \n";
498 ofs << "\n";
499
500 ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
501 ofs << "Capital Cost: " << this->capital_cost << " \n";
502 ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
503 << " per kWh charged/discharged \n";
504 ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
505 << " \n";
506 ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
507 << " \n";
508 ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
509
510 ofs << "\n-----\n\n";
511
512 // 2.2. LiIon attributes
513 ofs << "## LiIon Attributes\n";
514 ofs << "\n";
515
516 ofs << "Charging Efficiency: " << this->charging_efficiency << " \n";
517 ofs << "Discharging Efficiency: " << this->discharging_efficiency << " \n";
518 ofs << "\n";
519
520 ofs << "Initial State of Charge: " << this->init_SOC << " \n";
521 ofs << "Minimum State of Charge: " << this->min_SOC << " \n";
522 ofs << "Hyteresis State of Charge: " << this->hysteresis_SOC << " \n";
523 ofs << "Maximum State of Charge: " << this->max_SOC << " \n";
524 ofs << "\n";
525
526 ofs << "Replacement State of Health: " << this->replace_SOH << " \n";
527 ofs << "\n";
528
529 ofs << "Degradation Acceleration Coeff.: " << this->degradation_alpha << " \n";
530 ofs << "Degradation Acceleration Exp.: " << this->degradation_beta << " \n";
531 ofs << "Degradation Base Pre-Exponential Factor: "
532 << this->degradation_B_hat_cal_0 << " 1/sqrt(hrs) \n";
533 ofs << "Degradation Dimensionless Constant (r_cal): "
534 << this->degradation_r_cal << " \n";
535 ofs << "Degradation Base Activation Energy: "
536 << this->degradation_Ea_cal_0 << " J/mol \n";
537 ofs << "Degradation Pre-Exponential Factor: "
538 << this->degradation_a_cal << " J/mol \n";
539 ofs << "Degradation Dimensionless Constant (s_cal): "
540 << this->degradation_s_cal << " \n";
541 ofs << "Universal Gas Constant: " << this->gas_constant_JmolK
542 << " J/mol.K \n";
543 ofs << "Absolute Environmental Temperature: " << this->temperature_K << " K \n";
544 ofs << "\n";
545
546 ofs << "\n-----\n\n";
547
548 // 2.3. LiIon Results
549 ofs << "## Results\n";
550 ofs << "\n";
551
552 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
553 ofs << "\n";
554
555 ofs << "Total Discharge: " << this->total_discharge_kWh
556 << " kWh \n";
557
558 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
559 << " per kWh dispatched \n";
560 ofs << "\n";
561
562 ofs << "Replacements: " << this->n_replacements << " \n";
563
564 ofs << "\n-----\n\n";
565 ofs.close();
566 return;
567 } /* __writeSummary() */

```

4.13.3.10 __writeTimeSeries()

```

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

```

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

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

Parameters

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

Reimplemented from [Storage](#).

```
598 {
599     // 1. create filestream
600     write_path += "time_series_results.csv";
601     std::ofstream ofs;
602     ofs.open(write_path, std::ofstream::out);
603
604     // 2. write time series results (comma separated value)
605     ofs << "Time (since start of data) [hrs],";
606     ofs << "Charging Power [kW],";
607     ofs << "Discharging Power [kW],";
608     ofs << "Charge (at end of timestep) [kWh],";
609     ofs << "State of Health (at end of timestep) [ ],";
610     ofs << "Capital Cost (actual),";
611     ofs << "Operation and Maintenance Cost (actual),";
612     ofs << "\n";
613
614     for (int i = 0; i < max_lines; i++) {
615         ofs << time_vec_hrs_ptr->at(i) << ",";
616         ofs << this->charging_power_vec_kW[i] << ",";
617         ofs << this->discharging_power_vec_kW[i] << ",";
618         ofs << this->charge_vec_kWh[i] << ",";
619         ofs << this->SOH_vec[i] << ",";
620         ofs << this->capital_cost_vec[i] << ",";
621         ofs << this->operation_maintenance_cost_vec[i] << ",";
622         ofs << "\n";
623     }
624
625     ofs.close();
626     return;
627 } /* __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](#).

```
881 {
```

```

882 // 1. record charging power
883 this->charging_power_vec_kW[timestep] = charging_kW;
884
885 // 2. update charge and record
886 this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
887 this->charge_vec_kWh[timestep] = this->charge_kWh;
888
889 // 3. toggle depleted flag (if applicable)
890 this->__toggleDepleted();
891
892 // 4. model degradation
893 this->__handleDegradation(timestep, dt_hrs, charging_kW);
894
895 // 5. trigger replacement (if applicable)
896 if (this->SOH <= this->replace_SOH) {
897     this->handleReplacement(timestep);
898 }
899
900 // 6. capture operation and maintenance costs (if applicable)
901 if (charging_kW > 0) {
902     this->operation_maintenance_cost_vec[timestep] = charging_kW * dt_hrs *
903         this->operation_maintenance_cost_kWh;
904 }
905
906 this->power_kW = 0;
907 return;
908 } /* 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](#).

```

944 {
945     // 1. record discharging power, update total
946     this->discharging_power_vec_kW[timestep] = discharging_kW;
947     this->total_discharge_kWh += discharging_kW * dt_hrs;
948
949     // 2. update charge and record
950     this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
951     this->charge_vec_kWh[timestep] = this->charge_kWh;
952
953     // 3. update load
954     load_kW -= discharging_kW;
955
956     // 4. toggle depleted flag (if applicable)
957     this->__toggleDepleted();

```



```

958
959 // 5. model degradation
960 this->__handleDegradation(timestep, dt_hrs, discharging_kW);
961
962 // 6. trigger replacement (if applicable)
963 if (this->SOH <= this->replace_SOH) {
964     this->handleReplacement(timestep);
965 }
966
967 // 7. capture operation and maintenance costs (if applicable)
968 if (discharging_kW > 0) {
969     this->operation_maintenance_cost_vec[timestep] = discharging_kW * dt_hrs *
970     this->operation_maintenance_cost_kWh;
971 }
972
973 this->power_kW = 0;
974 return load_kW;
975 } /* 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

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

Returns

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

Reimplemented from [Storage](#).

```

825 {
826     // 1. get max charge
827     double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;
828
829     if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
830         max_charge_kWh = this->dynamic_energy_capacity_kWh;
831     }
832
833     // 2. compute acceptable power
834     // (accounting for the power currently being charged/discharged by the asset)
835     double acceptable_kW =
836         (max_charge_kWh - this->charge_kWh) /
837         (this->charging_efficiency * dt_hrs);
838
839     acceptable_kW -= this->power_kW;
840
841     if (acceptable_kW <= 0) {
842         return 0;
843     }
844
845     // 3. apply power constraint
846     if (acceptable_kW > this->power_capacity_kW) {
847         acceptable_kW = this->power_capacity_kW;
848     }
849
850     return acceptable_kW;
851 } /* 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

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

Returns

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

Reimplemented from [Storage](#).

```
784 {
785     // 1. get min charge
786     double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;
787
788     // 2. compute available power
789     // (accounting for the power currently being charged/discharged by the asset)
790     double available_kW =
791         ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
792         dt_hrs;
793
794     available_kW -= this->power_kW;
795
796     if (available_kW <= 0) {
797         return 0;
798     }
799
800     // 3. apply power constraint
801     if (available_kW > this->power_capacity_kW) {
802         available_kW = this->power_capacity_kW;
803     }
804
805     return available_kW;
806 } /* 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](#).

```
752 {
753     // 1. reset attributes
754     this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
755     this->SOH = 1;
756
757     // 2. invoke base class method
758     Storage::handleReplacement(timestep);
759
760     // 3. correct attributes
```

```
761     this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
762     this->is_depleted = false;
763
764     return;
765 } /* __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(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 gas_constant_JmolK

```
double LiIon::gas_constant_JmolK
```

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

4.13.4.12 hysteresis_SOC

```
double LiIon::hysteresis_SOC
```

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

4.13.4.13 init_SOC

```
double LiIon::init_SOC
```

The initial state of charge of the asset.

4.13.4.14 max_SOC

```
double LiIon::max_SOC
```

The maximum state of charge of the asset.

4.13.4.15 min_SOC

```
double LiIon::min_SOC
```

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

4.13.4.16 replace_SOH

```
double LiIon::replace_SOH
```

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

4.13.4.17 SOH

```
double LiIon::SOH
```

The state of health of the asset.

4.13.4.18 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.

4.13.4.19 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:

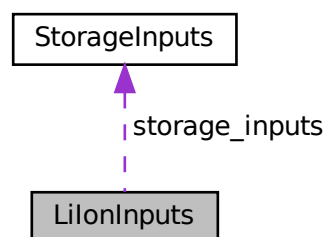
- header/Storage/[Lilon.h](#)
- source/Storage/[Lilon.cpp](#)

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.
- 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 \[2023\]](#)

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 replace_SOH

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

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

4.14.2.18 storage_inputs

```
StorageInputs LiIonInputs::storage_inputs
```

An encapsulated [StorageInputs](#) instance.

4.14.2.19 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:

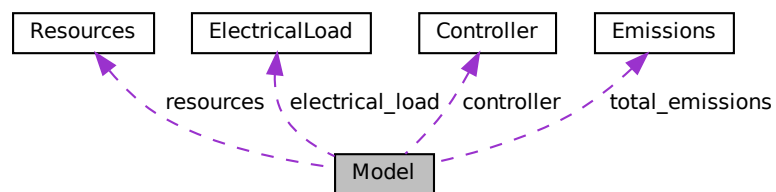
- header/Storage/[Lilon.h](#)

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_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.
- `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.

```
564 {
565     return;
566 } /* 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.
---------------------	--

```
583 {
584     // 1. check inputs
585     this->__checkInputs(model_inputs);
586
587     // 2. read in electrical load data
588     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
589
590     // 3. set control mode
591     this->controller.setControlMode(model_inputs.control_mode);
592
593     // 4. set public attributes
594     this->total_fuel_consumed_L = 0;
595     this->net_present_cost = 0;
596     this->total_dispatch_discharge_kWh = 0;
597     this->levellized_cost_of_energy_kWh = 0;
598
599     return;
600 } /* Model() */
```

4.15.2.3 ~Model()

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

Destructor for the [Model](#) class.

```
1110 {
1111     this->clear();
1112     return;
1113 } /* ~Model() */
```

4.15.3 Member Function Documentation

4.15.3.1 __checkInputs()

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

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

Parameters

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

```

40 {
41     // 1. check path_2_electrical_load_time_series
42     if (model_inputs.path_2_electrical_load_time_series.empty()) {
43         std::string error_str = "ERROR: Model() path_2_electrical_load_time_series ";
44         error_str += "cannot be empty";
45
46         #ifdef _WIN32
47             std::cout << error_str << std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     return;
54 } /* __checkInputs() */

```

4.15.3.2 __computeEconomics()

```

void Model::__computeEconomics (
    void ) [private]

```

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

```

236 {
237     this->__computeNetPresentCost();
238     this->__computeLevellingCostOfEnergy();
239
240     return;
241 } /* __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.

```

70 {
71     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
72         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
73
74         this->total_fuel_consumed_L +=
75             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
76
77         this->total_emissions.CO2_kg +=
78             this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
79
80         this->total_emissions.CO_kg +=
81             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
82
83         this->total_emissions.NOx_kg +=
84             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
85
86         this->total_emissions.SOx_kg +=
87             this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
88
89         this->total_emissions.CH4_kg +=
90             this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
91
92         this->total_emissions.PM_kg +=
93             this->combustion_ptr_vec[i]->total_emissions.PM_kg;
94     }
95
96     return;
97 } /* __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.

```
183 {
184     // 1. account for Combustion economics in levellized cost of energy
185     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
186         this->levellized_cost_of_energy_kWh +=
187             (
188                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
189                 this->combustion_ptr_vec[i]->total_dispatch_kWh
190             ) / this->total_dispatch_discharge_kWh;
191     }
192
193     // 2. account for Noncombustion economics in levellized cost of energy
194     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
195         this->levellized_cost_of_energy_kWh +=
196             (
197                 this->noncombustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
198                 this->noncombustion_ptr_vec[i]->total_dispatch_kWh
199             ) / this->total_dispatch_discharge_kWh;
200     }
201
202     // 3. account for Renewable economics in levellized cost of energy
203     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
204         this->levellized_cost_of_energy_kWh +=
205             (
206                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
207                 this->renewable_ptr_vec[i]->total_dispatch_kWh
208             ) / this->total_dispatch_discharge_kWh;
209     }
210
211     // 4. account for Storage economics in levellized cost of energy
212     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
213         this->levellized_cost_of_energy_kWh +=
214             (
215                 this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
216                 this->storage_ptr_vec[i]->total_discharge_kWh
217             ) / this->total_dispatch_discharge_kWh;
218     }
219
220     return;
221 } /* __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.

```
113 {
114     // 1. account for Combustion economics in net present cost
115     // increment total dispatch
116     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
117         this->combustion_ptr_vec[i]->computeEconomics(
118             &(this->electrical_load.time_vec_hrs)
119         );
120
121         this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;
122
123         this->total_dispatch_discharge_kWh +=
124             this->combustion_ptr_vec[i]->total_dispatch_kWh;
125     }
126
127     // 2. account for Noncombustion economics in net present cost
128     // increment total dispatch
129     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
130         this->noncombustion_ptr_vec[i]->computeEconomics(
131             &(this->electrical_load.time_vec_hrs)
132         );
133     }
```



```

134         this->net_present_cost += this->noncombustion_ptr_vec[i]->net_present_cost;
135
136         this->total_dispatch_discharge_kWh +=
137             this->noncombustion_ptr_vec[i]->total_dispatch_kWh;
138     }
139
140     // 3. account for Renewable economics in net present cost,
141     // increment total dispatch
142     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
143         this->renewable_ptr_vec[i]->computeEconomics(
144             &(this->electrical_load.time_vec_hrs)
145         );
146
147         this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;
148
149         this->total_dispatch_discharge_kWh +=
150             this->renewable_ptr_vec[i]->total_dispatch_kWh;
151     }
152
153     // 4. account for Storage economics in net present cost
154     // increment total dispatch
155     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
156         this->storage_ptr_vec[i]->computeEconomics(
157             &(this->electrical_load.time_vec_hrs)
158         );
159
160         this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;
161
162         this->total_dispatch_discharge_kWh +=
163             this->storage_ptr_vec[i]->total_dispatch_kWh;
164     }
165
166     return;
167 } /* __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.
-------------------	--

```

259 {
260     // 1. create subdirectory
261     write_path += "Model/";
262     std::filesystem::create_directory(write_path);
263
264     // 2. create filestream
265     write_path += "summary_results.md";
266     std::ofstream ofs;
267     ofs.open(write_path, std::ofstream::out);
268
269     // 3. write summary results (markdown)
270     ofs << "# Model Summary Results\n";
271     ofs << "\n-----\n\n";
272
273     // 3.1. ElectricalLoad
274     ofs << "## Electrical Load\n";
275     ofs << "\n";
276     ofs << "Path: " <<
277         this->electrical_load.path_2_electrical_load_time_series << " \n";
278     ofs << "Data Points: " << this->electrical_load.n_points << " \n";
279     ofs << "Years: " << this->electrical_load.n_years << " \n";
280     ofs << "Min: " << this->electrical_load.min_load_kW << " kW \n";
281     ofs << "Mean: " << this->electrical_load.mean_load_kW << " kW \n";
282     ofs << "Max: " << this->electrical_load.max_load_kW << " kW \n";
283     ofs << "\n-----\n\n";
284
285     // 3.2. Controller

```

```

286 ofs << "## Controller\n";
287 ofs << "\n";
288 ofs << "Control Mode: " << this->controller.control_string << " \n";
289 ofs << "\n-----\n\n";
290
291 // 3.3. Resources (1D)
292 ofs << "## 1D Renewable Resources\n";
293 ofs << "\n";
294
295 std::map<int, std::string>::iterator string_map_1D_iter =
296 this->resources.string_map_1D.begin();
297 std::map<int, std::string>::iterator path_map_1D_iter =
298 this->resources.path_map_1D.begin();
299
300 while (
301     string_map_1D_iter != this->resources.string_map_1D.end() and
302     path_map_1D_iter != this->resources.path_map_1D.end()
303 ) {
304     ofs << "Resource Key: " << string_map_1D_iter->first << " \n";
305     ofs << "Type: " << string_map_1D_iter->second << " \n";
306     ofs << "Path: " << path_map_1D_iter->second << " \n";
307     ofs << "\n";
308
309     string_map_1D_iter++;
310     path_map_1D_iter++;
311 }
312
313 ofs << "\n-----\n\n";
314
315 // 3.4. Resources (2D)
316 ofs << "## 2D Renewable Resources\n";
317 ofs << "\n";
318
319 std::map<int, std::string>::iterator string_map_2D_iter =
320 this->resources.string_map_2D.begin();
321 std::map<int, std::string>::iterator path_map_2D_iter =
322 this->resources.path_map_2D.begin();
323
324 while (
325     string_map_2D_iter != this->resources.string_map_2D.end() and
326     path_map_2D_iter != this->resources.path_map_2D.end()
327 ) {
328     ofs << "Resource Key: " << string_map_2D_iter->first << " \n";
329     ofs << "Type: " << string_map_2D_iter->second << " \n";
330     ofs << "Path: " << path_map_2D_iter->second << " \n";
331     ofs << "\n";
332
333     string_map_2D_iter++;
334     path_map_2D_iter++;
335 }
336
337 ofs << "\n-----\n\n";
338
339 // 3.5. Combustion
340 ofs << "## Combustion Assets\n";
341 ofs << "\n";
342
343 for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
344     ofs << "Asset Index: " << i << " \n";
345     ofs << "Type: " << this->combustion_ptr_vec[i]->type_str << " \n";
346     ofs << "Capacity: " << this->combustion_ptr_vec[i]->capacity_kW << " kW \n";
347     ofs << "\n";
348 }
349
350 ofs << "\n-----\n\n";
351
352 // 3.6. Noncombustion
353 ofs << "## Noncombustion Assets\n";
354 ofs << "\n";
355
356 for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
357     ofs << "Asset Index: " << i << " \n";
358     ofs << "Type: " << this->noncombustion_ptr_vec[i]->type_str << " \n";
359     ofs << "Capacity: " << this->noncombustion_ptr_vec[i]->capacity_kW << " kW \n";
360
361     if (this->noncombustion_ptr_vec[i]->type == NoncombustionType :: HYDRO) {
362         ofs << "Reservoir Capacity: " <<
363             ((Hydro*) (this->noncombustion_ptr_vec[i]))->reservoir_capacity_m3 <<
364             " m3 \n";
365     }
366
367     ofs << "\n";
368 }
369
370 ofs << "\n-----\n\n";
371
372 // 3.7. Renewable

```

```

373 ofs << "## Renewable Assets\n";
374 ofs << "\n";
375
376 for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
377     ofs << "Asset Index: " << i << " \n";
378     ofs << "Type: " << this->renewable_ptr_vec[i]->type_str << " \n";
379     ofs << "Capacity: " << this->renewable_ptr_vec[i]->capacity_kW << " kW \n";
380     ofs << "\n";
381 }
382
383 ofs << "\n-----\n\n";
384
385 // 3.8. Storage
386 ofs << "## Storage Assets\n";
387 ofs << "\n";
388
389 for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
390     ofs << "Asset Index: " << i << " \n";
391     ofs << "Type: " << this->storage_ptr_vec[i]->type_str << " \n";
392     ofs << "Power Capacity: " << this->storage_ptr_vec[i]->power_capacity_kW
393         << " kW \n";
394     ofs << "Energy Capacity: " << this->storage_ptr_vec[i]->energy_capacity_kWh
395         << " kWh \n";
396     ofs << "\n";
397 }
398
399 ofs << "\n-----\n\n";
400
401 // 3.9. Model Results
402 ofs << "## Results\n";
403 ofs << "\n";
404
405 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
406 ofs << "\n";
407
408 ofs << "Total Dispatch + Discharge: " << this->total_dispatch_discharge_kWh
409     << " kWh \n";
410
411 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
412     << " per kWh dispatched/discharged \n";
413 ofs << "\n";
414
415 ofs << "Total Fuel Consumed: " << this->total_fuel_consumed_L << " L "
416     << "(Annual Average: " <<
417         this->total_fuel_consumed_L / this->electrical_load.n_years
418     << " L/yr) \n";
419 ofs << "\n";
420
421 ofs << "Total Carbon Dioxide (CO2) Emissions: " <<
422     this->total_emissions.CO2_kg << " kg "
423     << "(Annual Average: " <<
424         this->total_emissions.CO2_kg / this->electrical_load.n_years
425     << " kg/yr) \n";
426
427 ofs << "Total Carbon Monoxide (CO) Emissions: " <<
428     this->total_emissions.CO_kg << " kg "
429     << "(Annual Average: " <<
430         this->total_emissions.CO_kg / this->electrical_load.n_years
431     << " kg/yr) \n";
432
433 ofs << "Total Nitrogen Oxides (NOx) Emissions: " <<
434     this->total_emissions.NOx_kg << " kg "
435     << "(Annual Average: " <<
436         this->total_emissions.NOx_kg / this->electrical_load.n_years
437     << " kg/yr) \n";
438
439 ofs << "Total Sulfur Oxides (SOx) Emissions: " <<
440     this->total_emissions.SOx_kg << " kg "
441     << "(Annual Average: " <<
442         this->total_emissions.SOx_kg / this->electrical_load.n_years
443     << " kg/yr) \n";
444
445 ofs << "Total Methane (CH4) Emissions: " << this->total_emissions.CH4_kg << " kg "
446     << "(Annual Average: " <<
447         this->total_emissions.CH4_kg / this->electrical_load.n_years
448     << " kg/yr) \n";
449
450 ofs << "Total Particulate Matter (PM) Emissions: " <<
451     this->total_emissions.PM_kg << " kg "
452     << "(Annual Average: " <<
453         this->total_emissions.PM_kg / this->electrical_load.n_years
454     << " kg/yr) \n";
455
456 ofs << "\n-----\n\n";
457
458 ofs.close();
459 return;

```

```
460 }    /* __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.

```
480 {
481     // 1. create filestream
482     write_path += "Model/time_series_results.csv";
483     std::ofstream ofs;
484     ofs.open(write_path, std::ofstream::out);
485
486     // 2. write time series results header (comma separated value)
487     ofs << "Time (since start of data) [hrs],";
488     ofs << "Electrical Load [kW],";
489     ofs << "Net Load [kW],";
490     ofs << "Missed Load [kW],";
491
492     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
493         ofs << this->renewable_ptr_vec[i]->capacity_kW << " kW "
494             << this->renewable_ptr_vec[i]->type_str << " Dispatch [kW],";
495     }
496
497     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
498         ofs << this->storage_ptr_vec[i]->power_capacity_kW << " kW "
499             << this->storage_ptr_vec[i]->energy_capacity_kWh << " kWh "
500             << this->storage_ptr_vec[i]->type_str << " Discharge [kW],";
501     }
502
503     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
504         ofs << this->noncombustion_ptr_vec[i]->capacity_kW << " kW "
505             << this->noncombustion_ptr_vec[i]->type_str << " Dispatch [kW],";
506     }
507
508     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
509         ofs << this->combustion_ptr_vec[i]->capacity_kW << " kW "
510             << this->combustion_ptr_vec[i]->type_str << " Dispatch [kW],";
511     }
512
513     ofs << "\n";
514
515     // 3. write time series results values (comma separated value)
516     for (int i = 0; i < max_lines; i++) {
517         // 3.1. load values
518         ofs << this->electrical_load.time_vec_hrs[i] << ",";
519         ofs << this->electrical_load.load_vec_kW[i] << ",";
520         ofs << this->controller.net_load_vec_kW[i] << ",";
521         ofs << this->controller.missed_load_vec_kW[i] << ",";
522
523         // 3.2. asset-wise dispatch/discharge
524         for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
525             ofs << this->renewable_ptr_vec[j]->dispatch_vec_kW[i] << ",";
526         }
527
528         for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
529             ofs << this->storage_ptr_vec[j]->discharging_power_vec_kW[i] << ",";
530         }
531
532         for (size_t j = 0; j < this->noncombustion_ptr_vec.size(); j++) {
533             ofs << this->noncombustion_ptr_vec[j]->dispatch_vec_kW[i] << ",";
534         }
535
536         for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
```

```

537         ofs << this->combustion_ptr_vec[j]->dispatch_vec_kW[i] << ", ";
538     }
539
540     ofs << "\n";
541 }
542
543 ofs.close();
544 return;
545 } /* __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.
----------------------	---

```

617 {
618     Combustion* diesel_ptr = new Diesel(
619         this->electrical_load.n_points,
620         this->electrical_load.n_years,
621         diesel_inputs
622     );
623
624     this->combustion_ptr_vec.push_back(diesel_ptr);
625
626     return;
627 } /* 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.
---------------------	--

```

720 {
721     Noncombustion* hydro_ptr = new Hydro(
722         this->electrical_load.n_points,
723         this->electrical_load.n_years,
724         hydro_inputs
725     );
726
727     this->noncombustion_ptr_vec.push_back(hydro_ptr);
728
729     return;
730 } /* addHydro() */

```

4.15.3.10 addLilOn()

```
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.
---------------------	--

```
855 {
856     Storage* liion_ptr = new LiIon(
857         this->electrical_load.n_points,
858         this->electrical_load.n_years,
859         liion_inputs
860     );
861     this->storage_ptr_vec.push_back(liion_ptr);
862
863     return;
864 } /* 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.

```
656 {
657     resources.addResource(
658         noncombustion_type,
659         path_2_resource_data,
660         resource_key,
661         &(this->electrical_load)
662     );
663     return;
664 } /* 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.

```

694 {
695     resources.addResource(
696         renewable_type,
697         path_2_resource_data,
698         resource_key,
699         &(this->electrical_load)
700     );
701
702     return;
703 } /* 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.
---------------------	--

```

747 {
748     Renewable* solar_ptr = new Solar(
749         this->electrical_load.n_points,
750         this->electrical_load.n_years,
751         solar_inputs
752     );
753
754     this->renewable_ptr_vec.push_back(solar_ptr);
755
756     return;
757 } /* 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.
---------------------	--

```

774 {
775     Renewable* tidal_ptr = new Tidal(
776         this->electrical_load.n_points,
777         this->electrical_load.n_years,
778         tidal_inputs

```



```

779     );
780
781     this->renewable_ptr_vec.push_back(tidal_ptr);
782
783     return;
784 } /* 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.
--------------------	---

```

801 {
802     Renewable* wave_ptr = new Wave(
803         this->electrical_load.n_points,
804         this->electrical_load.n_years,
805         wave_inputs
806     );
807
808     this->renewable_ptr_vec.push_back(wave_ptr);
809
810     return;
811 } /* 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.
--------------------	---

```

828 {
829     Renewable* wind_ptr = new Wind(
830         this->electrical_load.n_points,
831         this->electrical_load.n_years,
832         wind_inputs
833     );
834
835     this->renewable_ptr_vec.push_back(wind_ptr);
836
837     return;
838 } /* addWind() */

```

4.15.3.17 clear()

```

void Model::clear (
    void )

```

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

```

979 {
980     // 1. reset
981     this->reset();
982
983     // 2. clear components
984     controller.clear();
985     electrical_load.clear();
986     resources.clear();
987
988     return;
989 } /* 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.

```

924 {
925     // 1. clear combustion_ptr_vec
926     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
927         delete this->combustion_ptr_vec[i];
928     }
929     this->combustion_ptr_vec.clear();
930
931     // 2. clear noncombustion_ptr_vec
932     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
933         delete this->noncombustion_ptr_vec[i];
934     }
935     this->noncombustion_ptr_vec.clear();
936
937     // 3. clear renewable_ptr_vec
938     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
939         delete this->renewable_ptr_vec[i];
940     }
941     this->renewable_ptr_vec.clear();
942
943     // 4. clear storage_ptr_vec
944     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
945         delete this->storage_ptr_vec[i];
946     }
947     this->storage_ptr_vec.clear();
948
949     // 5. reset attributes
950     this->total_fuel_consumed_L = 0;
951
952     this->total_emissions.CO2_kg = 0;
953     this->total_emissions.CO_kg = 0;
954     this->total_emissions.NOx_kg = 0;
955     this->total_emissions.SOx_kg = 0;
956     this->total_emissions.CH4_kg = 0;
957     this->total_emissions.PM_kg = 0;
958
959     this->net_present_cost = 0;
960     this->total_dispatch_discharge_kWh = 0;
961     this->levellized_cost_of_energy_kWh = 0;
962
963     return;
964 } /* reset() */

```

4.15.3.19 run()

```

void Model::run (
    void )

```

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

```

880 {
881     // 1. init Controller
882     this->controller.init(
883         &(this->electrical_load),
884         &(this->renewable_ptr_vec),
885         &(this->resources),
886         &(this->combustion_ptr_vec)
887     );
888
889     // 2. apply dispatch control
890     this->controller.applyDispatchControl(
891         &(this->electrical_load),
892         &(this->resources),
893         &(this->combustion_ptr_vec),
894         &(this->noncombustion_ptr_vec),
895         &(this->renewable_ptr_vec),
896         &(this->storage_ptr_vec)
897     );
898
899     // 3. compute total fuel consumption and emissions
900     this->__computeFuelAndEmissions();
901
902     // 4. compute key economic metrics
903     this->__computeEconomics();
904
905     return;
906 } /* 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.

```

1017 {
1018     // 1. handle sentinel
1019     if (max_lines < 0) {
1020         max_lines = this->electrical_load.n_points;
1021     }
1022
1023     // 2. check for pre-existing, warn (and remove), then create
1024     if (write_path.back() != '/') {
1025         write_path += '/';
1026     }
1027
1028     if (std::filesystem::is_directory(write_path)) {
1029         std::string warning_str = "WARNING: Model::writeResults(): ";
1030         warning_str += write_path;
1031         warning_str += " already exists, contents will be overwritten!";
1032
1033         std::cout << warning_str << std::endl;
1034
1035         std::filesystem::remove_all(write_path);
1036     }
1037
1038     std::filesystem::create_directory(write_path);
1039
1040     // 3. write summary
1041     this->__writeSummary(write_path);
1042
1043     // 4. write time series
1044     if (max_lines > this->electrical_load.n_points) {

```

```

1045         max_lines = this->electrical_load.n_points;
1046     }
1047
1048     if (max_lines > 0) {
1049         this->__writeTimeSeries(write_path, max_lines);
1050     }
1051
1052     // 5. call out to Combustion :: writeResults()
1053     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
1054         this->combustion_ptr_vec[i]->writeResults(
1055             write_path,
1056             &(this->electrical_load.time_vec_hrs),
1057             i,
1058             max_lines
1059         );
1060     }
1061
1062     // 6. call out to Noncombustion :: writeResults()
1063     for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
1064         this->noncombustion_ptr_vec[i]->writeResults(
1065             write_path,
1066             &(this->electrical_load.time_vec_hrs),
1067             i,
1068             max_lines
1069         );
1070     }
1071
1072     // 7. call out to Renewable :: writeResults()
1073     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
1074         this->renewable_ptr_vec[i]->writeResults(
1075             write_path,
1076             &(this->electrical_load.time_vec_hrs),
1077             &(this->resources.resource_map_1D),
1078             &(this->resources.resource_map_2D),
1079             i,
1080             max_lines
1081         );
1082     }
1083
1084     // 8. call out to Storage :: writeResults()
1085     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
1086         this->storage_ptr_vec[i]->writeResults(
1087             write_path,
1088             &(this->electrical_load.time_vec_hrs),
1089             i,
1090             max_lines
1091         );
1092     }
1093
1094     return;
1095 } /* 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.

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

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

4.16 ModelInputs Struct Reference

A structure which bundles the necessary inputs for the [Model](#) constructor. Provides default values for every necessary input (except `path_2_electrical_load_time_series`, for which a valid input must be provided).

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

Public Attributes

- `std::string path_2_electrical_load_time_series = ""`
A string defining the path (either relative or absolute) to the given electrical load time series.
- `ControlMode control_mode = ControlMode :: LOAD_FOLLOWING`
The control mode to be applied by the [Controller](#) object.

4.16.1 Detailed Description

A structure which bundles the necessary inputs for the [Model](#) constructor. Provides default values for every necessary input (except `path_2_electrical_load_time_series`, for which a valid input must be provided).

4.16.2 Member Data Documentation

4.16.2.1 `control_mode`

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

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

4.16.2.2 `path_2_electrical_load_time_series`

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

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

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

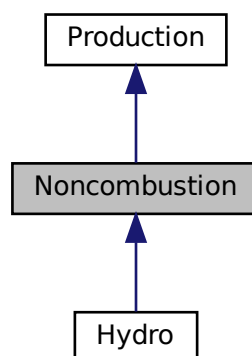
- [header/Model.h](#)

4.17 Noncombustion Class Reference

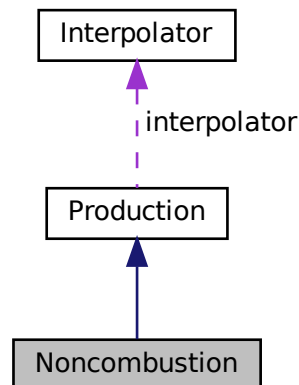
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

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

Inheritance diagram for Noncombustion:



Collaboration diagram for Noncombustion:



Public Member Functions

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

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

4.17.2.2 [Noncombustion\(\)](#) [2/2]

```
Noncombustion::Noncombustion (
    int n_points,
    double n_years,
    NoncombustionInputs noncombustion_inputs )
```

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.

```
133 :
134 Production(
135     n_points,
136     n_years,
137     noncombustion_inputs.production_inputs
138 )
139 {
```

```

140     // 1. check inputs
141     this->__checkInputs(noncombustion_inputs);
142
143     // 2. set attributes
144     //...
145
146     // 3. construction print
147     if (this->print_flag) {
148         std::cout << "Noncombustion object constructed at " << this << std::endl;
149     }
150
151     return;
152 } /* Noncombustion() */

```

4.17.2.3 ~Noncombustion()

```

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

```

Destructor for the [Noncombustion](#) class.

```

343 {
344     // 1. destruction print
345     if (this->print_flag) {
346         std::cout << "Noncombustion object at " << this << " destroyed" << std::endl;
347     }
348
349     return;
350 } /* ~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.
-----------------------------	--

```

40 {
41     //...
42
43     return;
44 } /* __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.

```
67 {
68     if (this->is_running) {
69         // handle stopping
70         if (production_kW <= 0) {
71             this->is_running = false;
72         }
73     }
74
75     else {
76         // handle starting
77         if (production_kW > 0) {
78             this->is_running = true;
79             this->n_starts++;
80         }
81     }
82
83     return;
84 } /* __handleStartStop() */
```

4.17.3.3 __writeSummary()

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

Reimplemented in [Hydro](#).

```
70 {return;}
```

4.17.3.4 __writeTimeSeries()

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

Reimplemented in [Hydro](#).

```
75 {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](#).

```

238 {
239     // 1. handle start/stop
240     this->__handleStartStop(timestep, dt_hrs, production_kW);
241
242     // 2. invoke base class method
243     load_kW = Production::commit(
244         timestep,
245         dt_hrs,
246         production_kW,
247         load_kW
248     );
249
250
251     //...
252
253     return load_kW;
254 } /* commit() */

```

4.17.3.6 commit() [2/2]

```

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

```

Reimplemented in [Hydro](#).

```

96 {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](#).

```

197 {
198     // 1. invoke base class method
199     Production::computeEconomics(time_vec_hrs_ptr);
200
201     return;
202 } /* 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](#).

```

170 {
171     // 1. reset attributes
172     //...
173
174     // 2. invoke base class method
175     Production::handleReplacement(timestep);
176
177     return;
178 } /* __handleReplacement() */

```

4.17.3.9 requestProductionkW() [1/2]

```

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

```

4.17.3.10 requestProductionkW() [2/2]

```

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

```

Reimplemented in [Hydro](#).

```

93 {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.

```
290 {
291     // 1. handle sentinel
292     if (max_lines < 0) {
293         max_lines = this->n_points;
294     }
295
296     // 2. create subdirectories
297     write_path += "Production/";
298     if (not std::filesystem::is_directory(write_path)) {
299         std::filesystem::create_directory(write_path);
300     }
301
302     write_path += "Noncombustion/";
303     if (not std::filesystem::is_directory(write_path)) {
304         std::filesystem::create_directory(write_path);
305     }
306
307     write_path += this->type_str;
308     write_path += "_";
309     write_path += std::to_string(int(ceil(this->capacity_kW)));
310     write_path += "kW_idx";
311     write_path += std::to_string(combustion_index);
312     write_path += "/";
313     std::filesystem::create_directory(write_path);
314
315     // 3. write summary
316     this->__writeSummary(write_path);
317
318     // 4. write time series
319     if (max_lines > this->n_points) {
320         max_lines = this->n_points;
321     }
322
323     if (max_lines > 0) {
324         this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
325     }
326
327     return;
328 } /* 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:

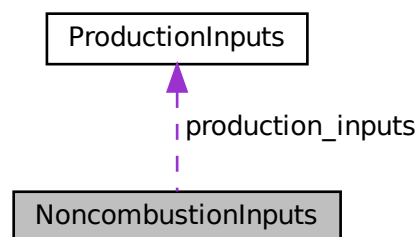
- [header/Production/Noncombustion/Noncombustion.h](#)
- [source/Production/Noncombustion/Noncombustion.cpp](#)

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:

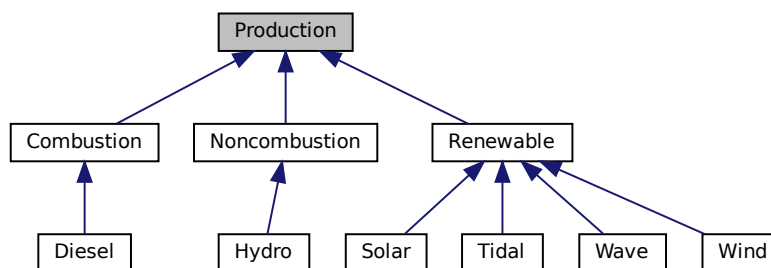
- `header/Production/Noncombustion/Noncombustion.h`

4.19 Production Class Reference

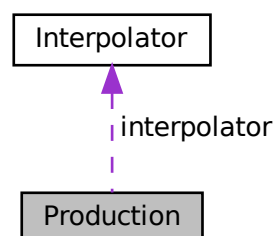
The base class of the `Production` hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

```
#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](#))
Constructor (intended) for the [Production](#) class.
- virtual void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeRealDiscountAnnual](#) (double, double)
Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual [~Production](#) (void)
Destructor for the [Production](#) class.

Public Attributes

- [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).
- 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::vector< bool > [is_running_vec](#)
A boolean vector for tracking if the asset is running at a particular point in time.
- std::vector< double > [production_vec_kW](#)
A vector of production [kW] at each point in the modelling time series.
- std::vector< double > [dispatch_vec_kW](#)
A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.
- std::vector< double > [storage_vec_kW](#)
A vector of storage [kW] at each point in the modelling time series. [Storage](#) is the amount of production that is sent to storage.
- std::vector< double > [curtailment_vec_kW](#)
A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.
- std::vector< double > [capital_cost_vec](#)
A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > [operation_maintenance_cost_vec](#)
A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- void [__checkInputs](#) (int, double, [ProductionInputs](#))
Helper method to check inputs to the [Production](#) constructor.

4.19.1 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.

```
112 {
113     return;
114 } /* Production() */
```

4.19.2.2 Production() [2/2]

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

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

Parameters

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

```
143 {
144     // 1. check inputs
145     this->__checkInputs(n_points, n_years, production_inputs);
146
147     // 2. set attributes
148     this->print_flag = production_inputs.print_flag;
149     this->is_running = false;
150     this->is_sunk = production_inputs.is_sunk;
151
152     this->n_points = n_points;
153     this->n_starts = 0;
154     this->n_replacements = 0;
155
156     this->n_years = n_years;
157
158     this->running_hours = 0;
159     this->replace_running_hrs = production_inputs.replace_running_hrs;
160
161     this->capacity_kW = production_inputs.capacity_kW;
162
163     this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
164     this->nominal_discount_annual = production_inputs.nominal_discount_annual;
165
166     this->real_discount_annual = this->computeRealDiscountAnnual (
167         production_inputs.nominal_inflation_annual,
168         production_inputs.nominal_discount_annual
169     );
170
171     this->capital_cost = 0;
172     this->operation_maintenance_cost_kWh = 0;
173     this->net_present_cost = 0;
174     this->total_dispatch_kWh = 0;
175     this->levellized_cost_of_energy_kWh = 0;
176
177     this->is_running_vec.resize(this->n_points, 0);
178
179     this->production_vec_kW.resize(this->n_points, 0);
180     this->dispatch_vec_kW.resize(this->n_points, 0);
181     this->storage_vec_kW.resize(this->n_points, 0);
182     this->curtailment_vec_kW.resize(this->n_points, 0);
183
184     this->capital_cost_vec.resize(this->n_points, 0);
185     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
186 }
```

```

187     // 3. construction print
188     if (this->print_flag) {
189         std::cout << "Production object constructed at " << this << std::endl;
190     }
191
192     return;
193 } /* Production() */

```

4.19.2.3 ~Production()

```

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

```

Destructor for the [Production](#) class.

```

411 {
412     // 1. destruction print
413     if (this->print_flag) {
414         std::cout << "Production object at " << this << " destroyed" << std::endl;
415     }
416
417     return;
418 } /* ~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

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

```

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

```

```

65         throw std::invalid_argument(error_str);
66     }
67
68     // 3. check capacity_kW
69     if (production_inputs.capacity_kW <= 0) {
70         std::string error_str = "ERROR: Production(): ";
71         error_str += "ProductionInputs::capacity_kW must be > 0";
72
73         #ifdef _WIN32
74             std::cout << error_str << std::endl;
75         #endif
76
77         throw std::invalid_argument(error_str);
78     }
79
80     // 4. check replace_running_hrs
81     if (production_inputs.replace_running_hrs <= 0) {
82         std::string error_str = "ERROR: Production(): ";
83         error_str += "ProductionInputs::replace_running_hrs must be > 0";
84
85         #ifdef _WIN32
86             std::cout << error_str << std::endl;
87         #endif
88
89         throw std::invalid_argument(error_str);
90     }
91
92     return;
93 } /* __checkInputs() */

```

4.19.3.2 commit()

```

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

```

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

Parameters

<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](#).

```

352 {
353     // 1. record production
354     this->production_vec_kW[timestep] = production_kW;
355
356     // 2. compute and record dispatch and curtailment
357     double dispatch_kW = 0;
358     double curtailment_kW = 0;
359
360     if (production_kW > load_kW) {
361         dispatch_kW = load_kW;
362         curtailment_kW = production_kW - dispatch_kW;
363     }

```

```

364
365     else {
366         dispatch_kW = production_kW;
367     }
368
369     this->dispatch_vec_kW[timestep] = dispatch_kW;
370     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
371     this->curtailment_vec_kW[timestep] = curtailment_kW;
372
373     // 3. update load
374     load_kW -= dispatch_kW;
375
376     // 4. update and log running attributes
377     if (this->is_running) {
378         // 4.1. log running state, running hours
379         this->is_running_vec[timestep] = this->is_running;
380         this->running_hours += dt_hrs;
381
382         // 4.2. incur operation and maintenance costs
383         double produced_kWh = production_kW * dt_hrs;
384
385         double operation_maintenance_cost =
386             this->operation_maintenance_cost_kWh * produced_kWh;
387         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
388     }
389
390     // 5. trigger replacement, if applicable
391     if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
392         this->handleReplacement(timestep);
393     }
394
395     return load_kW;
396 } /* commit() */

```

4.19.3.3 computeEconomics()

```

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

```

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

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

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

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

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

Parameters

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

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

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

```

281 {
282     // 1. compute net present cost
283     double t_hrs = 0;
284     double real_discount_scalar = 0;
285
286     for (int i = 0; i < this->n_points; i++) {
287         t_hrs = time_vec_hrs_ptr->at(i);
288
289         real_discount_scalar = 1.0 / pow(
290             1 + this->real_discount_annual,
291             t_hrs / 8760
292         );

```

```

293
294         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
295
296         this->net_present_cost +=
297             real_discount_scalar * this->operation_maintenance_cost_vec[i];
298     }
299
300     // assuming 8,760 hours per year
301     double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
302
303     double capital_recovery_factor =
304         (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
305         (pow(1 + this->real_discount_annual, n_years) - 1);
306
307     double total_annualized_cost = capital_recovery_factor *
308         this->net_present_cost;
309
310     this->levellized_cost_of_energy_kWh =
311         (n_years * total_annualized_cost) /
312         this->total_dispatch_kWh;
313
314     return;
315 } /* computeEconomics() */

```

4.19.3.4 computeRealDiscountAnnual()

```

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

```

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

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

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

Parameters

<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.

```

254 {
255     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
256     real_discount_annual /= 1 + nominal_inflation_annual;
257
258     return real_discount_annual;
259 } /* __computeRealDiscountAnnual() */

```

4.19.3.5 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](#).

```

211 {
212     // 1. reset attributes
213     this->is_running = false;
214
215     // 2. log replacement
216     this->n_replacements++;
217
218     // 3. incur capital cost in timestep
219     this->capital_cost_vec[timestep] = this->capital_cost;
220
221     return;
222 } /* __handleReplacement() */

```

4.19.4 Member Data Documentation

4.19.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.19.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.19.4.3 capital_cost_vec

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

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

4.19.4.4 curtailment_vec_kW

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

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

4.19.4.5 dispatch_vec_kW

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

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

4.19.4.6 interpolator

```
Interpolator Production::interpolator
```

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

4.19.4.7 is_running

```
bool Production::is_running
```

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

4.19.4.8 is_running_vec

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

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

4.19.4.9 is_sunk

```
bool Production::is_sunk
```

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

4.19.4.10 levlized_cost_of_energy_kWh

```
double Production::levlized_cost_of_energy_kWh
```

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

4.19.4.11 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.19.4.12 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.19.4.13 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.19.4.14 n_years

```
double Production::n_years
```

The number of years being modelled.

4.19.4.15 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.19.4.16 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

4.19.4.17 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

4.19.4.18 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.

4.19.4.19 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).

4.19.4.20 print_flag

```
bool Production::print_flag
```

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

4.19.4.21 production_vec_kW

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

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

4.19.4.22 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.

4.19.4.23 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.19.4.24 running_hours

```
double Production::running_hours
```

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

4.19.4.25 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.

4.19.4.26 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

4.19.4.27 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.

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 print_flag

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

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

4.20.2.6 replace_running_hrs

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

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

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

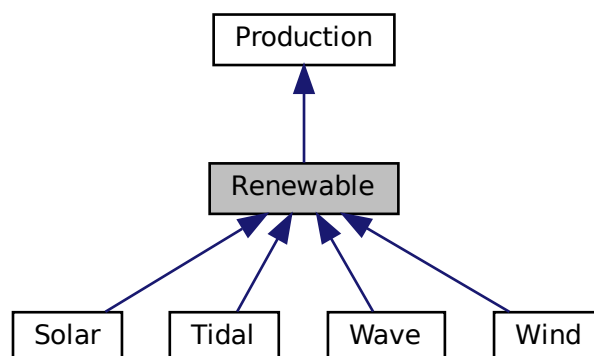
- header/Production/[Production.h](#)

4.21 Renewable Class Reference

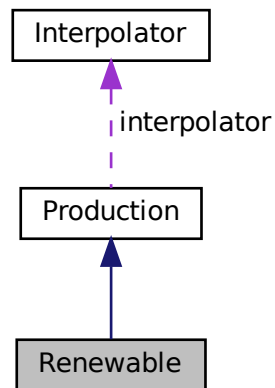
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

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

Public Attributes

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

Private Member Functions

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

4.21.1 Detailed Description

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

4.21.2 Constructor & Destructor Documentation

4.21.2.1 [Renewable\(\)](#) [1/2]

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

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

```
100 {
101     //...
102
103     return;
104 } /* Renewable() */
```

4.21.2.2 [Renewable\(\)](#) [2/2]

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

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.

```
132 :
133 Production(
134     n_points,
```



```

135     n_years,
136     renewable_inputs.production_inputs
137 )
138 {
139     // 1. check inputs
140     this->__checkInputs(renewable_inputs);
141
142     // 2. set attributes
143     //...
144
145     // 3. construction print
146     if (this->print_flag) {
147         std::cout << "Renewable object constructed at " << this << std::endl;
148     }
149
150     return;
151 } /* Renewable() */

```

4.21.2.3 ~Renewable()

```

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

```

Destructor for the [Renewable](#) class.

```

354 {
355     // 1. destruction print
356     if (this->print_flag) {
357         std::cout << "Renewable object at " << this << " destroyed" << std::endl;
358     }
359
360     return;
361 } /* ~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.

```

37 {
38     //...
39
40     return;
41 } /* __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.

```
64 {
65     if (this->is_running) {
66         // handle stopping
67         if (production_kW <= 0) {
68             this->is_running = false;
69         }
70     }
71
72     else {
73         // handle starting
74         if (production_kW > 0) {
75             this->is_running = true;
76             this->n_starts++;
77         }
78     }
79
80     return;
81 } /* __handleStartStop() */
```

4.21.3.3 __writeSummary()

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

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

```
72 {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](#).

```
79     {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](#).

```

235 {
236     // 1. handle start/stop
237     this->__handleStartStop(timestep, dt_hrs, production_kW);
238
239     // 2. invoke base class method
240     load_kW = Production::commit(
241         timestep,
242         dt_hrs,
243         production_kW,
244         load_kW
245     );
246
247
248     //...
249
250     return load_kW;
251 } /* 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](#).

```

194 {
195     // 1. invoke base class method
196     Production::computeEconomics(time_vec_hrs_ptr);
197
198     return;
199 } /* computeEconomics() */

```

4.21.3.7 computeProductionkW() [1/2]

```

virtual double Renewable::computeProductionkW (
    int ,

```

```
double ,
double ) [inline], [virtual]
```

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

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

4.21.3.8 computeProductionkW() [2/2]

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

Reimplemented in [Wave](#).

```
97 {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](#).

```
169 {
170     // 1. reset attributes
171     //...
172
173     // 2. invoke base class method
174     Production :: handleReplacement(timestep);
175
176     return;
177 } /* __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.

```

295 {
296     // 1. handle sentinel
297     if (max_lines < 0) {
298         max_lines = this->n_points;
299     }
300
301     // 2. create subdirectories
302     write_path += "Production/";
303     if (not std::filesystem::is_directory(write_path)) {
304         std::filesystem::create_directory(write_path);
305     }
306
307     write_path += "Renewable/";
308     if (not std::filesystem::is_directory(write_path)) {
309         std::filesystem::create_directory(write_path);
310     }
311
312     write_path += this->type_str;
313     write_path += "_";
314     write_path += std::to_string(int(ceil(this->capacity_kw)));
315     write_path += "kW_idx";
316     write_path += std::to_string(renewable_index);
317     write_path += "/";
318     std::filesystem::create_directory(write_path);
319
320     // 3. write summary
321     this->__writeSummary(write_path);
322
323     // 4. write time series
324     if (max_lines > this->n_points) {
325         max_lines = this->n_points;
326     }
327
328     if (max_lines > 0) {
329         this->__writeTimeSeries(
330             write_path,
331             time_vec_hrs_ptr,
332             resource_map_1D_ptr,
333             resource_map_2D_ptr,
334             max_lines
335         );
336     }
337
338     return;
339 } /* writeResults() */

```

4.21.4 Member Data Documentation

4.21.4.1 resource_key

```
int Renewable::resource_key
```

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

4.21.4.2 type

`RenewableType` `Renewable::type`

The type (`RenewableType`) of the asset.

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

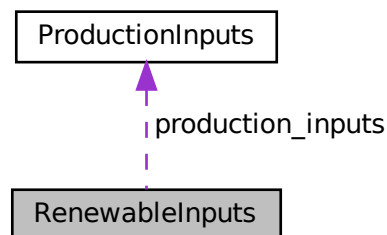
- `header/Production/Renewable/Renewable.h`
- `source/Production/Renewable/Renewable.cpp`

4.22 RenewableInputs Struct Reference

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

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

Collaboration diagram for `RenewableInputs`:



Public Attributes

- `ProductionInputs production_inputs`
An encapsulated `ProductionInputs` instance.

4.22.1 Detailed Description

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

4.22.2 Member Data Documentation

4.22.2.1 production_inputs

`ProductionInputs RenewableInputs::production_inputs`

An encapsulated `ProductionInputs` instance.

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

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

4.23 Resources Class Reference

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

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

Public Member Functions

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

Public Attributes

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

Private Member Functions

- void `__checkResourceKey1D` (int, [RenewableType](#))
Helper method to check if given resource key (1D) is already in use.
- void `__checkResourceKey2D` (int, [RenewableType](#))
Helper method to check if given resource key (2D) is already in use.
- void `__checkResourceKey1D` (int, [NoncombustionType](#))
Helper method to check if given resource key (1D) is already in use.
- void `__checkTimePoint` (double, double, std::string, [ElectricalLoad](#) *)
Helper method to check received time point against expected time point.
- void `__throwLengthError` (std::string, [ElectricalLoad](#) *)
Helper method to throw data length error.
- 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.

```
727 {
728     return;
729 } /* Resources() */
```

4.23.2.2 ~Resources()

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

Destructor for the [Resources](#) class.

```
939 {
940     this->clear();
941     return;
942 } /* ~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 .

```
114 {
115     if (this->resource_map_1D.count(resource_key) > 0) {
116         std::string error_str = "ERROR: Resources::addResource(";
117
118         switch (noncombustion_type) {
119             case (NoncombustionType :: HYDRO): {
120                 error_str += "HYDRO): ";
121
122                 break;
123             }
124
125             default: {
126                 error_str += "UNDEFINED_TYPE): ";
127
128                 break;
129             }
130         }
131
132         error_str += "resource key (1D) ";
133         error_str += std::to_string(resource_key);
134         error_str += " is already in use";
135
136         #ifdef _WIN32
137             std::cout << error_str << std::endl;
138         #endif
139
140         throw std::invalid_argument(error_str);
141     }
142
143     return;
144 } /* __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 .

```

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

```

167 {
168     if (this->resource_map_2D.count(resource_key) > 0) {
169         std::string error_str = "ERROR: Resources::addResource(";
170
171         switch (renewable_type) {
172             case (RenewableType :: WAVE): {
173                 error_str += "WAVE): ";
174
175                 break;
176             }
177
178             default: {
179                 error_str += "UNDEFINED_TYPE): ";
180
181                 break;
182             }
183         }
184     }

```

```

185         error_str += "resource key (2D) ";
186         error_str += std::to_string(resource_key);
187         error_str += " is already in use";
188
189         #ifdef _WIN32
190             std::cout << error_str << std::endl;
191         #endif
192
193         throw std::invalid_argument(error_str);
194     }
195
196     return;
197 } /* __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.

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.

```

232 {
233     if (time_received_hrs != time_expected_hrs) {
234         std::string error_str = "ERROR: Resources::addResource(): ";
235         error_str += "the given resource time series at ";
236         error_str += path_2_resource_data;
237         error_str += " does not align with the ";
238         error_str += "previously given electrical load time series at ";
239         error_str += electrical_load_ptr->path_2_electrical_load_time_series;
240
241         #ifdef _WIN32
242             std::cout << error_str << std::endl;
243         #endif
244
245         throw std::runtime_error(error_str);
246     }
247
248     return;
249 } /* __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.

```

320 {
321     // 1. init CSV reader, record path and type
322     io::CSVReader<2> CSV(path_2_resource_data);
323
324     CSV.read_header(
325         io::ignore_extra_column,
326         "Time (since start of data) [hrs]",
327         "Hydro Inflow [m3/hr]"
328     );
329
330     this->path_map_1D.insert(
331         std::pair<int, std::string>(resource_key, path_2_resource_data)
332     );
333
334     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "HYDRO"));
335
336     // 2. init map element
337     this->resource_map_1D.insert(
338         std::pair<int, std::vector<double>>(resource_key, {})
339     );
340     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
341
342
343     // 3. read in resource data, check against time series (point-wise and length)
344     int n_points = 0;
345     double time_hrs = 0;
346     double time_expected_hrs = 0;
347     double hydro_resource_m3hr = 0;
348
349     while (CSV.read_row(time_hrs, hydro_resource_m3hr)) {
350         if (n_points > electrical_load_ptr->n_points) {
351             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
352         }
353
354         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
355         this->__checkTimePoint(
356             time_hrs,
357             time_expected_hrs,
358             path_2_resource_data,
359             electrical_load_ptr
360         );
361
362         this->resource_map_1D[resource_key][n_points] = hydro_resource_m3hr;
363
364         n_points++;
365     }
366
367     // 4. check data length
368     if (n_points != electrical_load_ptr->n_points) {
369         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
370     }
371
372     return;
373 } /* __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.

```

403 {
404     // 1. init CSV reader, record path and type
405     io::CSVReader<2> CSV(path_2_resource_data);
406
407     CSV.read_header(
408         io::ignore_extra_column,
409         "Time (since start of data) [hrs]",
410         "Solar GHI [kW/m2]"
411     );
412
413     this->path_map_1D.insert(
414         std::pair<int, std::string>(resource_key, path_2_resource_data)
415     );
416
417     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));
418
419     // 2. init map element
420     this->resource_map_1D.insert(
421         std::pair<int, std::vector<double>>(resource_key, {})
422     );
423     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
424
425
426     // 3. read in resource data, check against time series (point-wise and length)
427     int n_points = 0;
428     double time_hrs = 0;
429     double time_expected_hrs = 0;
430     double solar_resource_kWm2 = 0;
431
432     while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
433         if (n_points > electrical_load_ptr->n_points) {
434             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
435         }
436
437         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
438         this->__checkTimePoint(
439             time_hrs,
440             time_expected_hrs,
441             path_2_resource_data,
442             electrical_load_ptr
443         );
444
445         this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
446
447         n_points++;
448     }
449
450     // 4. check data length
451     if (n_points != electrical_load_ptr->n_points) {
452         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
453     }
454
455     return;
456 } /* __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.

```

486 {
487     // 1. init CSV reader, record path and type
488     io::CSVReader<2> CSV(path_2_resource_data);
489
490     CSV.read_header(
491         io::ignore_extra_column,
492         "Time (since start of data) [hrs]",
493         "Tidal Speed (hub depth) [m/s]"
494     );
495
496     this->path_map_1D.insert(
497         std::pair<int, std::string>(resource_key, path_2_resource_data)
498     );
499
500     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));
501
502     // 2. init map element
503     this->resource_map_1D.insert(
504         std::pair<int, std::vector<double>>(resource_key, {})
505     );
506     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
507
508     // 3. read in resource data, check against time series (point-wise and length)
509     int n_points = 0;
510     double time_hrs = 0;
511     double time_expected_hrs = 0;
512     double tidal_resource_ms = 0;
513
514     while (CSV.read_row(time_hrs, tidal_resource_ms)) {
515         if (n_points > electrical_load_ptr->n_points) {
516             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
517         }
518
519         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
520         this->__checkTimePoint(
521             time_hrs,
522             time_expected_hrs,
523             path_2_resource_data,
524             electrical_load_ptr
525         );
526
527         this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;
528         n_points++;
529     }
530
531     // 4. check data length
532     if (n_points != electrical_load_ptr->n_points) {
533         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
534     }
535
536     return;
537 }
538 /* __readTidalResource() */
539 }

```

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.

```

569 {
570     // 1. init CSV reader, record path and type
571     io::CSVReader<3> CSV(path_2_resource_data);
572
573     CSV.read_header(
574         io::ignore_extra_column,
575         "Time (since start of data) [hrs]",
576         "Significant Wave Height [m]",
577         "Energy Period [s]"
578     );
579
580     this->path_map_2D.insert(
581         std::pair<int, std::string>(resource_key, path_2_resource_data)
582     );
583
584     this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));
585
586     // 2. init map element
587     this->resource_map_2D.insert(
588         std::pair<int, std::vector<std::vector<double>>>(resource_key, {})
589     );
590     this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});
591
592
593     // 3. read in resource data, check against time series (point-wise and length)
594     int n_points = 0;
595     double time_hrs = 0;
596     double time_expected_hrs = 0;
597     double significant_wave_height_m = 0;
598     double energy_period_s = 0;
599
600     while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
601         if (n_points > electrical_load_ptr->n_points) {
602             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
603         }
604
605         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
606         this->__checkTimePoint(
607             time_hrs,
608             time_expected_hrs,
609             path_2_resource_data,
610             electrical_load_ptr
611         );
612
613         this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
614         this->resource_map_2D[resource_key][n_points][1] = energy_period_s;
615
616         n_points++;
617     }
618
619     // 4. check data length
620     if (n_points != electrical_load_ptr->n_points) {
621         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
622     }
623
624     return;
625 } /* __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.

```

655 {
656     // 1. init CSV reader, record path and type
657     io::CSVReader<2> CSV(path_2_resource_data);
658
659     CSV.read_header(
660         io::ignore_extra_column,
661         "Time (since start of data) [hrs]",
662         "Wind Speed (hub height) [m/s]"
663     );
664
665     this->path_map_1D.insert(
666         std::pair<int, std::string>(resource_key, path_2_resource_data)
667     );
668
669     this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));
670
671     // 2. init map element
672     this->resource_map_1D.insert(
673         std::pair<int, std::vector<double>>(resource_key, {})
674     );
675     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
676
677
678     // 3. read in resource data, check against time series (point-wise and length)
679     int n_points = 0;
680     double time_hrs = 0;
681     double time_expected_hrs = 0;
682     double wind_resource_ms = 0;
683
684     while (CSV.read_row(time_hrs, wind_resource_ms)) {
685         if (n_points > electrical_load_ptr->n_points) {
686             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
687         }
688
689         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
690         this->__checkTimePoint(
691             time_hrs,
692             time_expected_hrs,
693             path_2_resource_data,
694             electrical_load_ptr
695         );
696
697         this->resource_map_1D[resource_key][n_points] = wind_resource_ms;
698
699         n_points++;
700     }
701
702     // 4. check data length
703     if (n_points != electrical_load_ptr->n_points) {
704         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
705     }
706
707     return;
708 } /* __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.

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.


```

275 {
276     std::string error_str = "ERROR: Resources::addResource(): ";
277     error_str += "the given resource time series at ";
278     error_str += path_2_resource_data;
279     error_str += " is not the same length as the previously given electrical";
280     error_str += " load time series at ";
281     error_str += electrical_load_ptr->path_2_electrical_load_time_series;
282
283     #ifdef _WIN32
284         std::cout << error_str << std::endl;
285     #endif
286
287     throw std::runtime_error(error_str);
288
289     return;
290 } /* __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.

```

766 {
767     switch (noncombustion_type) {
768         case (NoncombustionType :: HYDRO): {
769             this->__checkResourceKey1D(resource_key, noncombustion_type);
770
771             this->__readHydroResource (
772                 path_2_resource_data,
773                 resource_key,
774                 electrical_load_ptr
775             );
776
777             break;
778         }
779
780         default: {
781             std::string error_str = "ERROR: Resources :: addResource(): ";
782             error_str += "noncombustion type ";
783             error_str += std::to_string(noncombustion_type);
784             error_str += " has no associated resource";
785
786             #ifdef _WIN32
787                 std::cout << error_str << std::endl;
788             #endif
789
790             throw std::runtime_error(error_str);
791
792             break;
793         }
794     }
795
796     return;

```

```
797 } /* 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.

```
834 {
835     switch (renewable_type) {
836         case (RenewableType :: SOLAR): {
837             this->__checkResourceKey1D(resource_key, renewable_type);
838
839             this->__readSolarResource(
840                 path_2_resource_data,
841                 resource_key,
842                 electrical_load_ptr
843             );
844
845             break;
846         }
847
848         case (RenewableType :: TIDAL): {
849             this->__checkResourceKey1D(resource_key, renewable_type);
850
851             this->__readTidalResource(
852                 path_2_resource_data,
853                 resource_key,
854                 electrical_load_ptr
855             );
856
857             break;
858         }
859
860         case (RenewableType :: WAVE): {
861             this->__checkResourceKey2D(resource_key, renewable_type);
862
863             this->__readWaveResource(
864                 path_2_resource_data,
865                 resource_key,
866                 electrical_load_ptr
867             );
868
869             break;
870         }
871
872         case (RenewableType :: WIND): {
873             this->__checkResourceKey1D(resource_key, renewable_type);
874
875             this->__readWindResource(
876                 path_2_resource_data,
877                 resource_key,
878                 electrical_load_ptr
879             );
```

```

880
881         break;
882     }
883
884     default: {
885         std::string error_str = "ERROR: Resources :: addResource(: ";
886         error_str += "renewable type ";
887         error_str += std::to_string(renewable_type);
888         error_str += " not recognized";
889
890         #ifdef _WIN32
891             std::cout << error_str << std::endl;
892         #endif
893
894         throw std::runtime_error(error_str);
895
896         break;
897     }
898 }
899
900 return;
901 } /* addResource() */

```

4.23.3.13 clear()

```

void Resources::clear (
    void )

```

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

```

915 {
916     this->resource_map_1D.clear();
917     this->string_map_1D.clear();
918     this->path_map_1D.clear();
919
920     this->resource_map_2D.clear();
921     this->string_map_2D.clear();
922     this->path_map_2D.clear();
923
924     return;
925 } /* 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:

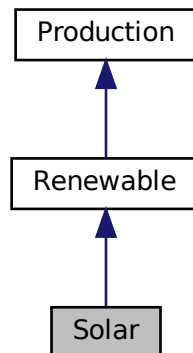
- header/[Resources.h](#)
- source/[Resources.cpp](#)

4.24 Solar Class Reference

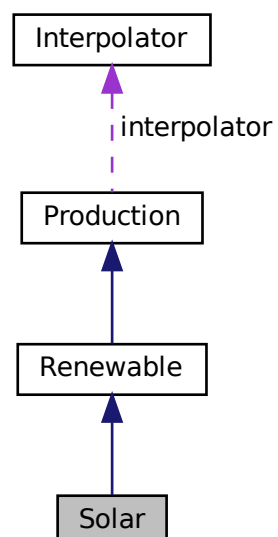
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

```
#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](#))
Constructor (intended) for the [Solar](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Solar](#) (void)
Destructor for the [Solar](#) class.

Public Attributes

- double [derating](#)
The derating of the solar PV array (i.e., shadowing, soiling, etc.).

Private Member Functions

- void [__checkInputs](#) ([SolarInputs](#))
Helper method to check inputs to the [Solar](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic solar PV array capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.
- void [__writeSummary](#) (std::string)
Helper method to write summary results for [Solar](#).
- void [__writeTimeSeries](#) (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)
Helper method to write time series results for [Solar](#).

4.24.1 Detailed Description

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

4.24.2 Constructor & Destructor Documentation

4.24.2.1 Solar() [1/2]

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

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

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

4.24.2.2 Solar() [2/2]

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

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

Parameters

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

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

4.24.2.3 ~Solar()

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

Destructor for the [Solar](#) class.

```
488 {
489     // 1. destruction print
490     if (this->print_flag) {
491         std::cout << "Solar object at " << this << " destroyed" << std::endl;
492     }
493
494     return;
495 } /* ~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.

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

4.24.3.2 __getGenericCapitalCost()

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

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

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

Returns

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

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


4.24.3.3 `__getGenericOpMaintCost()`

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

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

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

Returns

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

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

4.24.3.4 `__writeSummary()`

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

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

Parameters

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

Reimplemented from [Renewable](#).

```
123 {
124     // 1. create filestream
125     write_path += "summary_results.md";
126     std::ofstream ofs;
127     ofs.open(write_path, std::ofstream::out);
128
129     // 2. write summary results (markdown)
130     ofs << "# ";
131     ofs << std::to_string(int(ceil(this->capacity_kW)));
132     ofs << " kW SOLAR Summary Results\n";
133     ofs << "\n-----\n\n";
134
135     // 2.1. Production attributes
136     ofs << "## Production Attributes\n";
137     ofs << "\n";
138
139     ofs << "Capacity: " << this->capacity_kW << "kW\n";
140     ofs << "\n";
141
142     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << "\n";
143     ofs << "Capital Cost: " << this->capital_cost << "\n";
144     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
145         << " per kWh produced\n";
146     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
147         << "\n";
148     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
149         << "\n";
150     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << "\n";
151     ofs << "\n";
152
153     ofs << "Replacement Running Hours: " << this->replace_running_hrs << "\n";
154     ofs << "\n-----\n\n";
```

```

155
156 // 2.2. Renewable attributes
157 ofs << "## Renewable Attributes\n";
158 ofs << "\n";
159
160 ofs << "Resource Key (1D): " << this->resource_key << " \n";
161
162 ofs << "\n-----\n\n";
163
164 // 2.3. Solar attributes
165 ofs << "## Solar Attributes\n";
166 ofs << "\n";
167
168 ofs << "Derating Factor: " << this->derating << " \n";
169
170 ofs << "\n-----\n\n";
171
172 // 2.4. Solar Results
173 ofs << "## Results\n";
174 ofs << "\n";
175
176 ofs << "Net Present Cost: " << this->net_present_cost << " \n";
177 ofs << "\n";
178
179 ofs << "Total Dispatch: " << this->total_dispatch_kWh
180     << " kWh \n";
181
182 ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
183     << " per kWh dispatched \n";
184 ofs << "\n";
185
186 ofs << "Running Hours: " << this->running_hours << " \n";
187 ofs << "Replacements: " << this->n_replacements << " \n";
188
189 ofs << "\n-----\n\n";
190
191 ofs.close();
192 return;
193 } /* __writeSummary() */

```

4.24.3.5 __writeTimeSeries()

```

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

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

231 {
232 // 1. create filestream
233 write_path += "time_series_results.csv";
234 std::ofstream ofs;
235 ofs.open(write_path, std::ofstream::out);
236

```

```

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

```

4.24.3.6 commit()

```

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

```

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```

460 {
461     // 1. invoke base class method
462     load_kW = Renewable::commit(
463         timestep,
464         dt_hrs,
465         production_kW,
466         load_kW
467     );
468
469     //...
470
471     return load_kW;
472 } /* commit() */

```

4.24.3.7 computeProductionkW()

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

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

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

Parameters

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

Returns

The production [kW] of the solar PV array.

Reimplemented from [Renewable](#).

```
409 {
410     // check if no resource
411     if (solar_resource_kWm2 <= 0) {
412         return 0;
413     }
414
415     // compute production
416     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
417
418     // cap production at capacity
419     if (production_kW > this->capacity_kW) {
420         production_kW = this->capacity_kW;
421     }
422
423     return production_kW;
424 } /* computeProductionkW() */
```

4.24.3.8 handleReplacement()

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

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

Parameters

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

Reimplemented from [Renewable](#).

```
367 {
368     // 1. reset attributes
369     //...
```

```

370
371 // 2. invoke base class method
372 Renewable :: handleReplacement(timestep);
373
374 return;
375 } /* __handleReplacement() */

```

4.24.4 Member Data Documentation

4.24.4.1 derating

```
double Solar::derating
```

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

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

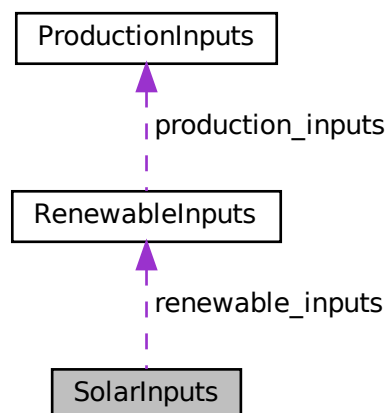
- header/Production/Renewable/[Solar.h](#)
- source/Production/Renewable/[Solar.cpp](#)

4.25 SolarInputs Struct Reference

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

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

Collaboration diagram for SolarInputs:



Public Attributes

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

4.25.1 Detailed Description

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

4.25.2 Member Data Documentation

4.25.2.1 capital_cost

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

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

4.25.2.2 derating

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

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

4.25.2.3 operation_maintenance_cost_kWh

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

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

4.25.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.25.2.5 resource_key

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

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

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

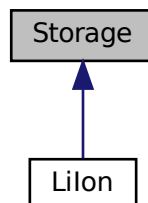
- header/Production/Renewable/[Solar.h](#)

4.26 Storage Class Reference

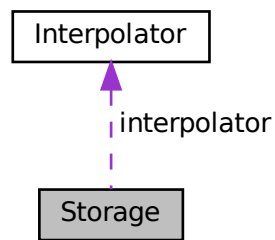
The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

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

Inheritance diagram for Storage:



Collaboration diagram for Storage:



Public Member Functions

- [Storage](#) (void)
Constructor (dummy) for the [Storage](#) class.
- [Storage](#) (int, double, [StorageInputs](#))
Constructor (intended) for the [Storage](#) class.
- virtual void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- void [computeEconomics](#) (std::vector< double > *)
Helper method to compute key economic metrics for the [Model](#) run.
- virtual double [getAvailablekW](#) (double)
- virtual double [getAcceptablekW](#) (double)
- virtual void [commitCharge](#) (int, double, double)
- virtual double [commitDischarge](#) (int, double, double, double)
- void [writeResults](#) (std::string, std::vector< double > *, int, int=-1)
Method which writes [Storage](#) results to an output directory.
- virtual [~Storage](#) (void)
Destructor for the [Storage](#) class.

Public Attributes

- [StorageType](#) type
The type ([StorageType](#)) of the asset.
- [Interpolator](#) interpolator
[Interpolator](#) component of [Storage](#).
- bool [print_flag](#)
A flag which indicates whether or not object construct/destruction should be verbose.
- bool [is_depleted](#)
A boolean which indicates whether or not the asset is currently considered depleted.
- bool [is_sunk](#)
A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- int [n_points](#)
The number of points in the modelling time series.

- int [n_replacements](#)
The number of times the asset has been replaced.
- double [n_years](#)
The number of years being modelled.
- double [power_capacity_kW](#)
The rated power capacity [kW] of the asset.
- double [energy_capacity_kWh](#)
The rated energy capacity [kWh] of the asset.
- double [charge_kWh](#)
The energy [kWh] stored in the asset.
- double [power_kW](#)
The power [kW] currently being charged/discharged by the asset.
- double [nominal_inflation_annual](#)
The nominal, annual inflation rate to use in computing model economics.
- double [nominal_discount_annual](#)
The nominal, annual discount rate to use in computing model economics.
- double [real_discount_annual](#)
The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double [capital_cost](#)
The capital cost of the asset (undefined currency).
- double [operation_maintenance_cost_kWh](#)
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy charged/discharged.
- double [net_present_cost](#)
The net present cost of this asset.
- double [total_discharge_kWh](#)
The total energy discharged [kWh] over the [Model](#) run.
- double [levellized_cost_of_energy_kWh](#)
The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only discharge.
- std::string [type_str](#)
A string describing the type of the asset.
- std::vector< double > [charge_vec_kWh](#)
A vector of the charge state [kWh] at each point in the modelling time series.
- std::vector< double > [charging_power_vec_kW](#)
A vector of the charging power [kW] at each point in the modelling time series.
- std::vector< double > [discharging_power_vec_kW](#)
A vector of the discharging power [kW] at each point in the modelling time series.
- std::vector< double > [capital_cost_vec](#)
A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > [operation_maintenance_cost_vec](#)
A vector of operation and maintenance costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).

Private Member Functions

- void [__checkInputs](#) (int, double, [StorageInputs](#))
Helper method to check inputs to the [Storage](#) constructor.
- double [__computeRealDiscountAnnual](#) (double, double)
Helper method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void [__writeSummary](#) (std::string)
- virtual void [__writeTimeSeries](#) (std::string, std::vector< double > *, int=-1)

4.26.1 Detailed Description

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

4.26.2 Constructor & Destructor Documentation

4.26.2.1 Storage() [1/2]

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

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

```
151 {
152     return;
153 } /* 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.

```
182 {
183     // 1. check inputs
184     this->__checkInputs(n_points, n_years, storage_inputs);
185
186     // 2. set attributes
187     this->print_flag = storage_inputs.print_flag;
188     this->is_depleted = false;
189     this->is_sunk = storage_inputs.is_sunk;
190
191     this->n_points = n_points;
192     this->n_replacements = 0;
193
194     this->n_years = n_years;
195
196     this->power_capacity_kW = storage_inputs.power_capacity_kW;
197     this->energy_capacity_kWh = storage_inputs.energy_capacity_kWh;
198
199     this->charge_kWh = 0;
200     this->power_kW = 0;
201
202     this->nominal_inflation_annual = storage_inputs.nominal_inflation_annual;
203     this->nominal_discount_annual = storage_inputs.nominal_discount_annual;
204
205     this->real_discount_annual = this->__computeRealDiscountAnnual(
206         storage_inputs.nominal_inflation_annual,
```

```

207     storage_inputs.nominal_discount_annual
208 );
209
210 this->capital_cost = 0;
211 this->operation_maintenance_cost_kWh = 0;
212 this->net_present_cost = 0;
213 this->total_discharge_kWh = 0;
214 this->levellized_cost_of_energy_kWh = 0;
215
216 this->charge_vec_kWh.resize(this->n_points, 0);
217 this->charging_power_vec_kW.resize(this->n_points, 0);
218 this->discharging_power_vec_kW.resize(this->n_points, 0);
219
220 this->capital_cost_vec.resize(this->n_points, 0);
221 this->operation_maintenance_cost_vec.resize(this->n_points, 0);
222
223 // 3. construction print
224 if (this->print_flag) {
225     std::cout << "Storage object constructed at " << this << std::endl;
226 }
227
228 return;
229 } /* Storage() */

```

4.26.2.3 ~Storage()

```

Storage::~Storage (
    void ) [virtual]

```

Destructor for the [Storage](#) class.

```

408 {
409     // 1. destruction print
410     if (this->print_flag) {
411         std::cout << "Storage object at " << this << " destroyed" << std::endl;
412     }
413
414     return;
415 } /* ~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.

```

45 {
46     // 1. check n_points
47     if (n_points <= 0) {
48         std::string error_str = "ERROR: Storage(): n_points must be > 0";

```

```

49
50     #ifdef _WIN32
51         std::cout << error_str << std::endl;
52     #endif
53
54     throw std::invalid_argument(error_str);
55 }
56
57 // 2. check n_years
58 if (n_years <= 0) {
59     std::string error_str = "ERROR: Storage(): n_years must be > 0";
60
61     #ifdef _WIN32
62         std::cout << error_str << std::endl;
63     #endif
64
65     throw std::invalid_argument(error_str);
66 }
67
68 // 3. check power_capacity_kW
69 if (storage_inputs.power_capacity_kW <= 0) {
70     std::string error_str = "ERROR: Storage(): ";
71     error_str += "StorageInputs::power_capacity_kW must be > 0";
72
73     #ifdef _WIN32
74         std::cout << error_str << std::endl;
75     #endif
76
77     throw std::invalid_argument(error_str);
78 }
79
80 // 4. check energy_capacity_kWh
81 if (storage_inputs.energy_capacity_kWh <= 0) {
82     std::string error_str = "ERROR: Storage(): ";
83     error_str += "StorageInputs::energy_capacity_kWh must be > 0";
84
85     #ifdef _WIN32
86         std::cout << error_str << std::endl;
87     #endif
88
89     throw std::invalid_argument(error_str);
90 }
91
92 return;
93 } /* __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.

```
127 {  
128     double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;  
129     real_discount_annual /= 1 + nominal_inflation_annual;  
130  
131     return real_discount_annual;  
132 } /* __computeRealDiscountAnnual() */
```

4.26.3.3 __writeSummary()

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

Reimplemented in [Lilon](#).

```
79 {return;}
```

4.26.3.4 __writeTimeSeries()

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

Reimplemented in [Lilon](#).

```
80 {return;}
```

4.26.3.5 commitCharge()

```
virtual void Storage::commitCharge (  
    int ,  
    double ,  
    double ) [inline], [virtual]
```

Reimplemented in [Lilon](#).

```
134 {return;}
```

4.26.3.6 commitDischarge()

```
virtual double Storage::commitDischarge (  
    int ,  
    double ,  
    double ,  
    double ) [inline], [virtual]
```

Reimplemented in [Lilon](#).

```
135 {return 0;}
```

4.26.3.7 computeEconomics()

```
void Storage::computeEconomics (
    std::vector< double > * time_vec_hrs_ptr )
```

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

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

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

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

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

Parameters

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

1. compute levellized cost of energy (per unit discharged)

```
282 {
283     // 1. compute net present cost
284     double t_hrs = 0;
285     double real_discount_scalar = 0;
286
287     for (int i = 0; i < this->n_points; i++) {
288         t_hrs = time_vec_hrs_ptr->at(i);
289
290         real_discount_scalar = 1.0 / pow(
291             1 + this->real_discount_annual,
292             t_hrs / 8760
293         );
294
295         this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
296
297         this->net_present_cost +=
298             real_discount_scalar * this->operation_maintenance_cost_vec[i];
299     }
300
301     // assuming 8,760 hours per year
302     double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
303
304     double capital_recovery_factor =
305         (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
306         (pow(1 + this->real_discount_annual, n_years) - 1);
307
308     double total_annualized_cost = capital_recovery_factor *
309         this->net_present_cost;
310
311     this->levellized_cost_of_energy_kWh =
312         (n_years * total_annualized_cost) /
313         this->total_discharge_kWh;
314
315     return;
316 } /* computeEconomics() */
```

4.26.3.8 getAcceptablekW()

```
virtual double Storage::getAcceptablekW (
    double ) [inline], [virtual]
```

Reimplemented in [Lilon](#).

```
132 {return 0;}
```

4.26.3.9 getAvailablekW()

```
virtual double Storage::getAvailablekW (
    double ) [inline], [virtual]
```

Reimplemented in [Lilon](#).

```
131 {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](#).

```
247 {
248     // 1. reset attributes
249     this->charge_kWh = 0;
250     this->power_kW = 0;
251
252     // 2. log replacement
253     this->n_replacements++;
254
255     // 3. incur capital cost in timestep
256     this->capital_cost_vec[timestep] = this->capital_cost;
257
258     return;
259 } /* __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 <i>time_vec_hrs</i> 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.

```

354 {
355     // 1. handle sentinel
356     if (max_lines < 0) {
357         max_lines = this->n_points;
358     }
359
360     // 2. create subdirectories
361     write_path += "Storage/";
362     if (not std::filesystem::is_directory(write_path)) {
363         std::filesystem::create_directory(write_path);
364     }
365
366     write_path += this->type_str;
367     write_path += "_";
368     write_path += std::to_string(int(ceil(this->power_capacity_kW)));
369     write_path += "kW_";
370     write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
371     write_path += "kWh_idx";
372     write_path += std::to_string(storage_index);
373     write_path += "/";
374     std::filesystem::create_directory(write_path);
375
376     // 3. write summary
377     this->__writeSummary(write_path);
378
379     // 4. write time series
380     if (max_lines > this->n_points) {
381         max_lines = this->n_points;
382     }
383
384     if (max_lines > 0) {
385         this->__writeTimeSeries(
386             write_path,
387             time_vec_hrs_ptr,
388             max_lines
389         );
390     }
391
392     return;
393 } /* 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:

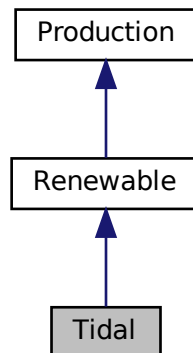
- header/Storage/[Storage.h](#)

4.28 Tidal Class Reference

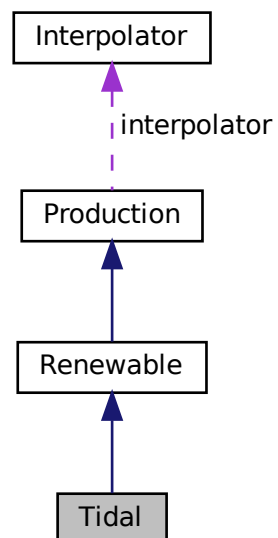
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

```
#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](#))
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::vector< double > *, 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.

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

4.28.2.2 Tidal() [2/2]

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

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

Parameters

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

```
457 :
458 Renewable(
459     n_points,
460     n_years,
461     tidal_inputs.renewable_inputs
462 )
463 {
464     // 1. check inputs
465     this->__checkInputs(tidal_inputs);
466
467     // 2. set attributes
468     this->type = RenewableType :: TIDAL;
469     this->type_str = "TIDAL";
470
471     this->resource_key = tidal_inputs.resource_key;
472
473     this->design_speed_ms = tidal_inputs.design_speed_ms;
474
475     this->power_model = tidal_inputs.power_model;
476
477     switch (this->power_model) {
478         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
479             this->power_model_string = "CUBIC";
480             break;
481         }
482
483         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
484             this->power_model_string = "EXPONENTIAL";
485             break;
486         }
487
488         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
```

```

491         this->power_model_string = "LOOKUP";
492
493         break;
494     }
495
496     default: {
497         std::string error_str = "ERROR: Tidal(): ";
498         error_str += "power production model ";
499         error_str += std::to_string(this->power_model);
500         error_str += " not recognized";
501
502         #ifdef _WIN32
503             std::cout << error_str << std::endl;
504         #endif
505
506         throw std::runtime_error(error_str);
507
508         break;
509     }
510 }
511
512 if (tidal_inputs.capital_cost < 0) {
513     this->capital_cost = this->__getGenericCapitalCost();
514 }
515
516 if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
517     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
518 }
519
520 if (not this->is_sunk) {
521     this->capital_cost_vec[0] = this->capital_cost;
522 }
523
524 // 3. construction print
525 if (this->print_flag) {
526     std::cout << "Tidal object constructed at " << this << std::endl;
527 }
528
529 return;
530 } /* Renewable() */

```

4.28.2.3 ~Tidal()

```

Tidal::~Tidal (
    void )

```

Destructor for the [Tidal](#) class.

```

710 {
711     // 1. destruction print
712     if (this->print_flag) {
713         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
714     }
715
716     return;
717 } /* ~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.

```

37 {

```

```

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

```

4.28.3.2 __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. \[2023\]](#)

Parameters

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

Returns

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

```

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

```

4.28.3.3 `__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.

```
195 {
196     double production = 0;
197
198     double turbine_speed =
199         (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
200
201     if (turbine_speed < -0.71 or turbine_speed > 0.65) {
202         production = 0;
203     }
204
205     else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
206         production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
207     }
208
209     else {
210         production = 1;
211     }
212
213     return production * this->capacity_kW;
214 } /* __computeExponentialProductionkW() */
```

4.28.3.4 `__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.

```

246 {
247     // *** WORK IN PROGRESS *** //
248
249     return 0;
250 } /* __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].

```

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

```

4.28.3.6 __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].

```

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

```

4.28.3.7 __writeSummary()

```

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

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

268 {
269     // 1. create filestream
270     write_path += "summary_results.md";
271     std::ofstream ofs;
272     ofs.open(write_path, std::ofstream::out);
273
274     // 2. write summary results (markdown)
275     ofs << "# ";
276     ofs << std::to_string(int(ceil(this->capacity_kW)));
277     ofs << " kW TIDAL Summary Results\n";
278     ofs << "\n-----\n\n";
279
280     // 2.1. Production attributes
281     ofs << "## Production Attributes\n";
282     ofs << "\n";
283
284     ofs << "Capacity: " << this->capacity_kW << "kW \n";
285     ofs << "\n";
286
287     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
288     ofs << "Capital Cost: " << this->capital_cost << " \n";
289     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
290         << " per kWh produced \n";
291     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
292         << " \n";
293     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
294         << " \n";
295     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
296     ofs << "\n";
297
298     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
299     ofs << "\n-----\n\n";
300
301     // 2.2. Renewable attributes
302     ofs << "## Renewable Attributes\n";
303     ofs << "\n";
304
305     ofs << "Resource Key (ID): " << this->resource_key << " \n";
306
307     ofs << "\n-----\n\n";
308
309     // 2.3. Tidal attributes
310     ofs << "## Tidal Attributes\n";
311     ofs << "\n";
312
313     ofs << "Power Production Model: " << this->power_model_string << " \n";
314     ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
315
316     ofs << "\n-----\n\n";
317
318     // 2.4. Tidal Results
319     ofs << "## Results\n";
320     ofs << "\n";
321
322     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
323     ofs << "\n";
324
325     ofs << "Total Dispatch: " << this->total_dispatch_kWh
326         << " kWh \n";
327
328     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
329         << " per kWh dispatched \n";
330     ofs << "\n";
331
332     ofs << "Running Hours: " << this->running_hours << " \n";
333     ofs << "Replacements: " << this->n_replacements << " \n";
334
335     ofs << "\n-----\n\n";
336
337     ofs.close();
338
339     return;
340 } /* __writeSummary() */

```

4.28.3.8 `__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](#).

```
378 {
379     // 1. create filestream
380     write_path += "time_series_results.csv";
381     std::ofstream ofs;
382     ofs.open(write_path, std::ofstream::out);
383
384     // 2. write time series results (comma separated value)
385     ofs << "Time (since start of data) [hrs],";
386     ofs << "Tidal Resource [m/s],";
387     ofs << "Production [kW],";
388     ofs << "Dispatch [kW],";
389     ofs << "Storage [kW],";
390     ofs << "Curtailment [kW],";
391     ofs << "Capital Cost (actual),";
392     ofs << "Operation and Maintenance Cost (actual),";
393     ofs << "\n";
394
395     for (int i = 0; i < max_lines; i++) {
396         ofs << time_vec_hrs_ptr->at(i) << ",";
397         ofs << resource_map_1D_ptr->at(this->resource_key)[i] << ",";
398         ofs << this->production_vec_kW[i] << ",";
399         ofs << this->dispatch_vec_kW[i] << ",";
400         ofs << this->storage_vec_kW[i] << ",";
401         ofs << this->curtailment_vec_kW[i] << ",";
402         ofs << this->capital_cost_vec[i] << ",";
403         ofs << this->operation_maintenance_cost_vec[i] << ",";
404         ofs << "\n";
405     }
406
407     return;
408 } /* __writeTimeSeries() */
```

4.28.3.9 `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](#).

```

682 {
683     // 1. invoke base class method
684     load_kW = Renewable::commit(
685         timestep,
686         dt_hrs,
687         production_kW,
688         load_kW
689     );
690
691
692     //...
693
694     return load_kW;
695 } /* 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](#).

```

588 {
589     // check if no resource
590     if (tidal_resource_ms <= 0) {
591         return 0;
592     }
593
594     // compute production
595     double production_kW = 0;

```



```

596
597     switch (this->power_model) {
598         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
599             production_kW = this->__computeCubicProductionkW(
600                 timestep,
601                 dt_hrs,
602                 tidal_resource_ms
603             );
604
605             break;
606         }
607
608         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
609             production_kW = this->__computeExponentialProductionkW(
610                 timestep,
611                 dt_hrs,
612                 tidal_resource_ms
613             );
614
615             break;
616         }
617
618         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
619             production_kW = this->__computeLookupProductionkW(
620                 timestep,
621                 dt_hrs,
622                 tidal_resource_ms
623             );
624
625             break;
626         }
627
628         default: {
629             std::string error_str = "ERROR: Tidal::computeProductionkW(): ";
630             error_str += "power model ";
631             error_str += std::to_string(this->power_model);
632             error_str += " not recognized";
633
634             #ifdef _WIN32
635                 std::cout << error_str << std::endl;
636             #endif
637
638             throw std::runtime_error(error_str);
639
640             break;
641         }
642     }
643
644     return production_kW;
645 } /* 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](#).

```

548 {
549     // 1. reset attributes
550     //...
551
552     // 2. invoke base class method
553     Renewable :: handleReplacement(timestep);
554
555     return;
556 } /* __handleReplacement() */

```

4.28.4 Member Data Documentation

4.28.4.1 design_speed_ms

```
double Tidal::design_speed_ms
```

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

4.28.4.2 power_model

```
TidalPowerProductionModel Tidal::power_model
```

The tidal power production model to be applied.

4.28.4.3 power_model_string

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

A string describing the active power production model.

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

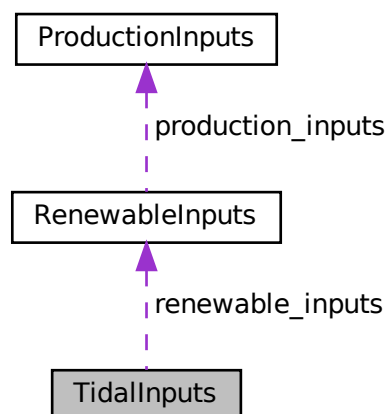
- [header/Production/Renewable/Tidal.h](#)
- [source/Production/Renewable/Tidal.cpp](#)

4.29 TidalInputs Struct Reference

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

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

Collaboration diagram for TidalInputs:



Public Attributes

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

4.29.1 Detailed Description

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

4.29.2 Member Data Documentation

4.29.2.1 capital_cost

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

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

4.29.2.2 design_speed_ms

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

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

4.29.2.3 operation_maintenance_cost_kWh

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

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

4.29.2.4 power_model

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

The tidal power production model to be applied.

4.29.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.29.2.6 resource_key

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

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

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

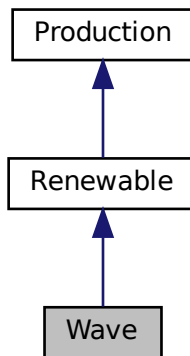
- [header/Production/Renewable/Tidal.h](#)

4.30 Wave Class Reference

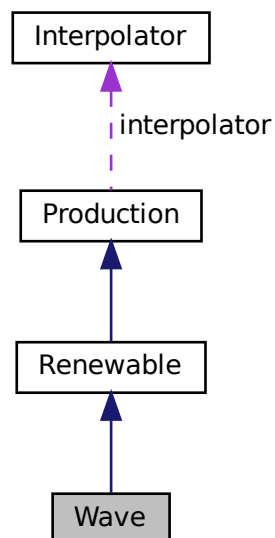
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

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

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

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

```
501 {
502     return;
503 } /* Wave() */
```

4.30.2.2 Wave() [2/2]

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

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

Parameters

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

```
531 :
532 Renewable(
533     n_points,
534     n_years,
535     wave_inputs.renewable_inputs
536 )
537 {
538     // 1. check inputs
539     this->__checkInputs(wave_inputs);
540
541     // 2. set attributes
542     this->type = RenewableType :: WAVE;
543     this->type_str = "WAVE";
544
545     this->resource_key = wave_inputs.resource_key;
546
547     this->design_significant_wave_height_m =
548         wave_inputs.design_significant_wave_height_m;
549     this->design_energy_period_s = wave_inputs.design_energy_period_s;
550
551     this->power_model = wave_inputs.power_model;
552
553     switch (this->power_model) {
554         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
555             this->power_model_string = "GAUSSIAN";
```

```

556
557     break;
558 }
559
560 case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
561     this->power_model_string = "PARABOLOID";
562
563     break;
564 }
565
566 case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
567     this->power_model_string = "LOOKUP";
568
569     this->interpolator.addData2D(
570         0,
571         wave_inputs.path_2_normalized_performance_matrix
572     );
573
574     break;
575 }
576
577 default: {
578     std::string error_str = "ERROR: Wave(): ";
579     error_str += "power production model ";
580     error_str += std::to_string(this->power_model);
581     error_str += " not recognized";
582
583     #ifdef _WIN32
584         std::cout << error_str << std::endl;
585     #endif
586
587     throw std::runtime_error(error_str);
588
589     break;
590 }
591 }
592
593 if (wave_inputs.capital_cost < 0) {
594     this->capital_cost = this->__getGenericCapitalCost();
595 }
596
597 if (wave_inputs.operation_maintenance_cost_kWh < 0) {
598     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
599 }
600
601 if (not this->is_sunk) {
602     this->capital_cost_vec[0] = this->capital_cost;
603 }
604
605 // 3. construction print
606 if (this->print_flag) {
607     std::cout << "Wave object constructed at " << this << std::endl;
608 }
609
610 return;
611 } /* Renewable() */

```

4.30.2.3 ~Wave()

```

Wave::~~Wave (
    void )

```

Destructor for the [Wave](#) class.

```

797 {
798     // 1. destruction print
799     if (this->print_flag) {
800         std::cout << "Wave object at " << this << " destroyed" << std::endl;
801     }
802
803     return;
804 } /* ~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

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

```
39 {
40     // 1. check design_significant_wave_height_m
41     if (wave_inputs.design_significant_wave_height_m <= 0) {
42         std::string error_str = "ERROR: Wave(): ";
43         error_str += "WaveInputs::design_significant_wave_height_m must be > 0";
44
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check design_energy_period_s
53     if (wave_inputs.design_energy_period_s <= 0) {
54         std::string error_str = "ERROR: Wave(): ";
55         error_str += "WaveInputs::design_energy_period_s must be > 0";
56
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     // 3. if WAVE_POWER_LOOKUP, check that path is given
65     if (
66         wave_inputs.power_model == WavePowerProductionModel::WAVE_POWER_LOOKUP and
67         wave_inputs.path_2_normalized_performance_matrix.empty()
68     ) {
69         std::string error_str = "ERROR: Wave() power model was set to ";
70         error_str += "WavePowerProductionModel::WAVE_POWER_LOOKUP, but no path to a ";
71         error_str += "normalized performance matrix was given";
72
73         #ifdef _WIN32
74             std::cout << error_str << std::endl;
75         #endif
76
77         throw std::invalid_argument(error_str);
78     }
79
80     return;
81 } /* __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.

```

176 {
177     double H_s_nondim =
178         (significant_wave_height_m - this->design_significant_wave_height_m) /
179         this->design_significant_wave_height_m;
180
181     double T_e_nondim =
182         (energy_period_s - this->design_energy_period_s) /
183         this->design_energy_period_s;
184
185     double production = exp(
186         -2.25119 * pow(T_e_nondim, 2) +
187         3.44570 * T_e_nondim * H_s_nondim -
188         4.01508 * pow(H_s_nondim, 2)
189     );
190
191     return production * this->capacity_kW;
192 } /* __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.

```

293 {
294     double prod = this->interpolator.interp2D(
295         0,
296         significant_wave_height_m,
297         energy_period_s
298     );
299

```

```

300     return prod * this->capacity_kW;
301 } /* __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.

```

233 {
234     // first, check for idealized wave breaking (deep water)
235     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
236         return 0;
237     }
238
239     // otherwise, apply generic quadratic performance model
240     // (with outputs bounded to [0, 1])
241     double production =
242         0.289 * significant_wave_height_m -
243         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
244         0.0169 * energy_period_s;
245
246     if (production < 0) {
247         production = 0;
248     }
249
250     else if (production > 1) {
251         production = 1;
252     }
253
254     return production * this->capacity_kW;
255 } /* __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].

```
103 {
104     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
105
106     return capital_cost_per_kW * this->capacity_kW;
107 } /* __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].

```
131 {
132     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
133
134     return operation_maintenance_cost_kWh;
135 } /* __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](#).

```

319 {
320     // 1. create filestream
321     write_path += "summary_results.md";
322     std::ofstream ofs;
323     ofs.open(write_path, std::ofstream::out);
324
325     // 2. write summary results (markdown)
326     ofs << "# ";
327     ofs << std::to_string(int(ceil(this->capacity_kW)));
328     ofs << " kW WAVE Summary Results\n";
329     ofs << "\n-----\n\n";
330
331     // 2.1. Production attributes
332     ofs << "## Production Attributes\n";
333     ofs << "\n";
334
335     ofs << "Capacity: " << this->capacity_kW << "kW \n";
336     ofs << "\n";
337
338     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
339     ofs << "Capital Cost: " << this->capital_cost << " \n";
340     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
341         << " per kWh produced \n";
342     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
343         << " \n";
344     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
345         << " \n";
346     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
347     ofs << "\n";
348
349     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
350     ofs << "\n-----\n\n";
351
352     // 2.2. Renewable attributes
353     ofs << "## Renewable Attributes\n";
354     ofs << "\n";
355
356     ofs << "Resource Key (2D): " << this->resource_key << " \n";
357
358     ofs << "\n-----\n\n";
359
360     // 2.3. Wave attributes
361     ofs << "## Wave Attributes\n";
362     ofs << "\n";
363
364     ofs << "Power Production Model: " << this->power_model_string << " \n";
365     switch (this->power_model) {
366     case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
367         ofs << "Design Significant Wave Height: "
368             << this->design_significant_wave_height_m << " m \n";
369
370         ofs << "Design Energy Period: " << this->design_energy_period_s << " s \n";
371
372         break;
373     }
374
375     case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
376         ofs << "Normalized Performance Matrix: "
377             << this->interpolator.path_map_2D[0] << " \n";
378
379         break;
380     }
381
382     default: {
383         // write nothing!
384
385         break;
386     }
387     }
388
389     ofs << "\n-----\n\n";
390
391     // 2.4. Wave Results
392     ofs << "## Results\n";
393     ofs << "\n";
394
395     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
396     ofs << "\n";
397
398     ofs << "Total Dispatch: " << this->total_dispatch_kWh
399         << " kWh \n";
400
401     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
402         << " per kWh dispatched \n";
403     ofs << "\n";
404

```

```

405     ofs << "Running Hours: " << this->running_hours << " \n";
406     ofs << "Replacements: " << this->n_replacements << " \n";
407
408     ofs << "\n-----\n\n";
409
410     ofs.close();
411
412     return;
413 } /* __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](#).

```

451 {
452     // 1. create filestream
453     write_path += "time_series_results.csv";
454     std::ofstream ofs;
455     ofs.open(write_path, std::ofstream::out);
456
457     // 2. write time series results (comma separated value)
458     ofs << "Time (since start of data) [hrs],";
459     ofs << "Significant Wave Height [m],";
460     ofs << "Energy Period [s],";
461     ofs << "Production [kW],";
462     ofs << "Dispatch [kW],";
463     ofs << "Storage [kW],";
464     ofs << "Curtailement [kW],";
465     ofs << "Capital Cost (actual),";
466     ofs << "Operation and Maintenance Cost (actual),";
467     ofs << "\n";
468
469     for (int i = 0; i < max_lines; i++) {
470         ofs << time_vec_hrs_ptr->at(i) << ",";
471         ofs << resource_map_2D_ptr->at(this->resource_key)[i][0] << ",";
472         ofs << resource_map_2D_ptr->at(this->resource_key)[i][1] << ",";
473         ofs << this->production_vec_kW[i] << ",";
474         ofs << this->dispatch_vec_kW[i] << ",";
475         ofs << this->storage_vec_kW[i] << ",";
476         ofs << this->curtailement_vec_kW[i] << ",";
477         ofs << this->capital_cost_vec[i] << ",";
478         ofs << this->operation_maintenance_cost_vec[i] << ",";
479         ofs << "\n";
480     }
481
482     return;
483 } /* __writeTimeSeries() */

```

4.30.3.9 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](#).

```
769 {
770     // 1. invoke base class method
771     load_kW = Renewable::commit(
772         timestep,
773         dt_hrs,
774         production_kW,
775         load_kW
776     );
777
778
779     //...
780
781     return load_kW;
782 } /* 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](#).

```

673 {
674     // check if no resource
675     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
676         return 0;
677     }
678
679     // compute production
680     double production_kW = 0;
681
682     switch (this->power_model) {
683         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
684             production_kW = this->__computeParaboloidProductionkW(
685                 timestep,
686                 dt_hrs,
687                 significant_wave_height_m,
688                 energy_period_s
689             );
690
691             break;
692         }
693
694         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
695             production_kW = this->__computeGaussianProductionkW(
696                 timestep,
697                 dt_hrs,
698                 significant_wave_height_m,
699                 energy_period_s
700             );
701
702             break;
703         }
704
705         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
706             production_kW = this->__computeLookupProductionkW(
707                 timestep,
708                 dt_hrs,
709                 significant_wave_height_m,
710                 energy_period_s
711             );
712
713             break;
714         }
715
716         default: {
717             std::string error_str = "ERROR: Wave::computeProductionkW(): ";
718             error_str += "power model ";
719             error_str += std::to_string(this->power_model);
720             error_str += " not recognized";
721
722             #ifdef _WIN32
723                 std::cout << error_str << std::endl;
724             #endif
725
726             throw std::runtime_error(error_str);
727
728             break;
729         }
730     }
731
732     return production_kW;
733 } /* 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](#).

```

629 {
630     // 1. reset attributes
631     //...
632
633     // 2. invoke base class method
634     Renewable::handleReplacement(timestep);
635
636     return;
637 } /* __handleReplacement() */

```

4.30.4 Member Data Documentation

4.30.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

4.30.4.2 design_significant_wave_height_m

```
double Wave::design_significant_wave_height_m
```

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

4.30.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

4.30.4.4 power_model_string

```
std::string Wave::power_model_string
```

A string describing the active power production model.

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

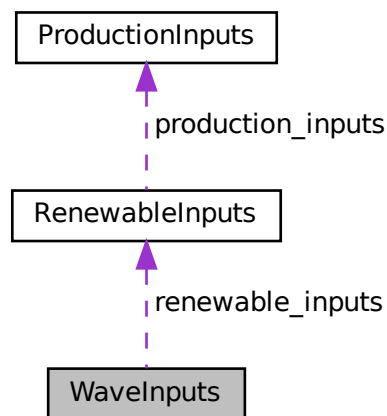
- header/Production/Renewable/[Wave.h](#)
- source/Production/Renewable/[Wave.cpp](#)

4.31 WaveInputs Struct Reference

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

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

Collaboration diagram for WaveInputs:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- `int` `resource_key` = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- `double` `capital_cost` = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- `double` `operation_maintenance_cost_kWh` = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- `double` `design_significant_wave_height_m` = 3
The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- `double` `design_energy_period_s` = 10
The energy period [s] at which the wave energy converter achieves its rated capacity.
- [WavePowerProductionModel](#) `power_model` = [WavePowerProductionModel](#) :: `WAVE_POWER_PARABOLOID`
The wave power production model to be applied.
- `std::string` `path_2_normalized_performance_matrix` = ""
A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.1 Detailed Description

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

4.31.2 Member Data Documentation

4.31.2.1 capital_cost

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

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

4.31.2.2 design_energy_period_s

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

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

4.31.2.3 design_significant_wave_height_m

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

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

4.31.2.4 operation_maintenance_cost_kWh

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

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

4.31.2.5 path_2_normalized_performance_matrix

```
std::string WaveInputs::path_2_normalized_performance_matrix = ""
```

A path (either relative or absolute) to a normalized performance matrix for the wave energy converter.

4.31.2.6 power_model

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

The wave power production model to be applied.

4.31.2.7 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.31.2.8 resource_key

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

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

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

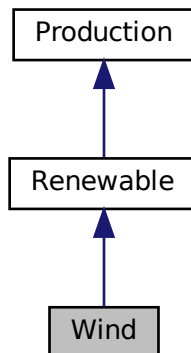
- [header/Production/Renewable/Wave.h](#)

4.32 Wind Class Reference

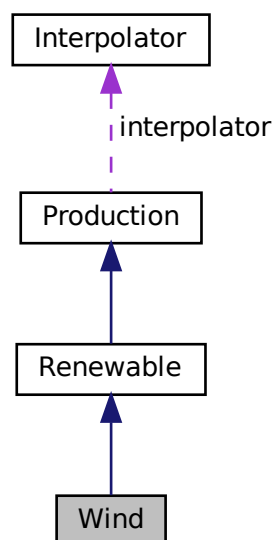
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

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

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- [Wind](#) (void)
Constructor (dummy) for the [Wind](#) class.
- [Wind](#) (int, double, [WindInputs](#))
Constructor (intended) for the [Wind](#) class.
- void [handleReplacement](#) (int)
Method to handle asset replacement and capital cost incursion, if applicable.
- double [computeProductionkW](#) (int, double, double)
Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.
- double [commit](#) (int, double, double, double)
Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- [~Wind](#) (void)
Destructor for the [Wind](#) class.

Public Attributes

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

Private Member Functions

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

4.32.1 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.

```
390 {
391     return;
392 } /* Wind() */
```

4.32.2.2 Wind() [2/2]

```
Wind::Wind (
    int n_points,
    double n_years,
    WindInputs wind_inputs )
```

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

Parameters

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

```
420 :
421 Renewable(
422     n_points,
423     n_years,
424     wind_inputs.renewable_inputs
425 )
426 {
427     // 1. check inputs
428     this->__checkInputs(wind_inputs);
429
430     // 2. set attributes
431     this->type = RenewableType :: WIND;
432     this->type_str = "WIND";
433
434     this->resource_key = wind_inputs.resource_key;
435
436     this->design_speed_ms = wind_inputs.design_speed_ms;
437
438     this->power_model = wind_inputs.power_model;
439
440     switch (this->power_model) {
441         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
442             this->power_model_string = "EXPONENTIAL";
443
444             break;
445         }
446
447         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
448             this->power_model_string = "LOOKUP";
449
450             break;
451         }
452
453         default: {
```

```

454         std::string error_str = "ERROR: Wind(): ";
455         error_str += "power production model ";
456         error_str += std::to_string(this->power_model);
457         error_str += " not recognized";
458
459         #ifdef _WIN32
460             std::cout << error_str << std::endl;
461         #endif
462
463         throw std::runtime_error(error_str);
464
465         break;
466     }
467 }
468
469 if (wind_inputs.capital_cost < 0) {
470     this->capital_cost = this->__getGenericCapitalCost();
471 }
472
473 if (wind_inputs.operation_maintenance_cost_kWh < 0) {
474     this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
475 }
476
477 if (not this->is_sunk) {
478     this->capital_cost_vec[0] = this->capital_cost;
479 }
480
481 // 3. construction print
482 if (this->print_flag) {
483     std::cout << "Wind object constructed at " << this << std::endl;
484 }
485
486 return;
487 } /* Renewable() */

```

4.32.2.3 ~Wind()

```

Wind::~Wind (
    void )

```

Destructor for the [Wind](#) class.

```

656 {
657     // 1. destruction print
658     if (this->print_flag) {
659         std::cout << "Wind object at " << this << " destroyed" << std::endl;
660     }
661
662     return;
663 } /* ~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.

Parameters

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


```

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

```

4.32.3.2 __computeExponentialProductionkW()

```

double Wind::__computeExponentialProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]

```

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

Ref: [Truelove et al. \[2019\]](#)

Parameters

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

Returns

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

```

140 {
141     double production = 0;
142
143     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
144         this->design_speed_ms;
145
146     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
147         production = 0;
148     }
149
150     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
151         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
152     }
153
154     else {
155         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
156     }
157
158     return production * this->capacity_kW;
159 } /* __computeExponentialProductionkW() */

```

4.32.3.3 __computeLookupProductionkW()

```

double Wind::__computeLookupProductionkW (
    int timestep,

```

```
double dt_hrs,
double wind_resource_ms ) [private]
```

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

Parameters

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

Returns

The interpolated production [kW] of the wind turbine.

```
191 {
192     // *** WORK IN PROGRESS *** //
193
194     return 0;
195 } /* __computeLookupProductionkW() */
```

4.32.3.4 __getGenericCapitalCost()

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

Helper method to generate a generic wind turbine capital cost.

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

Returns

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

```
75 {
76     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
77
78     return capital_cost_per_kW * this->capacity_kW;
79 } /* __getGenericCapitalCost() */
```

4.32.3.5 __getGenericOpMaintCost()

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

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

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

Returns

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

```
102 {
103     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
104
105     return operation_maintenance_cost_kWh;
106 } /* __getGenericOpMaintCost() */
```

4.32.3.6 `__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](#).

```
213 {
214     // 1. create filestream
215     write_path += "summary_results.md";
216     std::ofstream ofs;
217     ofs.open(write_path, std::ofstream::out);
218
219     // 2. write summary results (markdown)
220     ofs << "# ";
221     ofs << std::to_string(int(ceil(this->capacity_kW)));
222     ofs << " kW WIND Summary Results\n";
223     ofs << "\n-----\n\n";
224
225
226     // 2.1. Production attributes
227     ofs << "## Production Attributes\n";
228     ofs << "\n";
229
230     ofs << "Capacity: " << this->capacity_kW << "kW \n";
231     ofs << "\n";
232
233     ofs << "Sunk Cost (N = 0 / Y = 1): " << this->is_sunk << " \n";
234     ofs << "Capital Cost: " << this->capital_cost << " \n";
235     ofs << "Operation and Maintenance Cost: " << this->operation_maintenance_cost_kWh
236         << " per kWh produced \n";
237     ofs << "Nominal Inflation Rate (annual): " << this->nominal_inflation_annual
238         << " \n";
239     ofs << "Nominal Discount Rate (annual): " << this->nominal_discount_annual
240         << " \n";
241     ofs << "Real Discount Rate (annual): " << this->real_discount_annual << " \n";
242     ofs << "\n";
243
244     ofs << "Replacement Running Hours: " << this->replace_running_hrs << " \n";
245     ofs << "\n-----\n\n";
246
247     // 2.2. Renewable attributes
248     ofs << "## Renewable Attributes\n";
249     ofs << "\n";
250
251     ofs << "Resource Key (1D): " << this->resource_key << " \n";
252
253     ofs << "\n-----\n\n";
254
255     // 2.3. Wind attributes
256     ofs << "## Wind Attributes\n";
257     ofs << "\n";
258
259     ofs << "Power Production Model: " << this->power_model_string << " \n";
260     switch (this->power_model) {
261     case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
262         ofs << "Design Speed: " << this->design_speed_ms << " m/s \n";
263
264         break;
265     }
266
267     case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
268         //...
269
270         break;
271     }
272
273     default: {
274         // write nothing!
275
276         break;
277     }
278 }
```

```

278     }
279
280     ofs << "\n-----\n\n";
281
282     // 2.4. Wind Results
283     ofs << "## Results\n";
284     ofs << "\n";
285
286     ofs << "Net Present Cost: " << this->net_present_cost << " \n";
287     ofs << "\n";
288
289     ofs << "Total Dispatch: " << this->total_dispatch_kWh
290         << " kWh \n";
291
292     ofs << "Levellized Cost of Energy: " << this->levellized_cost_of_energy_kWh
293         << " per kWh dispatched \n";
294     ofs << "\n";
295
296     ofs << "Running Hours: " << this->running_hours << " \n";
297     ofs << "Replacements: " << this->n_replacements << " \n";
298
299     ofs << "\n-----\n\n";
300
301     ofs.close();
302
303     return;
304 } /* __writeSummary() */

```

4.32.3.7 __writeTimeSeries()

```

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

```

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

Parameters

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

Reimplemented from [Renewable](#).

```

342 {
343     // 1. create filestream
344     write_path += "time_series_results.csv";
345     std::ofstream ofs;
346     ofs.open(write_path, std::ofstream::out);
347
348     // 2. write time series results (comma separated value)
349     ofs << "Time (since start of data) [hrs],";
350     ofs << "Wind Resource [m/s],";
351     ofs << "Production [kW],";
352     ofs << "Dispatch [kW],";
353     ofs << "Storage [kW],";
354     ofs << "Curtailement [kW],";
355     ofs << "Capital Cost (actual),";
356     ofs << "Operation and Maintenance Cost (actual),";
357     ofs << "\n";
358
359     for (int i = 0; i < max_lines; i++) {

```

```

360         ofs << time_vec_hrs_ptr->at(i) << ", ";
361         ofs << resource_map_id_ptr->at(this->resource_key)[i] << ", ";
362         ofs << this->production_vec_kW[i] << ", ";
363         ofs << this->dispatch_vec_kW[i] << ", ";
364         ofs << this->storage_vec_kW[i] << ", ";
365         ofs << this->curtailment_vec_kW[i] << ", ";
366         ofs << this->capital_cost_vec[i] << ", ";
367         ofs << this->operation_maintenance_cost_vec[i] << ", ";
368         ofs << "\n";
369     }
370
371     return;
372 } /* __writeTimeSeries() */

```

4.32.3.8 commit()

```

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

```

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```

628 {
629     // 1. invoke base class method
630     load_kW = Renewable::commit(
631         timestep,
632         dt_hrs,
633         production_kW,
634         load_kW
635     );
636
637     //...
638
639     return load_kW;
640 } /* commit() */

```

4.32.3.9 computeProductionkW()

```

double Wind::computeProductionkW (
    int timestep,

```

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

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

Parameters

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

Returns

The production [kW] of the wind turbine.

Reimplemented from [Renewable](#).

```

545 {
546     // check if no resource
547     if (wind_resource_ms <= 0) {
548         return 0;
549     }
550
551     // compute production
552     double production_kW = 0;
553
554     switch (this->power_model) {
555         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
556             production_kW = this->__computeExponentialProductionkW(
557                 timestep,
558                 dt_hrs,
559                 wind_resource_ms
560             );
561
562             break;
563         }
564
565         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
566             production_kW = this->__computeLookupProductionkW(
567                 timestep,
568                 dt_hrs,
569                 wind_resource_ms
570             );
571
572             break;
573         }
574
575         default: {
576             std::string error_str = "ERROR: Wind::computeProductionkW(): ";
577             error_str += "power model ";
578             error_str += std::to_string(this->power_model);
579             error_str += " not recognized";
580
581             #ifdef _WIN32
582                 std::cout << error_str << std::endl;
583             #endif
584
585             throw std::runtime_error(error_str);
586
587             break;
588         }
589     }
590
591     return production_kW;
592 } /* computeProductionkW() */

```

4.32.3.10 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](#).

```

505 {
506     // 1. reset attributes
507     //...
508
509     // 2. invoke base class method
510     Renewable :: handleReplacement(timestep);
511
512     return;
513 } /* __handleReplacement() */

```

4.32.4 Member Data Documentation

4.32.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

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

4.32.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

4.32.4.3 power_model_string

```
std::string Wind::power_model_string
```

A string describing the active power production model.

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

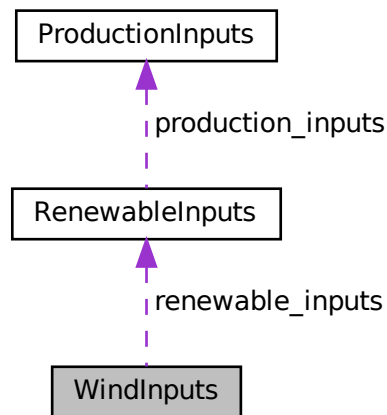
- header/Production/Renewable/[Wind.h](#)
- source/Production/Renewable/[Wind.cpp](#)

4.33 WindInputs Struct Reference

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

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

Collaboration diagram for WindInputs:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- int `resource_key` = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- double `capital_cost` = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double `operation_maintenance_cost_kWh` = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double `design_speed_ms` = 8
The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- [WindPowerProductionModel](#) `power_model` = [WindPowerProductionModel](#) :: [WIND_POWER_EXPONENTIAL](#)
The wind power production model to be applied.

4.33.1 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 = 8
```

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

4.33.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

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

4.33.2.4 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel :: WIND_POWER_EXPONENTIAL
```

The wind power production model to be applied.

4.33.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.33.2.6 resource_key

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

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

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

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

Chapter 5

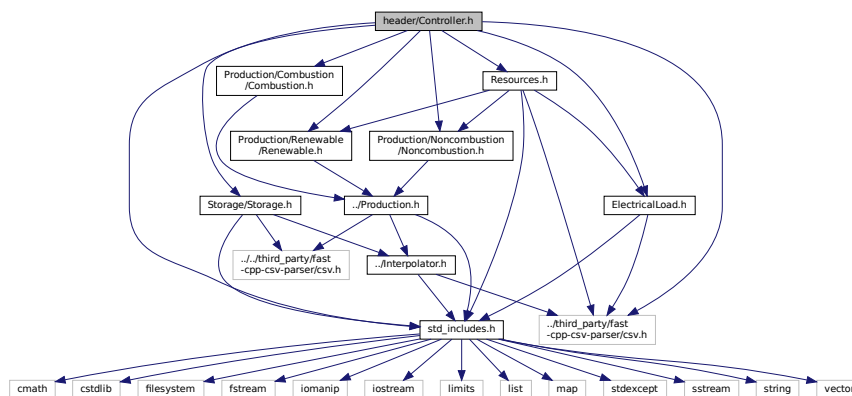
File Documentation

5.1 header/Controller.h File Reference

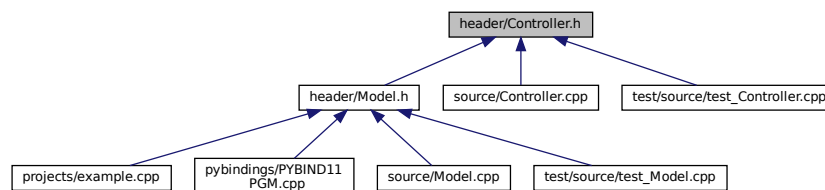
Header file for the [Controller](#) class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Combustion.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"
#include "Storage/Storage.h"
```

Include dependency graph for Controller.h:



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



Classes

- class [Controller](#)

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

Enumerations

- enum [ControlMode](#) { [LOAD_FOLLOWING](#) , [CYCLE_CHARGING](#) , [N_CONTROL_MODES](#) }

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

5.1.1 Detailed Description

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

5.1.2 Enumeration Type Documentation

5.1.2.1 ControlMode

enum [ControlMode](#)

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

Enumerator

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

```

44     {
45     LOAD\_FOLLOWING,
46     CYCLE\_CHARGING,
47     N\_CONTROL\_MODES
48     };

```

5.2 header/doxygen_cite.h File Reference

Header file which simply cites the doxygen tool.

5.2.1 Detailed Description

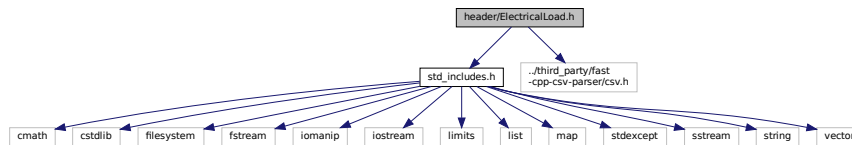
Header file which simply cites the doxygen tool.

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

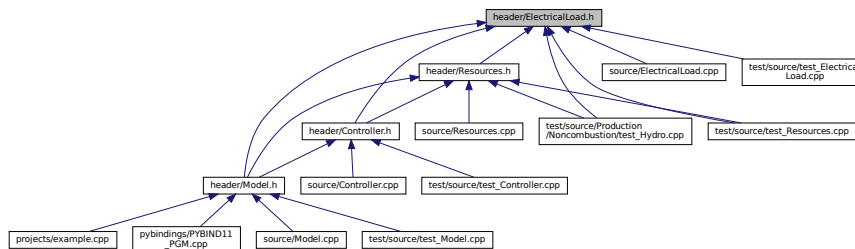
5.3 header/ElectricalLoad.h File Reference

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](#).

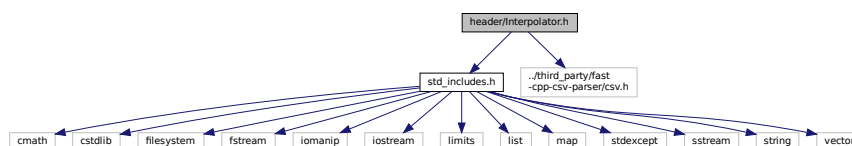
5.3.1 Detailed Description

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

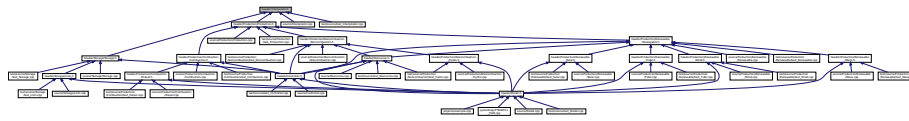
5.4 header/Interpolator.h File Reference

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

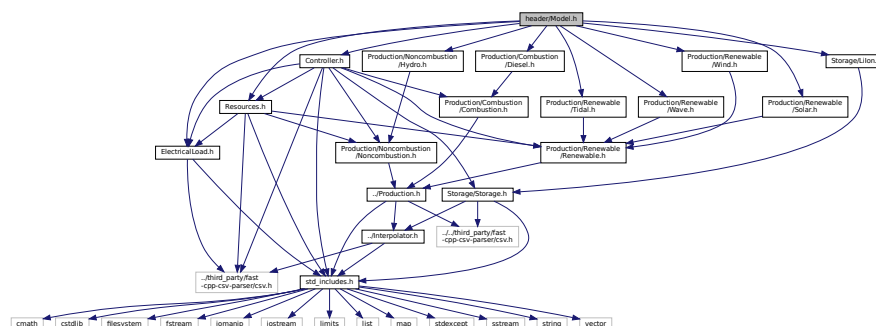
Header file for the [Interpolator](#) class.

5.5 header/Model.h File Reference

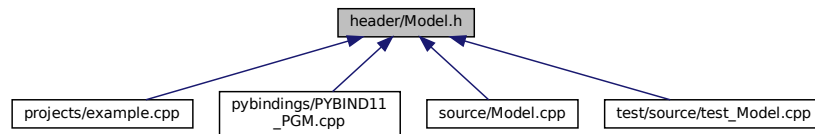
Header file for the [Model](#) class.

```
#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 [ModellInputs](#)

A structure which bundles the necessary inputs for the [Model](#) constructor. Provides default values for every necessary input (except `path_2_electrical_load_time_series`, for which a valid input must be provided).

- class [Model](#)

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

5.5.1 Detailed Description

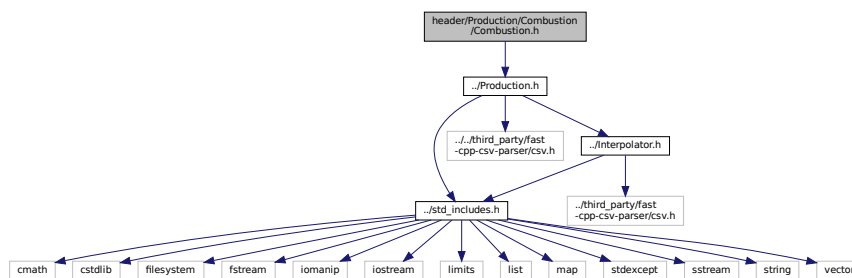
Header file for the [Model](#) class.

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

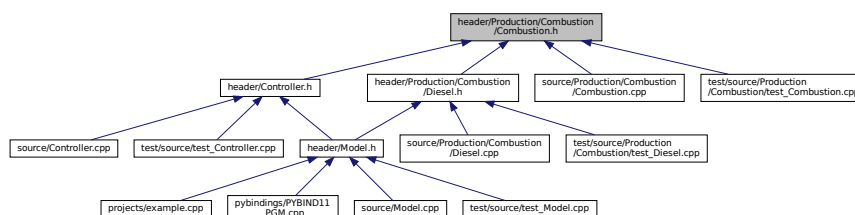
Header file for the [Combustion](#) class.

```
#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.

```

33         {
34     DIESEL,
35     N\_COMBUSTION\_TYPES
36 };

```

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.

```

46     {
47         FUEL_MODE_LINEAR,
48         FUEL_MODE_LOOKUP,
49         N_FUEL_MODES
50     };

```

5.7 header/Production/Combustion/Diesel.h File Reference

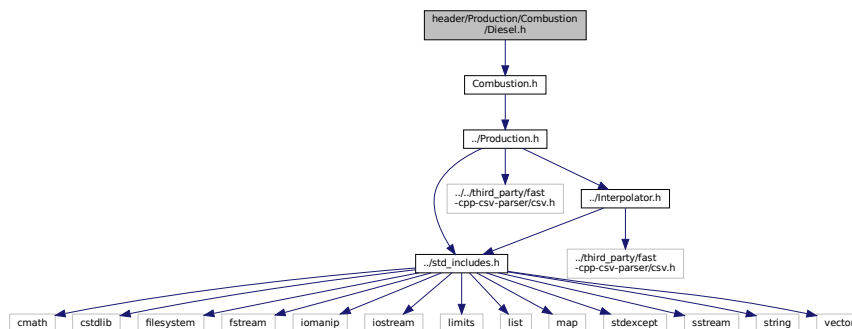
Header file for the [Diesel](#) class.

```

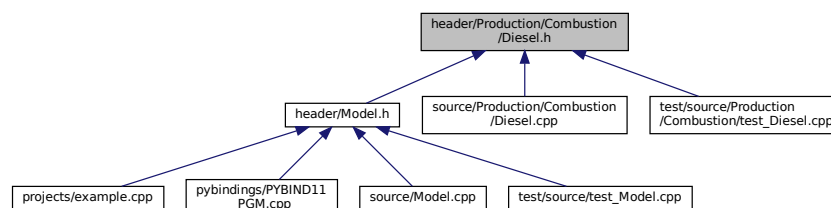
#include "Combustion.h"

```

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.

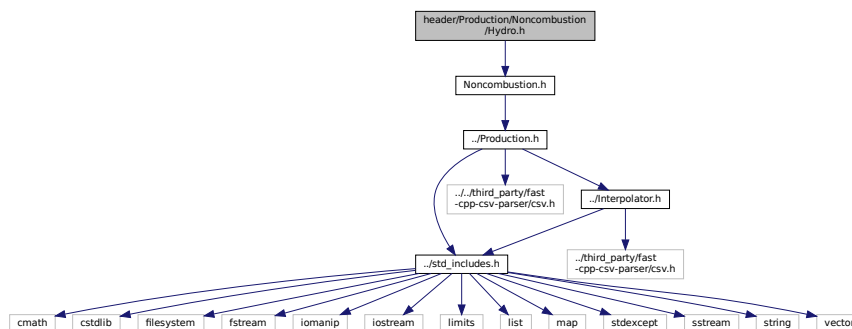
5.7.1 Detailed Description

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

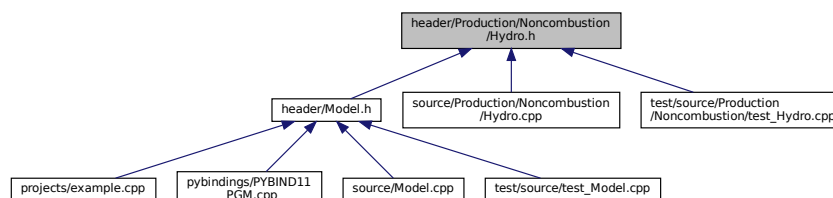
5.8 header/Production/Noncombustion/Hydro.h File Reference

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](#) , [N_HYDRO_TURBINES](#) }
- An enumeration of the types of hydroelectric turbine supported by PGMcpp.

5.8.1 Detailed Description

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

5.8.2 Enumeration Type Documentation

5.8.2.1 HydroTurbineType

enum [HydroTurbineType](#)

An enumeration of the types of hydroelectric turbine supported by PGMcpp.

Enumerator

HYDRO_TURBINE_PELTON	A Pelton turbine.
HYDRO_TURBINE_FRANCIS	A Francis turbine.
N_HYDRO_TURBINES	A simple hack to get the number of elements in HydroTurbineType.

```

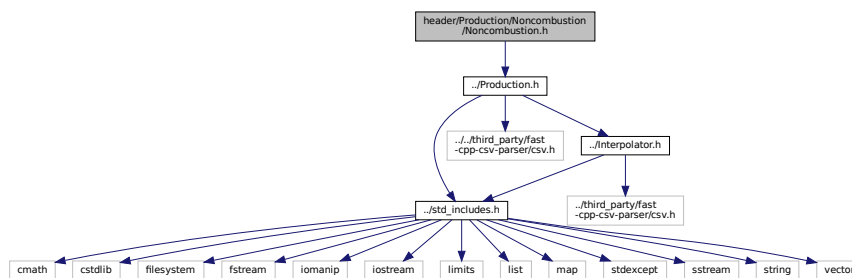
33     {
34     HYDRO_TURBINE_PELTON,
35     HYDRO_TURBINE_FRANCIS,
36     N_HYDRO_TURBINES
37 };

```

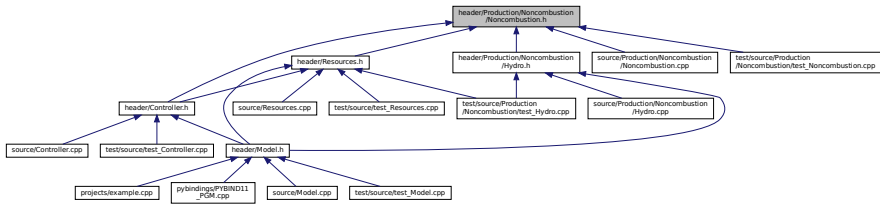
5.9 header/Production/Noncombustion/Noncombustion.h File Reference

```
#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

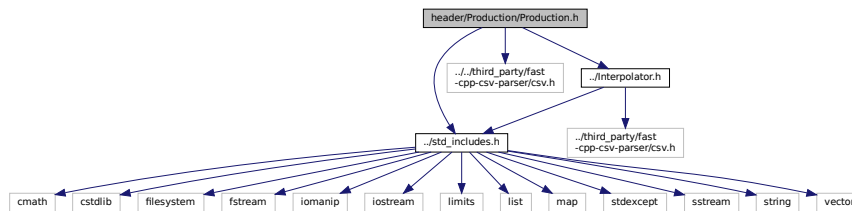
HYDRO	A hydroelectric generator (either with reservoir or not)
N_NONCOMBUSTION_TYPES	A simple hack to get the number of elements in NoncombustionType.

```
33         {
34     HYDRO,
35     N\_NONCOMBUSTION\_TYPES
36 };
```

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.

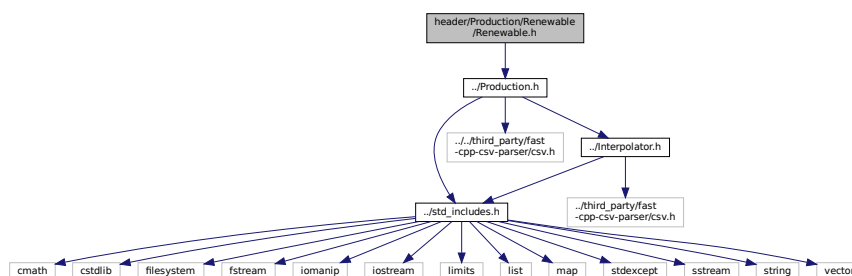
5.10.1 Detailed Description

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:
```



[illegible]

- struct RenewableInputs

- class Renewable

- enum RenewableType {
SOLAR , TIDAL , WAVE , WIND ,
N_RENEWABLE_TYPES }

5.11.2.1 RenewableType

Enumerator

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

```

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

```

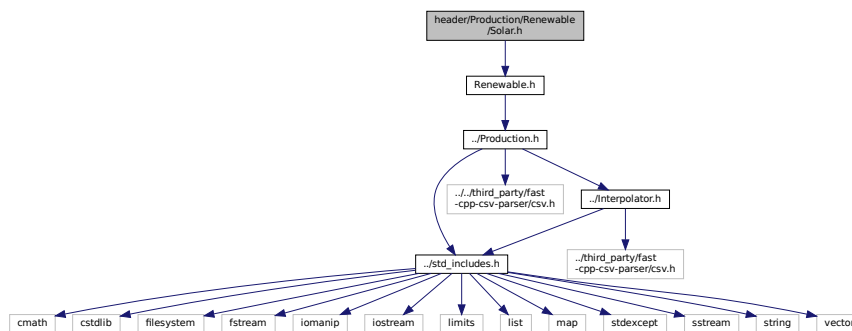
5.12 header/Production/Renewable/Solar.h File Reference

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

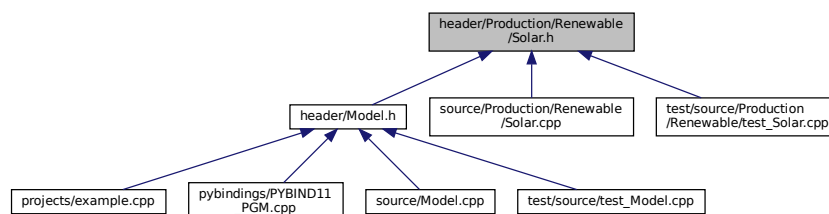
```
#include "Renewable.h"

```

Include dependency graph for Solar.h:



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



Classes

- struct [SolarInputs](#)

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

- class [Solar](#)

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

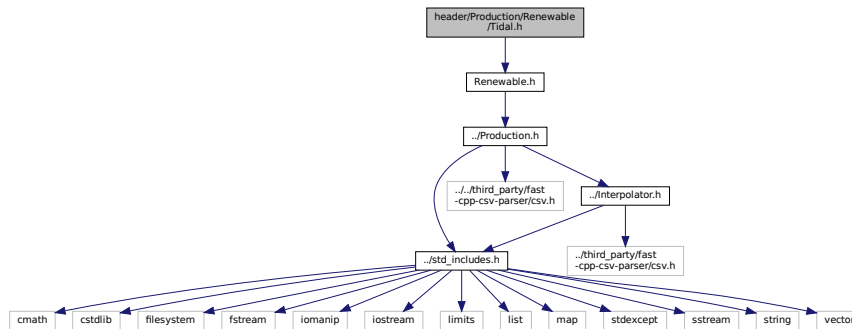
5.12.1 Detailed Description

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

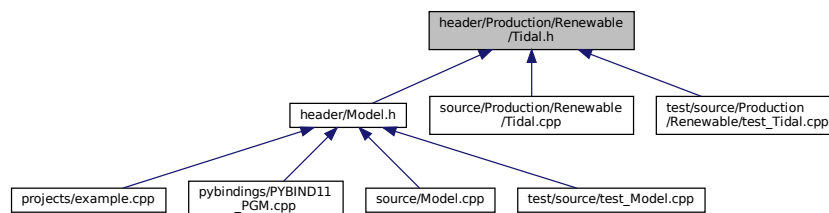
5.13 header/Production/Renewable/Tidal.h File Reference

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

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



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



Classes

- struct [TidalInputs](#)
A structure which bundles the necessary inputs for the [Tidal](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [RenewableInputs](#).
- class [Tidal](#)
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

Enumerations

- enum [TidalPowerProductionModel](#) { [TIDAL_POWER_CUBIC](#) , [TIDAL_POWER_EXPONENTIAL](#) , [TIDAL_POWER_LOOKUP](#) , [N_TIDAL_POWER_PRODUCTION_MODELS](#) }

5.13.1 Detailed Description

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

5.13.2 Enumeration Type Documentation

5.13.2.1 TidalPowerProductionModel

```
enum TidalPowerProductionModel
```

Enumerator

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

```

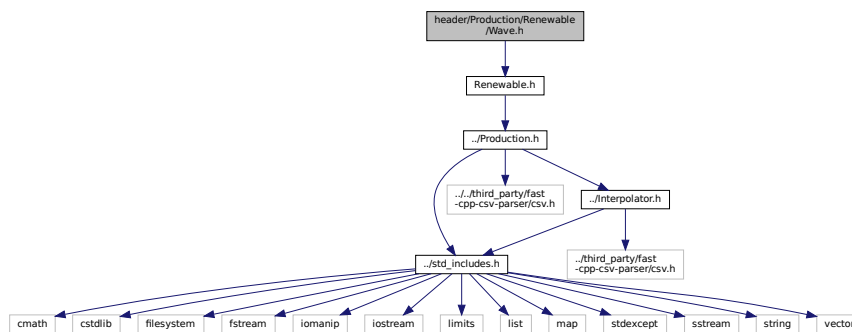
34
35     TIDAL_POWER_CUBIC,
36     TIDAL_POWER_EXPONENTIAL,
37     TIDAL_POWER_LOOKUP,
38     N_TIDAL_POWER_PRODUCTION_MODELS
39 };

```

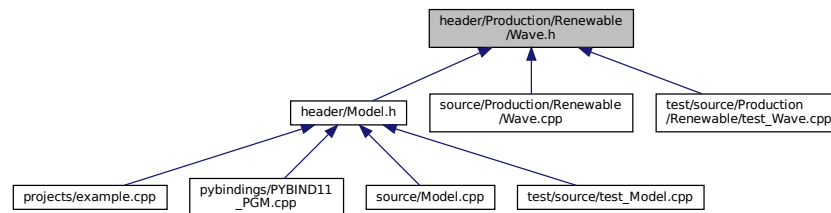
5.14 header/Production/Renewable/Wave.h File Reference

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 .

```

34     {
35         WAVE\_POWER\_GAUSSIAN,
```

```

36     WAVE_POWER_PARABOLOID,
37     WAVE_POWER_LOOKUP,
38     N_WAVE_POWER_PRODUCTION_MODELS
39 };

```

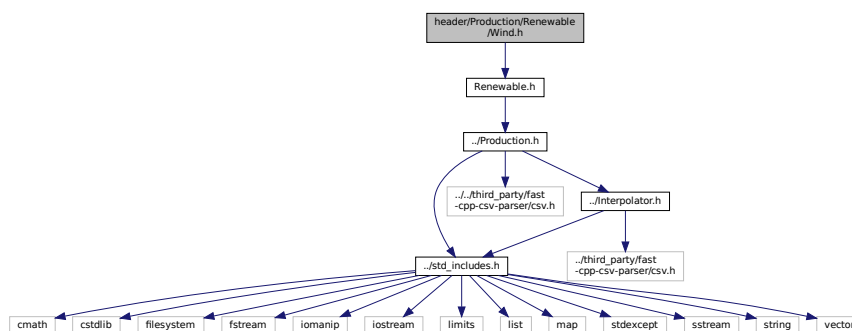
5.15 header/Production/Renewable/Wind.h File Reference

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

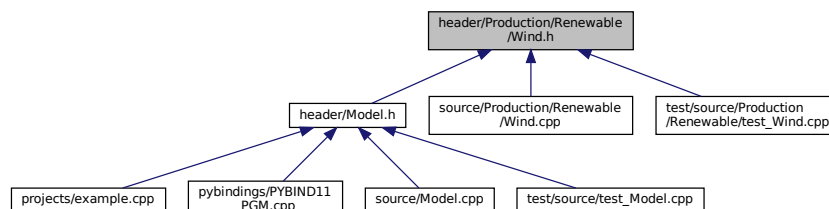
```
#include "Renewable.h"

```

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

```

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

```

5.16 header/Resources.h File Reference

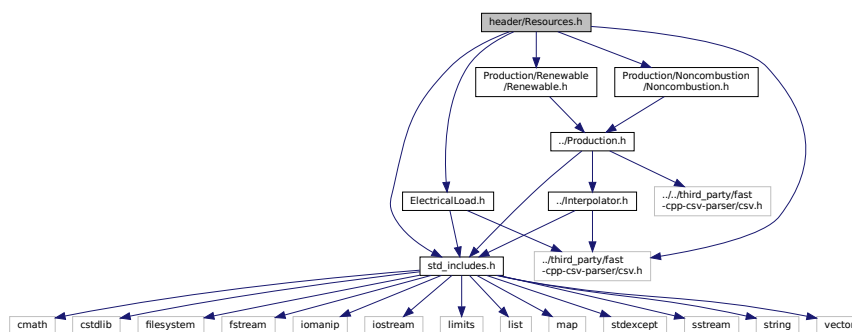
Header file for the [Resources](#) class.

```

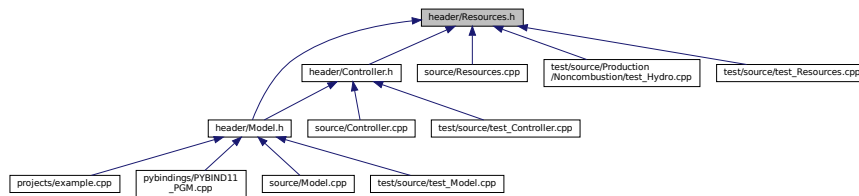
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"

```

Include dependency graph for Resources.h:



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



Classes

- class [Resources](#)

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

5.16.1 Detailed Description

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

5.17 header/std_includes.h File Reference

Header file which simply batches together some standard includes.

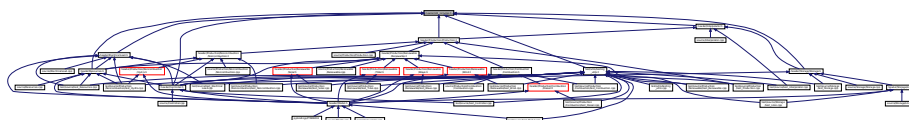
```

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

Include dependency graph for std_includes.h:



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



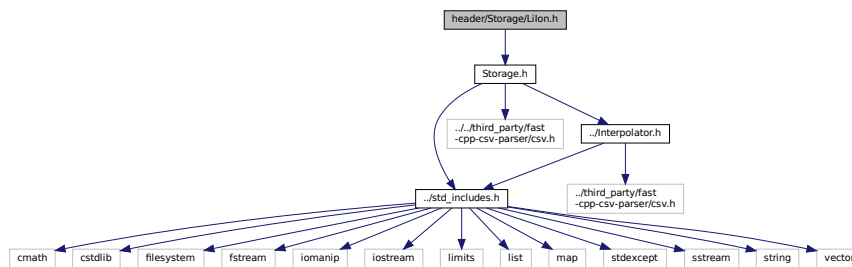
5.17.1 Detailed Description

Header file which simply batches together some standard includes.

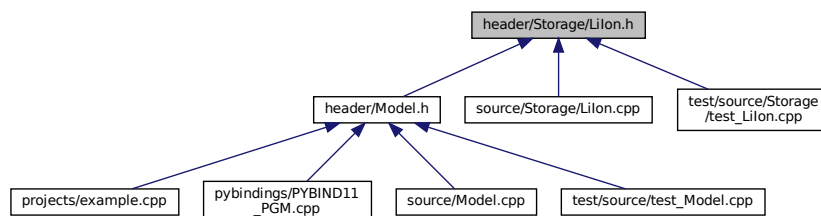
5.18 header/Storage/Lilon.h File Reference

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

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



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



Classes

- struct [LilonInputs](#)
A structure which bundles the necessary inputs for the [Lilon](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [StorageInputs](#).
- class [Lilon](#)
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

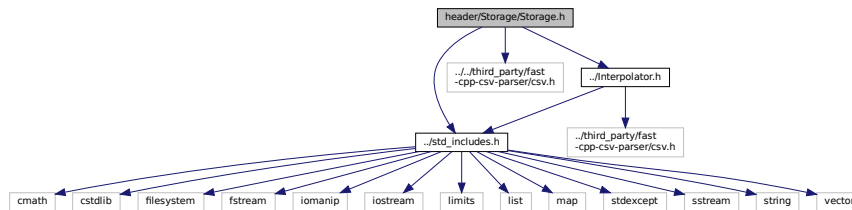
5.18.1 Detailed Description

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

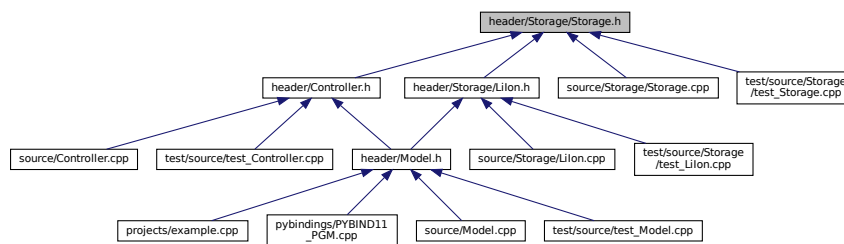
5.19 header/Storage/Storage.h File Reference

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

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



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



Classes

- struct [StorageInputs](#)
A structure which bundles the necessary inputs for the [Storage](#) constructor. Provides default values for every necessary input.
- class [Storage](#)
The base class of the [Storage](#) hierarchy. This hierarchy contains derived classes which model the storage of energy.

Enumerations

- enum [StorageType](#) { [LIION](#) , [N_STORAGE_TYPES](#) }
An enumeration of the types of [Storage](#) asset supported by PGMcpp.

5.19.1 Detailed Description

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

5.20.1.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     // 1. construct Model object
29     std::string path_2_electrical_load_time_series =
30         "data/test/electrical_load/electrical_load-generic_peak-500kW_1yr_dt-1hr.csv";
31
32     ModelInputs model_inputs;
33
34     model_inputs.path_2_electrical_load_time_series =
35         path_2_electrical_load_time_series;
36
37     model_inputs.control_mode = ControlMode :: CYCLE_CHARGING;
38
39     Model model(model_inputs);
40
41
42     // 2. add Diesel objects to Model
43     //     assume diesel generators are sunk assets (no initial capital cost)
44     DieselInputs diesel_inputs;
45
46     // 2.1. add 1 x 300 kW diesel generator (since mean load is ~250 kW)
47     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 300;    //<-- accessing and changing
48     //     an encapsulated structure attributed
49     diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
50
51     model.addDiesel(diesel_inputs);
52
53     // 2.2. add 2 x 150 kW diesel generators (since max load is 500 kW)
54     diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
55
56     model.addDiesel(diesel_inputs);
57     model.addDiesel(diesel_inputs);
58
59     // 3. add renewable resources to Model
60
61     // 3.1. add solar resource time series
62     int solar_resource_key = 0;
63     std::string path_2_solar_resource_data =
64         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
65
66     model.addResource (
67         RenewableType :: SOLAR,
68         path_2_solar_resource_data,
69         solar_resource_key
70     );
71
72     // 3.2. add tidal resource time series
73     int tidal_resource_key = 1;
74     std::string path_2_tidal_resource_data =
75         "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
76
77     model.addResource (
78         RenewableType :: TIDAL,
79         path_2_tidal_resource_data,
80         tidal_resource_key
81     );
82
83     // 3.3. add wave resource time series
84     int wave_resource_key = 2;
85     std::string path_2_wave_resource_data =
86         "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
87
88     model.addResource (
89         RenewableType :: WAVE,
90         path_2_wave_resource_data,
91         wave_resource_key
92     );
93
94     // 3.4. add wind resource time series
95     int wind_resource_key = 3;
96     std::string path_2_wind_resource_data =
97         "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
98
99     model.addResource (
100         RenewableType :: WIND,
101         path_2_wind_resource_data,
102         wind_resource_key
103     );
104
105

```

```

106 // 4. add Renewable objects to Model
107
108 // 4.1. add 1 x 250 kW solar PV array
109 SolarInputs solar_inputs;
110
111 solar_inputs.renewable_inputs.production_inputs.capacity_kW = 250;
112 solar_inputs.resource_key = solar_resource_key;
113
114 model.addSolar(solar_inputs);
115
116 // 4.2. add 1 x 120 kW tidal turbine
117 TidalInputs tidal_inputs;
118
119 tidal_inputs.renewable_inputs.production_inputs.capacity_kW = 120;
120 tidal_inputs.design_speed_ms = 2.5;
121 tidal_inputs.resource_key = tidal_resource_key;
122
123 model.addTidal(tidal_inputs);
124
125 // 4.3. add 1 x 150 kW wind turbine
126 WindInputs wind_inputs;
127
128 wind_inputs.renewable_inputs.production_inputs.capacity_kW = 150;
129 wind_inputs.resource_key = wind_resource_key;
130
131 model.addWind(wind_inputs);
132
133 // 4.4. add 1 x 100 kW wave energy converter
134 WaveInputs wave_inputs;
135
136 wave_inputs.renewable_inputs.production_inputs.capacity_kW = 100;
137 wave_inputs.resource_key = wave_resource_key;
138
139 model.addWave(wave_inputs);
140
141
142 // 5. add LiIon object to Model
143
144 // 5.1. add 1 x (500 kW, ) lithium ion battery energy storage system
145 LiIonInputs liion_inputs;
146
147 liion_inputs.storage_inputs.power_capacity_kW = 500;
148 liion_inputs.storage_inputs.energy_capacity_kWh = 1050; //<-- about 4 hours of mean load autonomy
149
150 model.addLiIon(liion_inputs);
151
152
153 // 6. run and write results
154 model.run();
155
156 model.writeResults("projects/example_cpp");
157
158 return 0;
159 } /* 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/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

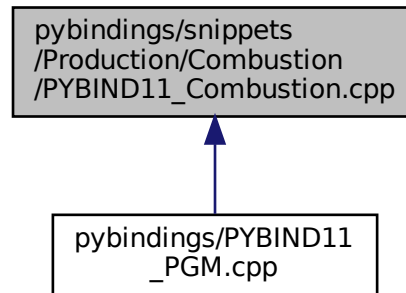
5.21.2.1 PYBIND11_MODULE()

```
PYBIND11_MODULE (
    PGMcpp ,
    m )
{
31
32
33     #include "snippets/PYBIND11_Controller.cpp"
34     #include "snippets/PYBIND11_ElectricalLoad.cpp"
35     #include "snippets/PYBIND11_Interpolator.cpp"
36     #include "snippets/PYBIND11_Model.cpp"
37     #include "snippets/PYBIND11_Resources.cpp"
38
39     #include "snippets/Production/PYBIND11_Production.cpp"
40
41     #include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
42     #include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
43
44     #include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
45     #include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
46     #include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
47     #include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
48     #include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
49
50     #include "snippets/Storage/PYBIND11_Storage.cpp"
51     #include "snippets/Storage/PYBIND11_LiIon.cpp"
52
53 } /* 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")

Variables

- [&CombustionInputs::production_inputs](#) [&CombustionInputs::nominal_fuel_escalation_annual](#) [def_readwrite](#)("path_2_fuel_interp_data", [&CombustionInputs::path_2_fuel_interp_data](#)) [.def](#)(pybind11 [&Emissions::CO2_kg](#) [def_readwrite](#) ("CO_kg", [&Emissions::CO_kg](#)) [.def_readwrite](#)("NOx_kg")

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_readwrite()

```
& CombustionInputs::production_inputs def_readwrite (
    "fuel_mode" ,
    &CombustionInputs::fuel_mode )
```

5.22.2.2 value() [1/2]

```
FuelMode::FUEL_MODE_LINEAR value (
    "FUEL_MODE_LOOKUP" ,
    FuelMode::FUEL_MODE_LOOKUP )
```

5.22.2.3 value() [2/2]

```
CombustionType::DIESEL value (
    "N_COMBUSTION_TYPES" ,
    CombustionType::N_COMBUSTION_TYPES )
```

5.22.3 Variable Documentation

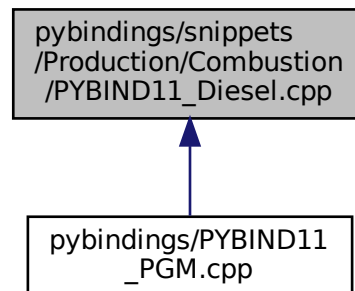
5.22.3.1 def_readwrite

```
&StorageInputs::print_flag &StorageInputs::power_capacity_kW &StorageInputs::nominal_inflation_annual
def_readwrite (
    "CO_kg" ,
    &Emissions::CO_kg )
```

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()

```
&InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D::
&InterpolatorStruct2D::z_matrix def (
    pybind11::init() )
```

5.23.2.2 def_readwrite() [1/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
& DieselInputs::SOx_emissions_intensity_kgL def_readwrite (
    "CH4_emissions_intensity_kgL" ,
    &DieselInputs::CH4_emissions_intensity_kgL )
```

5.23.2.3 def_readwrite() [2/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh def_readwrite (
    "CO2_emissions_intensity_kgL" ,
    &DieselInputs::CO2_emissions_intensity_kgL )
```

5.23.2.4 def_readwrite() [3/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs def_readwrite (
    "linear_fuel_slope_LkWh" ,
    &DieselInputs::linear_fuel_slope_LkWh )
```

5.23.2.5 def_readwrite() [4/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L
def_readwrite (
    "minimum_load_ratio" ,
    &DieselInputs::minimum_load_ratio )
```


5.23.2.6 def_readwrite() [5/8]

```
& Diesel::minimum_load_ratio def_readwrite (
    "minimum_runtime_hrs" ,
    & Diesel::minimum_runtime_hrs )
```

5.23.2.7 def_readwrite() [6/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost & DieselInputs::fuel_cost_L &
DieselInputs::minimum_runtime_hrs & DieselInputs::linear_fuel_intercept_LkWh & DieselInputs::CO_emissions_inte
def_readwrite (
    "NOx_emissions_intensity_kgL" ,
    & DieselInputs::NOx_emissions_intensity_kgL )
```

5.23.2.8 def_readwrite() [7/8]

```
& DieselInputs::combustion_inputs & DieselInputs::capital_cost def_readwrite (
    "operation_maintenance_cost_kWh" ,
    & DieselInputs::operation_maintenance_cost_kWh )
```

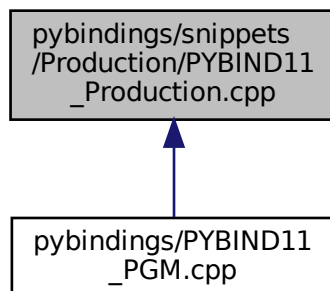
5.23.2.9 def_readwrite() [8/8]

```
& DieselInputs::combustion_inputs def_readwrite (
    "replace_running_hrs" ,
    & DieselInputs::replace_running_hrs )
```

5.24 pybindings/snippets/Production/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"`

Variables

- `&ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual def_readwrite("replace_running_hrs", &ProductionInputs::replace_running_hrs) .def(pybind11 &Production::interpolator def_readwrite ("print_flag", &Production::print_flag) .def_readwrite("is_running"`

5.24.1 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!

5.24.2 Function Documentation

5.24.2.1 `def_readwrite()` [1/2]

```
& ProductionInputs::print_flag def_readwrite (
    "is_sunk" ,
    &ProductionInputs::is_sunk )
```

5.24.2.2 `def_readwrite()` [2/2]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW def_readwrite (
    "nominal_inflation_annual" ,
    &ProductionInputs::nominal_inflation_annual )
```

5.24.3 Variable Documentation

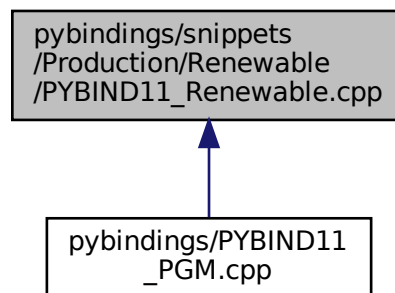
5.24.3.1 def_readwrite

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
def_readwrite ("replace_running_hrs", &ProductionInputs::replace_running_hrs) .def(pybind11 &
Production::interpolator & Production::is_running & Production::n_points & Production::n_replacements
& Production::running_hours & Production::capacity_kW & Production::nominal_discount_annual &
Production::capital_cost & Production::net_present_cost & Production::levellized_cost_of_energy_kWh
& Production::is_running_vec & Production::dispatch_vec_kW & Production::curtailment_vec_kW
def_readwrite("capital_cost_vec", &Production::capital_cost_vec) .def_readwrite("operation_↵
maintenance_cost_vec" (
    "print_flag" ,
    &Production::print_flag )
```

5.25 pybindings/snippets/Production/Renewable/PYBIND11_↵ Renewable.cpp File Reference

Bindings file for the [Renewable](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [RenewableType::SOLAR](#) value ("TIDAL", RenewableType::TIDAL) .value("WAVE"
- [RenewableType::SOLAR](#) [RenewableType::WAVE](#) value ("WIND", RenewableType::WIND) .value("N_↵ RENEWABLE_TYPES"
- &[RenewableInputs::production_inputs](#) def (pybind11::init())

5.25.1 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!

5.25.2 Function Documentation

5.25.2.1 def()

```
& RenewableInputs::production_inputs def (  
    pybind11::init() )
```

5.25.2.2 value() [1/2]

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

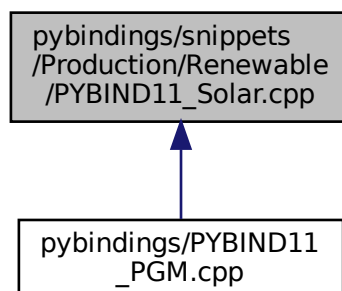
5.25.2.3 value() [2/2]

```
RenewableType::SOLAR RenewableType::WAVE value (  
    "WIND" ,  
    RenewableType::WIND )
```

5.26 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference

Bindings file for the [Solar](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- `&SolarInputs::renewable_inputs def_readwrite ("resource_key", &SolarInputs::resource_key) .def_readwrite("capital_cost"`
- `&SolarInputs::renewable_inputs &SolarInputs::capital_cost def_readwrite ("operation_maintenance_cost_kWh", &SolarInputs::operation_maintenance_cost_kWh) .def_readwrite("derating"`
- `&SolarInputs::renewable_inputs &SolarInputs::capital_cost &SolarInputs::derating def (pybind11::init())`

5.26.1 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!

5.26.2 Function Documentation

5.26.2.1 `def()`

```
& SolarInputs::renewable_inputs & SolarInputs::capital_cost & SolarInputs::derating def (
    pybind11::init() )
```

5.26.2.2 `def_readwrite()` [1/2]

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

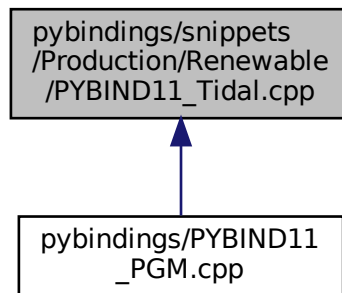
5.26.2.3 `def_readwrite()` [2/2]

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

5.27 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference

Bindings file for the [Tidal](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [TidalPowerProductionModel::TIDAL_POWER_CUBIC](#) [value](#) ("TIDAL_POWER_EXPONENTIAL", [TidalPowerProductionModel::TIDAL_POWER_EXPONENTIAL](#)) [.value\("TIDAL_POWER_LOOKUP"](#)
- [TidalPowerProductionModel::TIDAL_POWER_CUBIC](#) [TidalPowerProductionModel::TIDAL_POWER_LOOKUP](#) [value](#) ("N_TIDAL_POWER_PRODUCTION_MODELS", [TidalPowerProductionModel::N_TIDAL_POWER_PRODUCTION_MODELS](#))
- [&TidalInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", [&TidalInputs::resource_key](#)) [.def_readwrite\("capital_cost"](#)
- [&TidalInputs::renewable_inputs](#) [&TidalInputs::capital_cost](#) [def_readwrite](#) ("operation_maintenance_cost_kWh", [&TidalInputs::operation_maintenance_cost_kWh](#)) [.def_readwrite\("design_speed_ms"](#)

Variables

- [&TidalInputs::renewable_inputs](#) [&TidalInputs::capital_cost](#) [&TidalInputs::design_speed_ms](#) [def_readwrite\("power_model"](#), [&TidalInputs::power_model](#)) [.def\(pybind11 &Tidal::design_speed_ms](#) [def_readwrite](#) ("power_model", [&Tidal::power_model](#)) [.def_readwrite\("power_model_string"](#)

5.27.1 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!

5.27.2 Function Documentation

5.27.2.1 def_readwrite() [1/2]

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

5.27.2.2 def_readwrite() [2/2]

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

5.27.2.3 value() [1/2]

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

5.27.2.4 value() [2/2]

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

5.27.3 Variable Documentation

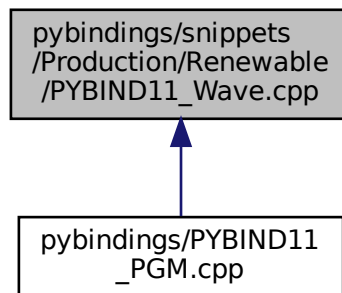
5.27.3.1 def_readwrite

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost & TidalInputs::design_speed_ms
def_readwrite ("power_model", &TidalInputs::power_model) .def(pybind11 & Tidal::design_speed_ms
def_readwrite("power_model", &Tidal::power_model) .def_readwrite("power_model_string" (
    "power_model" ,
    &Tidal::power_model )
```

5.28 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference

Bindings file for the [Wave](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [WavePowerProductionModel::WAVE_POWER_GAUSSIAN](#) [value](#) ("WAVE_POWER_PARABOLOID", WavePowerProductionModel::WAVE_POWER_PARABOLOID) .value("WAVE_POWER_LOOKUP"
- [WavePowerProductionModel::WAVE_POWER_GAUSSIAN](#) [WavePowerProductionModel::WAVE_POWER_LOOKUP](#) [value](#) ("N_WAVE_POWER_PRODUCTION_MODELS", WavePowerProductionModel::N_WAVE_POWER_PRODUCTION_MODELS)
- [&WaveInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", &WaveInputs::resource_key) .def_readwrite("capital_cost"
- [&WaveInputs::renewable_inputs](#) [&WaveInputs::capital_cost](#) [def_readwrite](#) ("operation_maintenance_cost_kWh", &WaveInputs::operation_maintenance_cost_kWh) .def_readwrite("design_significant_wave_height_m"
- [&WaveInputs::renewable_inputs](#) [&WaveInputs::capital_cost](#) [&WaveInputs::design_significant_wave_height_m](#) [def_readwrite](#) ("design_energy_period_s", &WaveInputs::design_energy_period_s) .def_readwrite("power_model"

Variables

- [&WaveInputs::renewable_inputs](#) [&WaveInputs::capital_cost](#) [&WaveInputs::design_significant_wave_height_m](#) [&WaveInputs::power_model](#) [def_readwrite](#) ("path_2_normalized_performance_matrix", &WaveInputs::path_2_normalized_performance_matrix) .def(pybind11 &Wave::design_significant_wave_height_m [def_readwrite](#) ("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power_model"

5.28.1 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!

5.28.2 Function Documentation

5.28.2.1 `def_readwrite()` [1/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
def_readwrite (
    "design_energy_period_s" ,
    &WaveInputs::design_energy_period_s )
```

5.28.2.2 `def_readwrite()` [2/3]

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

5.28.2.3 `def_readwrite()` [3/3]

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

5.28.2.4 `value()` [1/2]

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

5.28.2.5 `value()` [2/2]

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

5.28.3 Variable Documentation

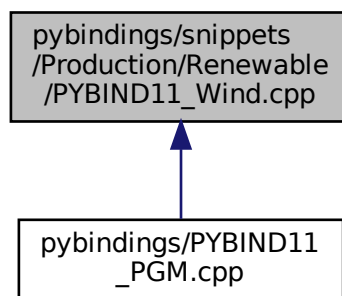
5.28.3.1 def_readwrite

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
& WaveInputs::power_model def_readwrite ( "path_2_normalized_performance_matrix", &Wave←
Inputs::path_2_normalized_performance_matrix ) .def(pybind11 & Wave::design_significant_wave_height_m
def_readwrite("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power←
_model" (
    "design_energy_period_s" ,
    &Wave::design_energy_period_s )
```

5.29 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference

Bindings file for the [Wind](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [WindPowerProductionModel::WIND_POWER_EXPONENTIAL](#) value ("WIND_POWER_LOOKUP", Wind←
PowerProductionModel::WIND_POWER_LOOKUP) .value("N_WIND_POWER_PRODUCTION_MODELS"
- [&WindInputs::renewable_inputs](#) [def_readwrite](#) ("resource_key", &WindInputs::resource_key) .def_←
readwrite("capital_cost"
- [&WindInputs::renewable_inputs](#) [&WindInputs::capital_cost](#) [def_readwrite](#) ("operation_maintenance_cost_←
kWh", &WindInputs::operation_maintenance_cost_kWh) .def_readwrite("design_speed_ms"

Variables

- [&WindInputs::renewable_inputs](#) [&WindInputs::capital_cost](#) [&WindInputs::design_speed_ms](#) [def_←](#)
readwrite("power_model", &WindInputs::power_model) .def(pybind11 &Wind::design_speed_ms [def_readwrite](#)
("power_model", &Wind::power_model) .def_readwrite("power_model_string"

5.29.1 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!

5.29.2 Function Documentation

5.29.2.1 `def_readwrite()` [1/2]

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

5.29.2.2 `def_readwrite()` [2/2]

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

5.29.2.3 `value()`

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

5.29.3 Variable Documentation

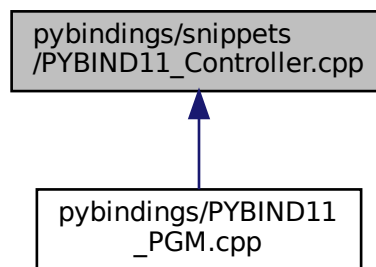
5.29.3.1 `def_readwrite`

```
& WindInputs::renewable_inputs & WindInputs::capital_cost & WindInputs::design_speed_ms def↵
_readwrite ("power_model", &WindInputs::power_model) .def(pybind11 & Wind::design_speed_ms
def_readwrite("power_model", &Wind::power_model) .def_readwrite("power_model_string" (
    "power_model" ,
    &Wind::power_model )
```

5.30 pybindings/snippets/PYBIND11_Controller.cpp File Reference

Bindings file for the [Controller](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Functions

- [ControlMode::LOAD_FOLLOWING](#) [value](#) ("CYCLE_CHARGING", [ControlMode::CYCLE_CHARGING](#))
[.value\("N_CONTROL_MODES"](#)
- [&Controller::control_mode](#) [def_readwrite](#) ("control_string", [&Controller::control_string](#)) [.def_readwrite\("net↔
_load_vec_kW"](#)
- [&Controller::control_mode](#) [&Controller::net_load_vec_kW](#) [def_readwrite](#) ("missed_load_vec_kW", [&Controller↔
::missed_load_vec_kW](#)) [.def_readwrite\("combustion_map"](#)
- [&Controller::control_mode](#) [&Controller::net_load_vec_kW](#) [&Controller::combustion_map](#) [def](#) ([pybind11↔
::init<>\(\)](#)) [.def\("setControlMode"](#)
- [&Controller::control_mode](#) [&Controller::net_load_vec_kW](#) [&Controller::combustion_map](#) [&Controller::setControlMode](#)
[def](#) ("init", [&Controller::init](#)) [.def\("applyDispatchControl"](#)
- [&Controller::control_mode](#) [&Controller::net_load_vec_kW](#) [&Controller::combustion_map](#) [&Controller::setControlMode](#)
[&Controller::applyDispatchControl](#) [def](#) ("clear", [&Controller::clear](#))

5.30.1 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!

5.30.2 Function Documentation

5.30.2.1 def() [1/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
& Controller::applyDispatchControl def (
    "clear" ,
    &Controller::clear )
```

5.30.2.2 def() [2/3]

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

5.30.2.3 def() [3/3]

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

5.30.2.4 def_readwrite() [1/2]

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

5.30.2.5 def_readwrite() [2/2]

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

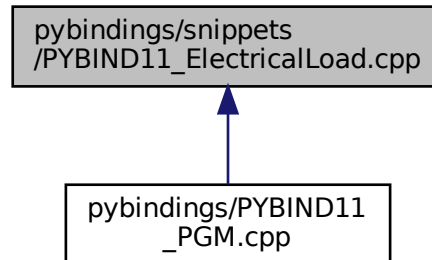
5.30.2.6 value()

```
ControlMode::LOAD_FOLLOWING value (
    "CYCLE_CHARGING" ,
    ControlMode::CYCLE_CHARGING )
```

5.31 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"

5.31.1 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!

5.31.2 Function Documentation

5.31.2.1 `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 )
  
```

5.31.2.2 `def_readwrite()` [2/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW def_readwrite (
    "mean_load_kW" ,
    &ElectricalLoad::mean_load_kW )
```

5.31.2.3 `def_readwrite()` [3/4]

```
& ElectricalLoad::n_points def_readwrite (
    "n_years" ,
    &ElectricalLoad::n_years )
```

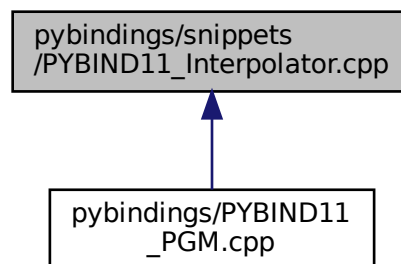
5.31.2.4 `def_readwrite()` [4/4]

```
& 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 )
```

5.32 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"`

5.32.1 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!

5.32.2 Function Documentation

5.32.2.1 `def()`

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x & InterpolatorStruct1D::y_vec
def (
    pybind11::init() )
```

5.32.2.2 `def_readwrite()` [1/7]

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x def_readwrite (
    "max_x" ,
    &InterpolatorStruct1D::max_x )
```


5.32.2.3 def_readwrite() [2/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x &
InterpolatorStruct2D::min_y def_readwrite (
    "max_y" ,
    &InterpolatorStruct2D::max_y )
```

5.32.2.4 def_readwrite() [3/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec def_readwrite (
    "min_x" ,
    &InterpolatorStruct2D::min_x )
```

5.32.2.5 def_readwrite() [4/7]

```
& InterpolatorStruct2D::n_rows def_readwrite (
    "n_cols" ,
    &InterpolatorStruct2D::n_cols )
```

5.32.2.6 def_readwrite() [5/7]

```
& Interpolator::interp_map_1D def_readwrite (
    "path_map_1D" ,
    &Interpolator::path_map_1D )
```

5.32.2.7 def_readwrite() [6/7]

```
& InterpolatorStruct1D::n_points def_readwrite (
    "x_vec" ,
    &InterpolatorStruct1D::x_vec )
```

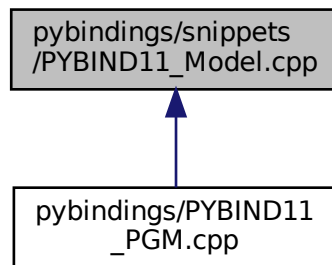
5.32.2.8 def_readwrite() [7/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x
def_readwrite (
    "y_vec" ,
    &InterpolatorStruct2D::y_vec )
```

5.33 pybindings/snippets/PYBIND11_Model.cpp File Reference

Bindings file for the [Model](#) class. Intended to be #include'd in [PYBIND11_PGM.cpp](#).

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



Variables

- [&ModelInputs::path_2_electrical_load_time_series](#) `def_readwrite("control_mode", &ModelInputs::control_mode)` `.def(pybind11 &Model::total_fuel_consumed_L` `def_readwrite ("total_emissions", &Model::total_emissions)` `.def_readwrite("net_present_cost"`

5.33.1 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!

5.33.2 Variable Documentation

5.33.2.1 def_readwrite

```

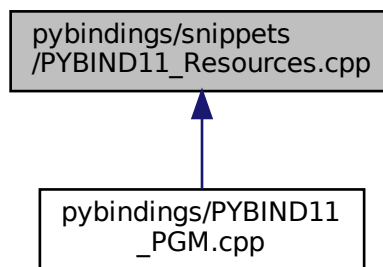
& ModelInputs::path\_2\_electrical\_load\_time\_series def_readwrite ("control_mode", &ModelInputs::control_mode)
.def(pybind11 & Model::total\_fuel\_consumed\_L & Model::net\_present\_cost
& Model::levellized\_cost\_of\_energy\_kWh & Model::electrical\_load & Model::combustion\_ptr\_vec
def_readwrite("renewable_ptr_vec", &Model::renewable_ptr_vec) .def_readwrite("storage_ptr_vec"
(
    "total_emissions" ,
    &Model::total_emissions )

```

5.34 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"

5.34.1 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!

5.34.2 Function Documentation

5.34.2.1 [def_readwrite\(\)](#) [1/2]

```

& Resources::resource_map_1D & Resources::path_map_1D def_readwrite (
    "resource_map_2D" ,
    &Resources::resource_map_2D )
  
```

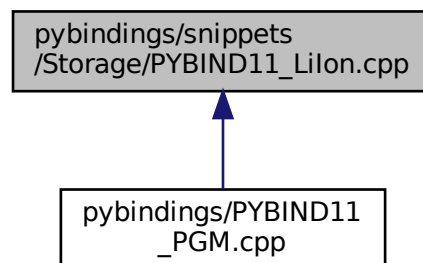
5.34.2.2 def_readwrite() [2/2]

```
& Resources::resource_map_1D def_readwrite (
    "string_map_1D" ,
    &Resources::string_map_1D )
```

5.35 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("degradation_alpha"`
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::degradation_alpha def_readwrite ("degradation_beta", &LilonInputs::degradation_beta) .def_readwrite("degradation_B_hat_cal_0"`
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::degradation_alpha &LilonInputs::degradation_B_hat_cal_0 def_readwrite ("degradation_r_cal", &LilonInputs::degradation_r_cal) .def_readwrite("degradation_Ea_cal_0"`
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::degradation_alpha &LilonInputs::degradation_B_hat_cal_0 &LilonInputs::degradation_Ea_cal_0 def_readwrite ("degradation_a_cal", &LilonInputs::degradation_a_cal) .def_readwrite("degradation_s_cal"`

- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::degradation_alpha &LilonInputs::degradation_B_hat_cal_0 &LilonInputs::degradation_Ea_cal_0 &LilonInputs::degradation_s_cal` `def_readwrite` ("gas_constant_JmolK", `&LilonInputs::gas_constant_JmolK`) `.def_readwrite` ("gas_constant_JmolK")
- `&LilonInputs::storage_inputs &LilonInputs::operation_maintenance_cost_kWh &LilonInputs::min_SOC &LilonInputs::max_SOC &LilonInputs::discharging_efficiency &LilonInputs::degradation_alpha &LilonInputs::degradation_B_hat_cal_0 &LilonInputs::degradation_Ea_cal_0 &LilonInputs::degradation_s_cal &LilonInputs::gas_constant_JmolK` `def` (`pybind11::init()`)
- `&Lilon::dynamic_energy_capacity_kWh` `def_readwrite` ("SOH", `&Lilon::SOH`) `.def_readwrite` ("replace_SOH")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH` `def_readwrite` ("degradation_alpha", `&Lilon::degradation_alpha`) `.def_readwrite` ("degradation_beta")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta` `def_readwrite` ("degradation_B_hat_cal_0", `&Lilon::degradation_B_hat_cal_0`) `.def_readwrite` ("degradation_r_cal")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal` `def_readwrite` ("degradation_Ea_cal_0", `&Lilon::degradation_Ea_cal_0`) `.def_readwrite` ("degradation_a_cal")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal &Lilon::degradation_a_cal` `def_readwrite` ("degradation_s_cal", `&Lilon::degradation_s_cal`) `.def_readwrite` ("gas_constant_JmolK")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal &Lilon::degradation_a_cal &Lilon::gas_constant_JmolK` `def_readwrite` ("temperature_K", `&Lilon::temperature_K`) `.def_readwrite` ("init_SOC")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal &Lilon::degradation_a_cal &Lilon::gas_constant_JmolK &Lilon::init_SOC` `def_readwrite` ("min_SOC", `&Lilon::min_SOC`) `.def_readwrite` ("hysteresis_SOC")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal &Lilon::degradation_a_cal &Lilon::gas_constant_JmolK &Lilon::init_SOC &Lilon::hysteresis_SOC` `def_readwrite` ("max_SOC", `&Lilon::max_SOC`) `.def_readwrite` ("charging_efficiency")
- `&Lilon::dynamic_energy_capacity_kWh &Lilon::replace_SOH &Lilon::degradation_beta &Lilon::degradation_r_cal &Lilon::degradation_a_cal &Lilon::gas_constant_JmolK &Lilon::init_SOC &Lilon::hysteresis_SOC &Lilon::charging_efficiency` `def_readwrite` ("discharging_efficiency", `&Lilon::discharging_efficiency`) `.def_readwrite` ("SOH_vec")

5.35.1 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!

5.35.2 Function Documentation

5.35.2.1 `def()`

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::degradation_alpha
& LiIonInputs::degradation_B_hat_cal_0 & LiIonInputs::degradation_Ea_cal_0 & LiIonInputs::degradation_s_cal
& LiIonInputs::gas_constant_JmolK def (
    pybind11::init() )
```

5.35.2.2 def_readwrite() [1/18]

```
& LiIonInputs::storage_inputs def_readwrite (
    "capital_cost" ,
    &LiIonInputs::capital_cost )
```

5.35.2.3 def_readwrite() [2/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC def_readwrite (
    "charging_efficiency" ,
    &LiIonInputs::charging_efficiency )
```

5.35.2.4 def_readwrite() [3/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::degradation_alpha
& LiIonInputs::degradation_B_hat_cal_0 & LiIonInputs::degradation_Ea_cal_0 def_readwrite (
    "degradation_a_cal" ,
    &LiIonInputs::degradation_a_cal )
```

5.35.2.5 def_readwrite() [4/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH def_readwrite (
    "degradation_alpha" ,
    &LiIon::degradation_alpha )
```

5.35.2.6 def_readwrite() [5/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta def_↔
readwrite (
    "degradation_B_hat_cal_0" ,
    &LiIon::degradation_B_hat_cal_0 )
```

5.35.2.7 def_readwrite() [6/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::degradation_alpha
def_readwrite (
    "degradation_beta" ,
    &LiIonInputs::degradation_beta )
```

5.35.2.8 def_readwrite() [7/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal
def_readwrite (
    "degradation_Ea_cal_0" ,
    &LiIon::degradation_Ea_cal_0 )
```

5.35.2.9 def_readwrite() [8/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::degradation_alpha
& LiIonInputs::degradation_B_hat_cal_0 def_readwrite (
    "degradation_r_cal" ,
    &LiIonInputs::degradation_r_cal )
```

5.35.2.10 def_readwrite() [9/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal
& LiIon::degradation_a_cal def_readwrite (
    "degradation_s_cal" ,
    &LiIon::degradation_s_cal )
```

5.35.2.11 def_readwrite() [10/18]

```
& LiIon::dynamic_energy_capacity_kWh & 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 )
```

5.35.2.12 def_readwrite() [11/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency & LiIonInputs::degradation_alpha
& LiIonInputs::degradation_B_hat_cal_0 & LiIonInputs::degradation_Ea_cal_0 & LiIonInputs::degradation_s_cal
def_readwrite (
    "gas_constant_JmolK" ,
    &LiIonInputs::gas_constant_JmolK )
```

5.35.2.13 def_readwrite() [12/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
def_readwrite (
    "hysteresis_SOC" ,
    &LiIonInputs::hysteresis_SOC )
```

5.35.2.14 def_readwrite() [13/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh def_readwrite (
    "init_SOC" ,
    &LiIonInputs::init_SOC )
```

5.35.2.15 def_readwrite() [14/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal
& LiIon::degradation_a_cal & LiIon::gas_constant_JmolK & LiIon::init_SOC & LiIon::hysteresis_SOC
def_readwrite (
    "max_SOC" ,
    &LiIon::max_SOC )
```

5.35.2.16 def_readwrite() [15/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal
& LiIon::degradation_a_cal & LiIon::gas_constant_JmolK & LiIon::init_SOC def_readwrite (
    "min_SOC" ,
    &LiIon::min_SOC )
```

5.35.2.17 def_readwrite() [16/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
& LiIonInputs::max_SOC & LiIonInputs::discharging_efficiency def_readwrite (
    "replace_SOH" ,
    &LiIonInputs::replace_SOH )
```

5.35.2.18 def_readwrite() [17/18]

```
& LiIon::dynamic_energy_capacity_kWh def_readwrite (
    "SOH" ,
    &LiIon::SOH )
```

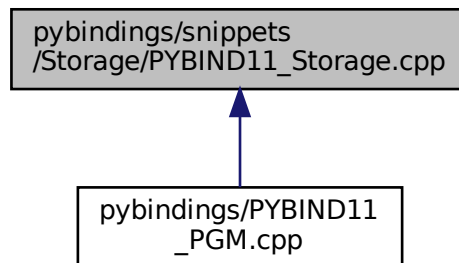

5.35.2.19 `def_readwrite()` [18/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH & LiIon::degradation_beta & LiIon::degradation_r_cal
& LiIon::degradation_a_cal & LiIon::gas_constant_JmolK def_readwrite (
    "temperature_K" ,
    &LiIon::temperature_K )
```

5.36 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")

5.36.1 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!

5.36.2 Function Documentation

5.36.2.1 `def_readwrite()` [1/2]

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW def_readwrite (
    "energy_capacity_kWh" ,
    &StorageInputs::energy_capacity_kWh )
```

5.36.2.2 `def_readwrite()` [2/2]

```
& StorageInputs::print_flag def_readwrite (
    "is_sunk" ,
    &StorageInputs::is_sunk )
```

5.36.2.3 `value()`

```
StorageType::LIION value (
    "N_STORAGE_TYPES" ,
    StorageType::N_STORAGE_TYPES )
```

5.36.3 Variable Documentation

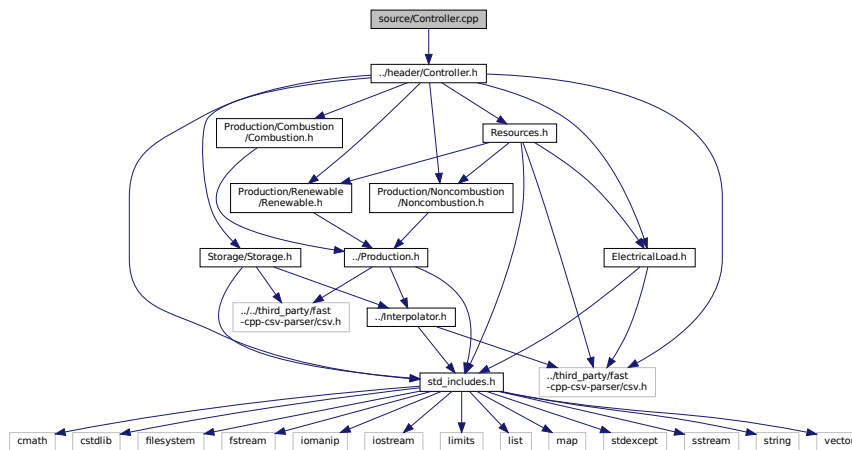
5.36.3.1 `def_readwrite`

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW & StorageInputs::nominal_inflation_annual
def_readwrite ("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11
& Storage::type & Storage::print_flag & Storage::is_sunk & Storage::n_replacements & Storage::power_capacity_k
& Storage::charge_kWh & Storage::nominal_inflation_annual & Storage::real_discount_annual &
Storage::operation_maintenance_cost_kWh & Storage::total_discharge_kWh & Storage::type_str &
Storage::charging_power_vec_kW def_readwrite("discharging_power_vec_kW", &Storage::discharging←
_power_vec_kW) .def_readwrite("capital_cost_vec" (
    "interpolator" ,
    &Storage::interpolator )
```

5.37 source/Controller.cpp File Reference

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

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



5.37.1 Detailed Description

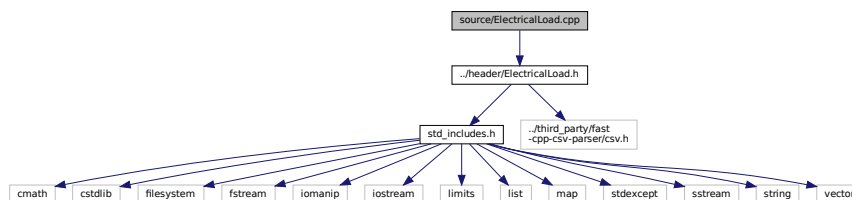
Implementation file for the [Controller](#) class.

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

5.38 source/ElectricalLoad.cpp File Reference

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

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



5.38.1 Detailed Description

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

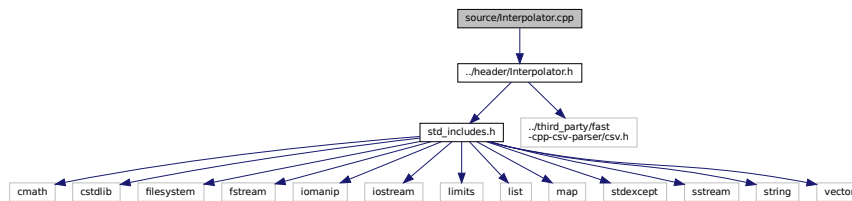
A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

5.39 source/Interpolator.cpp File Reference

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

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

Include dependency graph for Interpolator.cpp:



5.39.1 Detailed Description

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

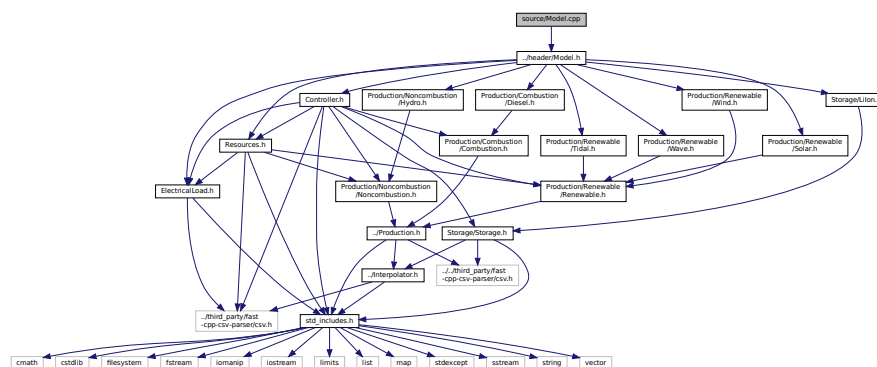
A class which contains interpolation data and functionality. Intended to serve as a component of the [Production](#) and [Storage](#) hierarchies.

5.40 source/Model.cpp File Reference

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

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

Include dependency graph for Model.cpp:



5.40.1 Detailed Description

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

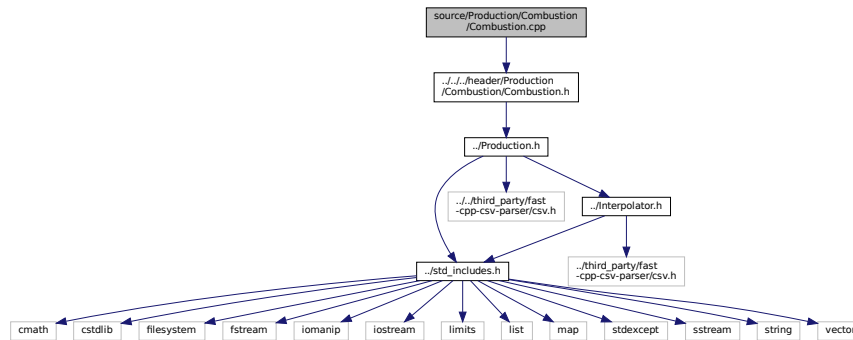
A container class which forms the centre of PGMcpp. The [Model](#) class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

5.41 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:



5.41.1 Detailed Description

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

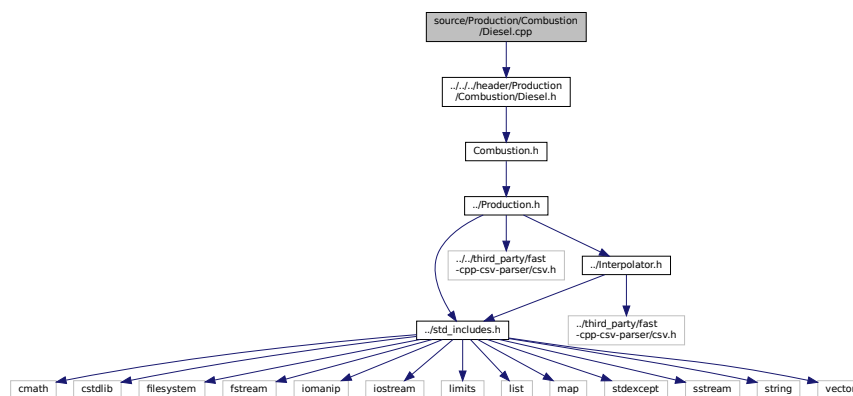
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

5.42 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:



5.42.1 Detailed Description

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

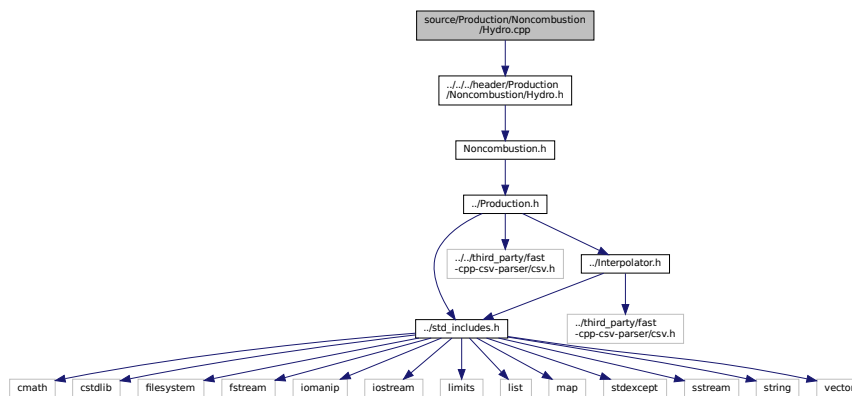
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

5.43 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:



5.43.1 Detailed Description

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

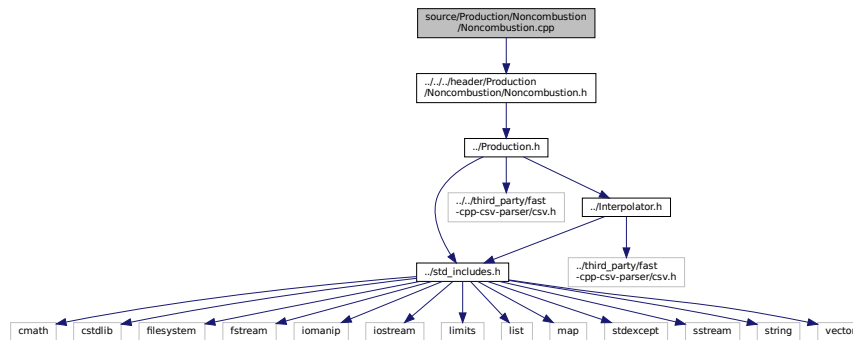
A derived class of the [Noncombustion](#) branch of [Production](#) which models production using a hydroelectric asset (either with reservoir or not).

5.44 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:



5.44.1 Detailed Description

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

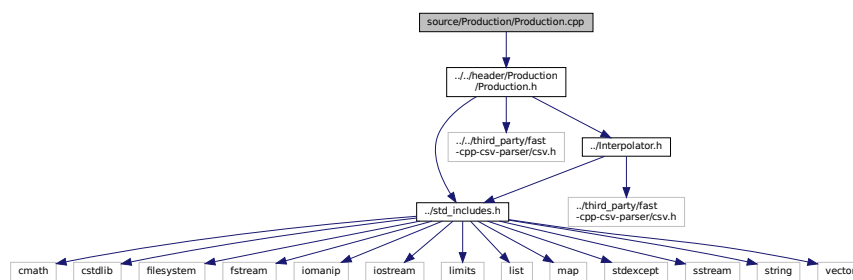
The root of the [Noncombustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

5.45 source/Production/Production.cpp File Reference

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

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

Include dependency graph for Production.cpp:



5.45.1 Detailed Description

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

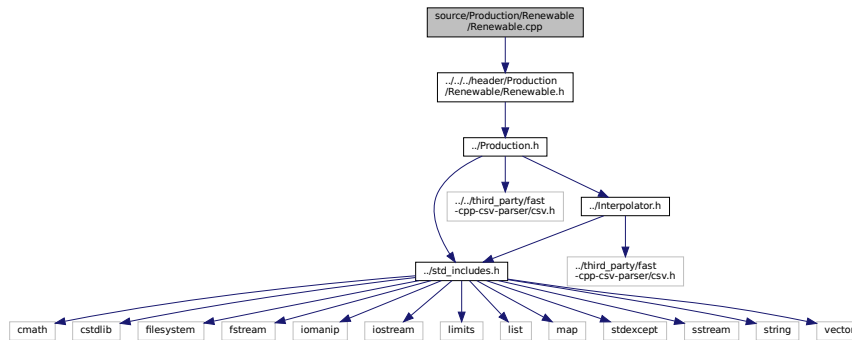
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.46 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:



5.46.1 Detailed Description

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

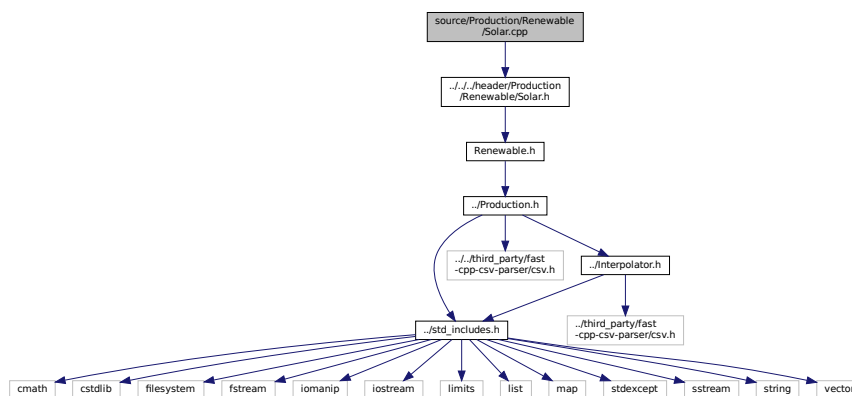
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

5.47 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:



5.47.1 Detailed Description

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

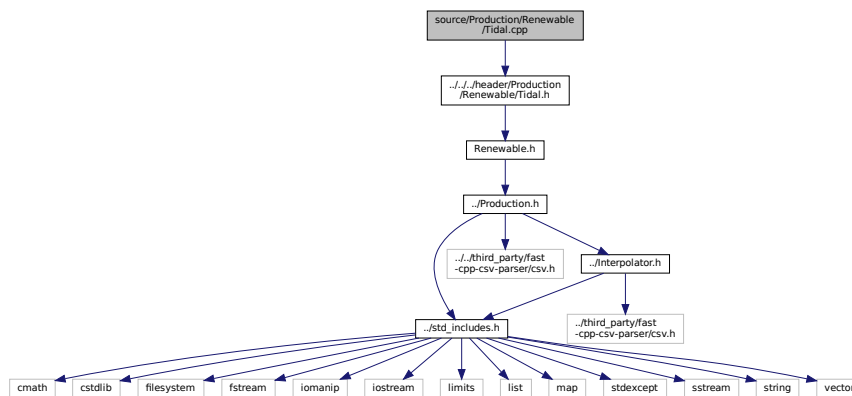
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

5.48 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:



5.48.1 Detailed Description

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

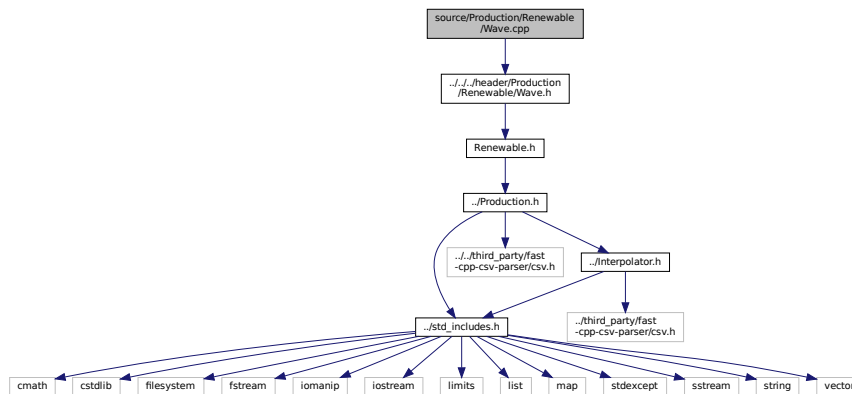
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

5.49 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:



5.49.1 Detailed Description

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

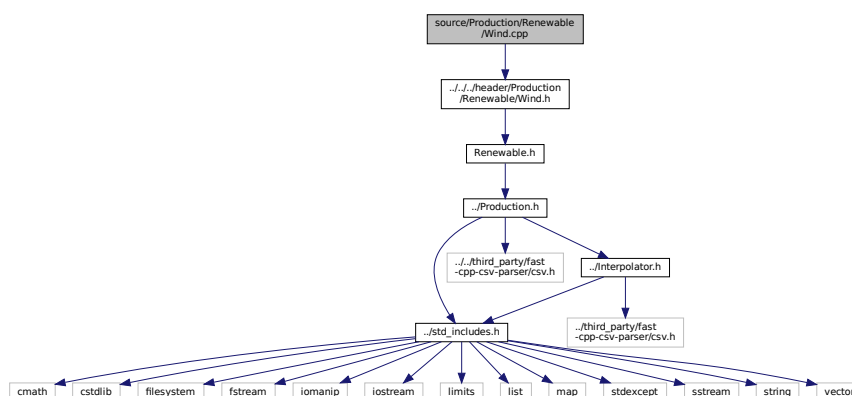
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

5.50 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:



5.50.1 Detailed Description

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

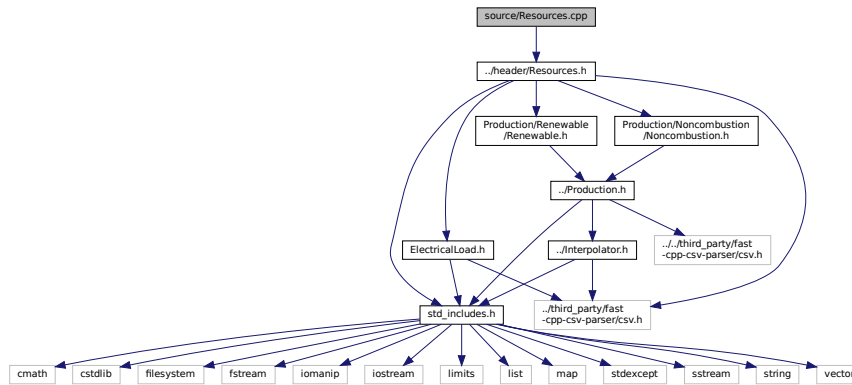
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

5.51 source/Resources.cpp File Reference

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

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

Include dependency graph for Resources.cpp:



5.51.1 Detailed Description

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

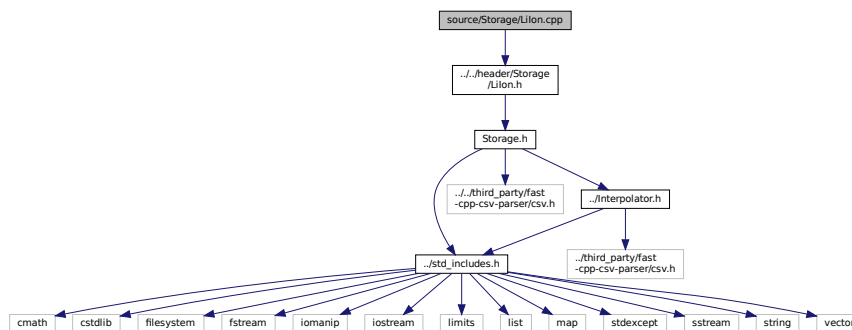
A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

5.52 source/Storage/Lilon.cpp File Reference

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

```
#include "../..//header/Storage/LiIon.h"
```

Include dependency graph for Lilon.cpp:



5.52.1 Detailed Description

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

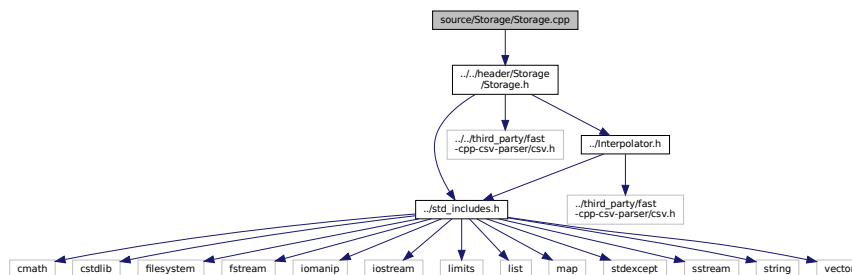
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

5.53 source/Storage/Storage.cpp File Reference

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

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

Include dependency graph for Storage.cpp:



5.53.1 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.

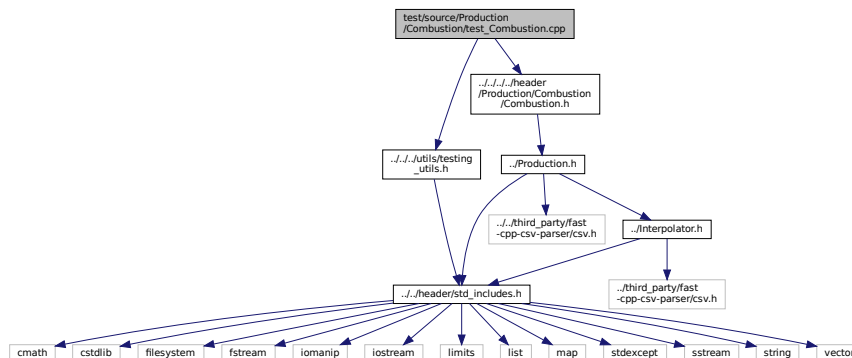
5.54 test/source/Production/Combustion/test_Combustion.cpp File Reference

Testing suite for [Combustion](#) class.

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

```
#include "../../../header/Production/Combustion/Combustion.h"
```

Include dependency graph for test_Combustion.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.54.1 Detailed Description

Testing suite for [Combustion](#) class.

A suite of tests for the [Combustion](#) class.

5.54.2 Function Documentation

5.54.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Combustion");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         CombustionInputs combustion_inputs;
42
43         Combustion test_combustion(8760, 1, combustion_inputs);
44
45         // ===== END CONSTRUCTION ===== //
46
47
48
49         // ===== ATTRIBUTES ===== //
50
51         testTruth(
52             not combustion_inputs.production_inputs.print_flag,
53             __FILE__,
54             __LINE__
55         );
56
57         testFloatEquals(
58             test_combustion.fuel_consumption_vec_L.size(),
59             8760,
60             __FILE__,
61             __LINE__
62         );
63
64         testFloatEquals(
65             test_combustion.fuel_cost_vec.size(),
66             8760,
67             __FILE__,
68             __LINE__
69         );
70
71         testFloatEquals(
72             test_combustion.CO2_emissions_vec_kg.size(),
73             8760,
74             __FILE__,
75             __LINE__
76         );
77
```

```

78 testFloatEquals(
79     test_combustion.CO_emissions_vec_kg.size(),
80     8760,
81     __FILE__,
82     __LINE__
83 );
84
85 testFloatEquals(
86     test_combustion.NOx_emissions_vec_kg.size(),
87     8760,
88     __FILE__,
89     __LINE__
90 );
91
92 testFloatEquals(
93     test_combustion.SOx_emissions_vec_kg.size(),
94     8760,
95     __FILE__,
96     __LINE__
97 );
98
99 testFloatEquals(
100    test_combustion.CH4_emissions_vec_kg.size(),
101    8760,
102    __FILE__,
103    __LINE__
104 );
105
106 testFloatEquals(
107    test_combustion.PM_emissions_vec_kg.size(),
108    8760,
109    __FILE__,
110    __LINE__
111 );
112
113 // ===== END ATTRIBUTES ===== //
114
115 }    /* try */
116
117
118 catch (...) {
119     //...
120
121     printGold(" ..... ");
122     printRed("FAIL");
123     std::cout << std::endl;
124     throw;
125 }
126
127
128 printGold(" ..... ");
129 printGreen("PASS");
130 std::cout << std::endl;
131 return 0;
132
133 }    /* main() */

```

5.55 test/source/Production/Combustion/test_Diesel.cpp File Reference

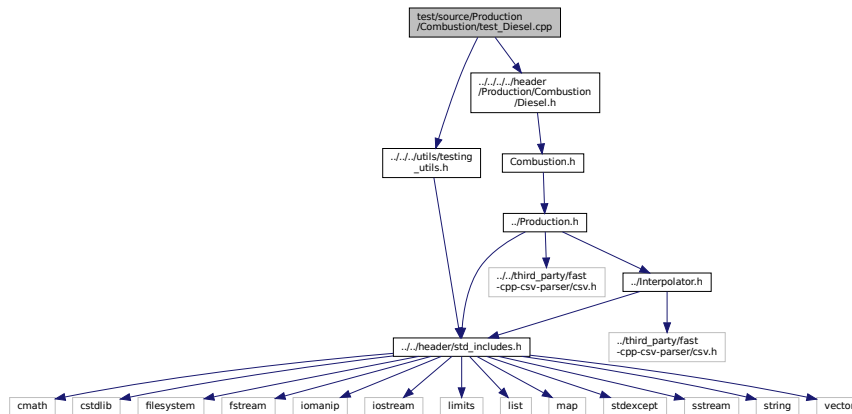
Testing suite for [Diesel](#) class.

```

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

```

Include dependency graph for test_Diesel.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.55.1 Detailed Description

Testing suite for [Diesel](#) class.

A suite of tests for the [Diesel](#) class.

5.55.2 Function Documentation

5.55.2.1 main()

```

int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Production <-- Combustion <-- Diesel");
    33
    34     srand(time(NULL));
    35
    36
    37     Combustion* test_diesel_ptr;
    38
    39     try {
    40
    41     // ===== CONSTRUCTION ===== //
    42
    43     bool error_flag = true;
    44
    45     try {
    46         DieselInputs bad_diesel_inputs;

```

```

47     bad_diesel_inputs.fuel_cost_L = -1;
48
49     Diesel bad_diesel(8760, 1, bad_diesel_inputs);
50
51     error_flag = false;
52 } catch (...) {
53     // Task failed successfully! =P
54 }
55 if (not error_flag) {
56     expectedErrorNotDetected(__FILE__, __LINE__);
57 }
58
59 DieselInputs diesel_inputs;
60
61 test_diesel_ptr = new Diesel(8760, 1, diesel_inputs);
62
63
64 diesel_inputs.combustion_inputs.fuel_mode = FuelMode :: FUEL_MODE_LOOKUP;
65 diesel_inputs.combustion_inputs.path_2_fuel_interp_data =
66     "data/test/interpolation/diesel_fuel_curve.csv";
67
68 Diesel test_diesel_lookup(8760, 1, diesel_inputs);
69
70
71 // ===== END CONSTRUCTION ===== //
72
73
74
75 // ===== ATTRIBUTES ===== //
76
77 testTruth(
78     not diesel_inputs.combustion_inputs.production_inputs.print_flag,
79     __FILE__,
80     __LINE__
81 );
82
83 testFloatEquals(
84     test_diesel_ptr->type,
85     CombustionType :: DIESEL,
86     __FILE__,
87     __LINE__
88 );
89
90 testTruth(
91     test_diesel_ptr->type_str == "DIESEL",
92     __FILE__,
93     __LINE__
94 );
95
96 testFloatEquals(
97     test_diesel_ptr->linear_fuel_slope_LkWh,
98     0.265675,
99     __FILE__,
100    __LINE__
101 );
102
103 testFloatEquals(
104     test_diesel_ptr->linear_fuel_intercept_LkWh,
105     0.026676,
106     __FILE__,
107     __LINE__
108 );
109
110 testFloatEquals(
111     test_diesel_ptr->capital_cost,
112     94125.375446,
113     __FILE__,
114     __LINE__
115 );
116
117 testFloatEquals(
118     test_diesel_ptr->operation_maintenance_cost_kWh,
119     0.069905,
120     __FILE__,
121     __LINE__
122 );
123
124 testFloatEquals(
125     ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
126     0.2,
127     __FILE__,
128     __LINE__
129 );
130
131 testFloatEquals(
132     ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
133     4,

```



```

134     __FILE__,
135     __LINE__
136 );
137
138 testFloatEquals(
139     test_diesel_ptr->replace_running_hrs,
140     30000,
141     __FILE__,
142     __LINE__
143 );
144
145 // ===== END ATTRIBUTES ===== //
146
147
148
149 // ===== METHODS ===== //
150
151 // test capacity constraint
152 testFloatEquals(
153     test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
154     test_diesel_ptr->capacity_kW,
155     __FILE__,
156     __LINE__
157 );
158
159 // test minimum load ratio constraint
160 testFloatEquals(
161     test_diesel_ptr->requestProductionkW(
162         0,
163         1,
164         0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
165             test_diesel_ptr->capacity_kW
166     ),
167     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
168     __FILE__,
169     __LINE__
170 );
171
172 // test commit()
173 std::vector<double> dt_vec_hrs (48, 1);
174
175 std::vector<double> load_vec_kW = {
176     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
177     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
178     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
179     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
180 };
181
182 std::vector<bool> expected_is_running_vec = {
183     1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
184     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
185     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
186     1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
187 };
188
189 double load_kW = 0;
190 double production_kW = 0;
191 double roll = 0;
192
193 for (int i = 0; i < 48; i++) {
194     roll = (double)rand() / RAND_MAX;
195
196     if (roll >= 0.95) {
197         roll = 1.25;
198     }
199
200     load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
201     load_kW = load_vec_kW[i];
202
203     production_kW = test_diesel_ptr->requestProductionkW(
204         i,
205         dt_vec_hrs[i],
206         load_kW
207     );
208
209     load_kW = test_diesel_ptr->commit(
210         i,
211         dt_vec_hrs[i],
212         production_kW,
213         load_kW
214     );
215
216     // load_kW <= load_vec_kW (i.e., after vs before)
217     testLessThanOrEqualTo(
218         load_kW,
219         load_vec_kW[i],
220         __FILE__,

```

```

221     __LINE__
222 );
223
224 // production = dispatch + storage + curtailment
225 testFloatEquals(
226     test_diesel_ptr->production_vec_kW[i] -
227     test_diesel_ptr->dispatch_vec_kW[i] -
228     test_diesel_ptr->storage_vec_kW[i] -
229     test_diesel_ptr->curtailment_vec_kW[i],
230     0,
231     __FILE__,
232     __LINE__
233 );
234
235 // capacity constraint
236 if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
237     testFloatEquals(
238         test_diesel_ptr->production_vec_kW[i],
239         test_diesel_ptr->capacity_kW,
240         __FILE__,
241         __LINE__
242     );
243 }
244
245 // minimum load ratio constraint
246 else if (
247     test_diesel_ptr->is_running and
248     test_diesel_ptr->production_vec_kW[i] > 0 and
249     load_vec_kW[i] <
250     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
251 ) {
252     testFloatEquals(
253         test_diesel_ptr->production_vec_kW[i],
254         ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
255         test_diesel_ptr->capacity_kW,
256         __FILE__,
257         __LINE__
258     );
259 }
260
261 // minimum runtime constraint
262 testFloatEquals(
263     test_diesel_ptr->is_running_vec[i],
264     expected_is_running_vec[i],
265     __FILE__,
266     __LINE__
267 );
268
269 // O&M, fuel consumption, and emissions > 0 whenever diesel is running
270 if (test_diesel_ptr->is_running) {
271     testGreaterThan(
272         test_diesel_ptr->operation_maintenance_cost_vec[i],
273         0,
274         __FILE__,
275         __LINE__
276     );
277
278     testGreaterThan(
279         test_diesel_ptr->fuel_consumption_vec_L[i],
280         0,
281         __FILE__,
282         __LINE__
283     );
284
285     testGreaterThan(
286         test_diesel_ptr->fuel_cost_vec[i],
287         0,
288         __FILE__,
289         __LINE__
290     );
291
292     testGreaterThan(
293         test_diesel_ptr->CO2_emissions_vec_kg[i],
294         0,
295         __FILE__,
296         __LINE__
297     );
298
299     testGreaterThan(
300         test_diesel_ptr->CO_emissions_vec_kg[i],
301         0,
302         __FILE__,
303         __LINE__
304     );
305
306     testGreaterThan(
307         test_diesel_ptr->NOx_emissions_vec_kg[i],

```

```

308         0,
309         __FILE__,
310         __LINE__
311     );
312
313     testGreaterThan(
314         test_diesel_ptr->SOx_emissions_vec_kg[i],
315         0,
316         __FILE__,
317         __LINE__
318     );
319
320     testGreaterThan(
321         test_diesel_ptr->CH4_emissions_vec_kg[i],
322         0,
323         __FILE__,
324         __LINE__
325     );
326
327     testGreaterThan(
328         test_diesel_ptr->PM_emissions_vec_kg[i],
329         0,
330         __FILE__,
331         __LINE__
332     );
333 }
334
335 // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
336 else {
337     testFloatEquals(
338         test_diesel_ptr->operation_maintenance_cost_vec[i],
339         0,
340         __FILE__,
341         __LINE__
342     );
343
344     testFloatEquals(
345         test_diesel_ptr->fuel_consumption_vec_L[i],
346         0,
347         __FILE__,
348         __LINE__
349     );
350
351     testFloatEquals(
352         test_diesel_ptr->fuel_cost_vec[i],
353         0,
354         __FILE__,
355         __LINE__
356     );
357
358     testFloatEquals(
359         test_diesel_ptr->CO2_emissions_vec_kg[i],
360         0,
361         __FILE__,
362         __LINE__
363     );
364
365     testFloatEquals(
366         test_diesel_ptr->CO_emissions_vec_kg[i],
367         0,
368         __FILE__,
369         __LINE__
370     );
371
372     testFloatEquals(
373         test_diesel_ptr->NOx_emissions_vec_kg[i],
374         0,
375         __FILE__,
376         __LINE__
377     );
378
379     testFloatEquals(
380         test_diesel_ptr->SOx_emissions_vec_kg[i],
381         0,
382         __FILE__,
383         __LINE__
384     );
385
386     testFloatEquals(
387         test_diesel_ptr->CH4_emissions_vec_kg[i],
388         0,
389         __FILE__,
390         __LINE__
391     );
392
393     testFloatEquals(
394         test_diesel_ptr->PM_emissions_vec_kg[i],

```

```

395         0,
396         __FILE__,
397         __LINE__
398     );
399 }
400 }
401
402 std::vector<double> load_ratio_vec = {
403     0,
404     0.170812859791767,
405     0.322739274162545,
406     0.369750203682042,
407     0.443532869135929,
408     0.471567864244626,
409     0.536513734479662,
410     0.586125806988674,
411     0.601101175455075,
412     0.658356862575221,
413     0.70576929893201,
414     0.784069734739331,
415     0.805765927542453,
416     0.884747873186048,
417     0.930870496062112,
418     0.979415217694769,
419     1
420 };
421
422 std::vector<double> expected_fuel_consumption_vec_L = {
423     4.68079520372916,
424     8.35159603357656,
425     11.7422361561399,
426     12.9931187917615,
427     14.8786636301325,
428     15.5746957307243,
429     17.1419229487141,
430     18.3041866133728,
431     18.6530540913696,
432     19.9569217633299,
433     21.012354614584,
434     22.7142305879957,
435     23.1916726441968,
436     24.8602332554707,
437     25.8172124624032,
438     26.8256741279932,
439     27.254952
440 };
441
442 for (size_t i = 0; i < load_ratio_vec.size(); i++) {
443     testFloatEquals(
444         test_diesel_lookup.getFuelConsumptionL(
445             1, load_ratio_vec[i] * test_diesel_lookup.capacity_kW
446         ),
447         expected_fuel_consumption_vec_L[i],
448         __FILE__,
449         __LINE__
450     );
451 }
452
453 // ===== END METHODS ===== //
454
455 } /* try */
456
457
458 catch (...) {
459     delete test_diesel_ptr;
460
461     printGold(" ..... ");
462     printRed("FAIL");
463     std::cout << std::endl;
464     throw;
465 }
466
467 delete test_diesel_ptr;
468
469 printGold(" ..... ");
470 printGreen("PASS");
471 std::cout << std::endl;
472 return 0;
473
474
475 } /* main() */

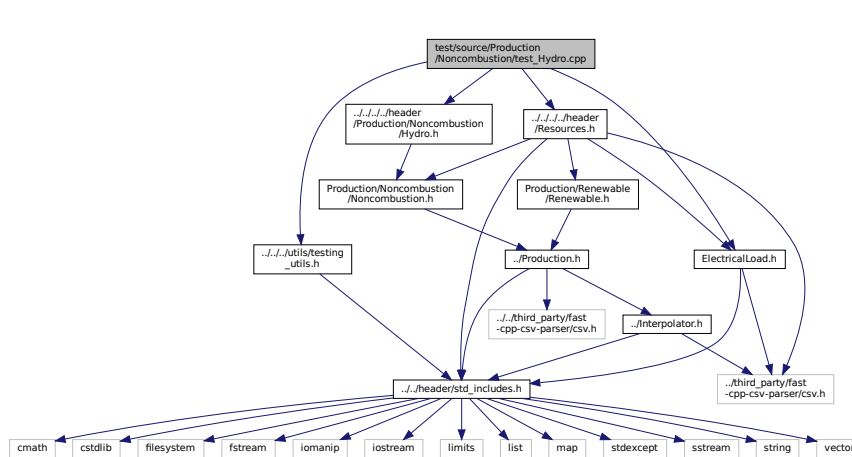
```

5.56 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

- int [main](#) (int argc, char **argv)

5.56.1 Detailed Description

Testing suite for [Hydro](#) class.

A suite of tests for the [Hydro](#) class.

5.56.2 Function Documentation

5.56.2.1 main()

```

int main (
    int argc,
    char ** argv )
29 {
30     #ifdef _WIN32
31         activateVirtualTerminal();
32     #endif /* _WIN32 */
33
34     printGold("\tTesting Production <-- Noncombustion <-- Hydro");
35
36     srand(time(NULL));
37
38
39     Noncombustion* test_hydro_ptr;
40
41     try {
42
43         // ===== CONSTRUCTION ===== //
44
45         std::string path_2_electrical_load_time_series =
46             "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
47
48         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
49
50         Resources test_resources;
51
52         HydroInputs hydro_inputs;
53         int hydro_resource_key = 0;
54
55         hydro_inputs.reservoir_capacity_m3 = 1000;
56         hydro_inputs.resource_key = hydro_resource_key;
57
58         test_hydro_ptr = new Hydro(8760, 1, hydro_inputs);
59
60         // ===== END CONSTRUCTION ===== //
61
62
63
64         // ===== ATTRIBUTES ===== //
65
66         testTruth(
67             not hydro_inputs.noncombustion_inputs.production_inputs.print_flag,
68             __FILE__,
69             __LINE__
70         );
71
72         testFloatEquals(
73             test_hydro_ptr->type,
74             NoncombustionType :: HYDRO,
75             __FILE__,
76             __LINE__
77         );
78
79         testTruth(
80             test_hydro_ptr->type_str == "HYDRO",
81             __FILE__,
82             __LINE__
83         );
84
85         testFloatEquals(
86             ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
87             1000,
88             __FILE__,
89             __LINE__
90         );
91
92
93         // ===== END ATTRIBUTES ===== //
94
95
96
97         // ===== METHODS ===== //
98
99         std::string path_2_hydro_resource_data =
100             "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
101
102         test_resources.addResource(
103             NoncombustionType::HYDRO,
104             path_2_hydro_resource_data,
105             hydro_resource_key,
106             &test_electrical_load
107         );
108

```

```

109 double load_kW = 100 * (double)rand() / RAND_MAX;
110 double production_kW = 0;
111
112 for (int i = 0; i < 8760; i++) {
113     production_kW = test_hydro_ptr->requestProductionkW(
114         i,
115         1,
116         load_kW,
117         test_resources.resource_map_1D[test_hydro_ptr->resource_key][i]
118     );
119
120     load_kW = test_hydro_ptr->commit(
121         i,
122         1,
123         production_kW,
124         load_kW,
125         test_resources.resource_map_1D[test_hydro_ptr->resource_key][i]
126     );
127
128     testGreaterThanOrEqualTo(
129         test_hydro_ptr->production_vec_kW[i],
130         0,
131         __FILE__,
132         __LINE__
133     );
134
135     testLessThanOrEqualTo(
136         test_hydro_ptr->production_vec_kW[i],
137         test_hydro_ptr->capacity_kW,
138         __FILE__,
139         __LINE__
140     );
141
142     testFloatEquals(
143         test_hydro_ptr->production_vec_kW[i] -
144         test_hydro_ptr->dispatch_vec_kW[i] -
145         test_hydro_ptr->curtailment_vec_kW[i] -
146         test_hydro_ptr->storage_vec_kW[i],
147         0,
148         __FILE__,
149         __LINE__
150     );
151
152     testGreaterThanOrEqualTo(
153         ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
154         0,
155         __FILE__,
156         __LINE__
157     );
158
159     testLessThanOrEqualTo(
160         ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
161         ((Hydro*)test_hydro_ptr)->maximum_flow_m3hr,
162         __FILE__,
163         __LINE__
164     );
165
166     testGreaterThanOrEqualTo(
167         ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
168         0,
169         __FILE__,
170         __LINE__
171     );
172
173     testLessThanOrEqualTo(
174         ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
175         ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
176         __FILE__,
177         __LINE__
178     );
179 }
180
181 // ===== END METHODS ===== //
182
183 } /* try */
184
185
186 catch (...) {
187     delete test_hydro_ptr;
188
189     printGold(" ... ");
190     printRed("FAIL");
191     std::cout << std::endl;
192     throw;
193 }
194
195

```

```

196 delete test_hydro_ptr;
197
198 printGold(" ... ");
199 printGreen("PASS");
200 std::cout << std::endl;
201 return 0;
202
203 } /* main() */

```

5.57 test/source/Production/Noncombustion/test_Noncombustion.cpp

File Reference

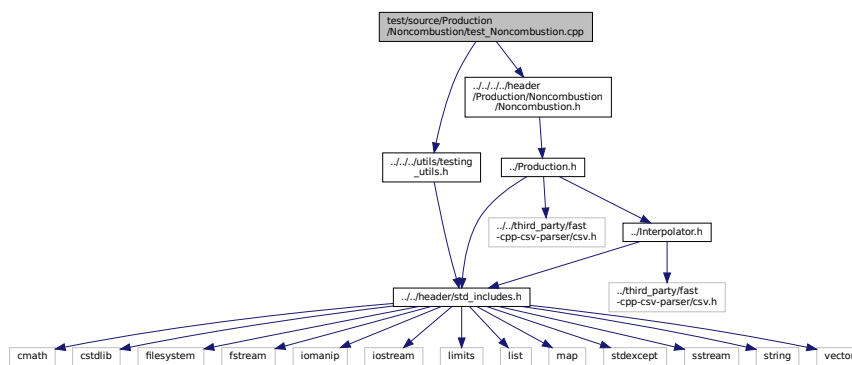
Testing suite for [Noncombustion](#) class.

```

#include "../.../utils/testing_utils.h"
#include "../.../header/Production/Noncombustion/Noncombustion.h"

```

Include dependency graph for test_Noncombustion.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.57.1 Detailed Description

Testing suite for [Noncombustion](#) class.

A suite of tests for the [Noncombustion](#) class.

5.57.2 Function Documentation

5.57.2.1 main()

```

int main (
    int argc,
    char ** argv )
28 {
29     #ifdef _WIN32
30         activateVirtualTerminal();
31     #endif /* _WIN32 */
32
33     printGold("\tTesting Production <-- Noncombustion");
34
35     srand(time(NULL));
36
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         NoncombustionInputs noncombustion_inputs;
43
44         Noncombustion test_noncombustion(8760, 1, noncombustion_inputs);
45
46         // ===== END CONSTRUCTION ===== //
47
48
49
50         // ===== ATTRIBUTES ===== //
51
52         testTruth(
53             not noncombustion_inputs.production_inputs.print_flag,
54             __FILE__,
55             __LINE__
56         );
57
58         // ===== END ATTRIBUTES ===== //
59
60     } /* try */
61
62
63     catch (...) {
64         //...
65
66         printGold(" ..... ");
67         printRed("FAIL");
68         std::cout << std::endl;
69         throw;
70     }
71
72
73     printGold(" ..... ");
74     printGreen("PASS");
75     std::cout << std::endl;
76     return 0;
77
78 } /* main() */

```

5.58 test/source/Production/Renewable/test_Renewable.cpp File Reference

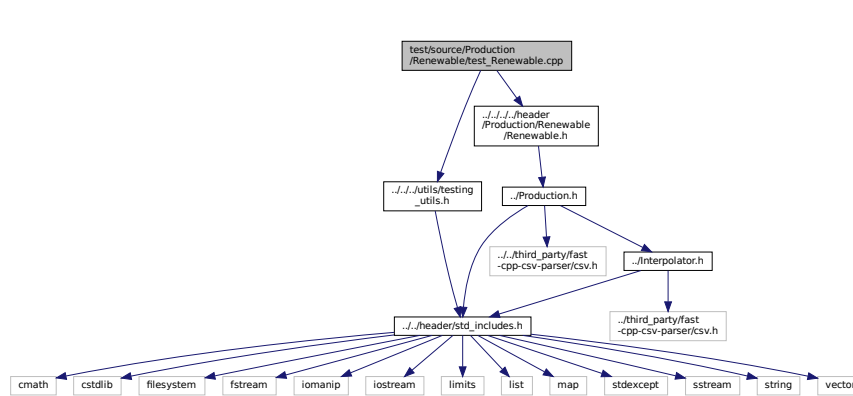
Testing suite for [Renewable](#) class.

```

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

```

Include dependency graph for test_Renewable.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.58.1 Detailed Description

Testing suite for [Renewable](#) class.

A suite of tests for the [Renewable](#) class.

5.58.2 Function Documentation

5.58.2.1 main()

```

int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Production <-- Renewable");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38
    39 // ===== CONSTRUCTION ===== //
    40
    41     RenewableInputs renewable_inputs;
    42
    43     Renewable test_renewable(8760, 1, renewable_inputs);
    44
    45 // ===== END CONSTRUCTION ===== //
    46
    47
    48

```

```

49 // ===== ATTRIBUTES ===== //
50
51 testTruth(
52     not renewable_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56
57 // ===== END ATTRIBUTES ===== //
58
59 } /* try */
60
61
62 catch (...) {
63     //...
64
65     printGold(" ..... ");
66     printRed("FAIL");
67     std::cout << std::endl;
68     throw;
69 }
70
71
72 printGold(" ..... ");
73 printGreen("PASS");
74 std::cout << std::endl;
75 return 0;
76 } /* main() */

```

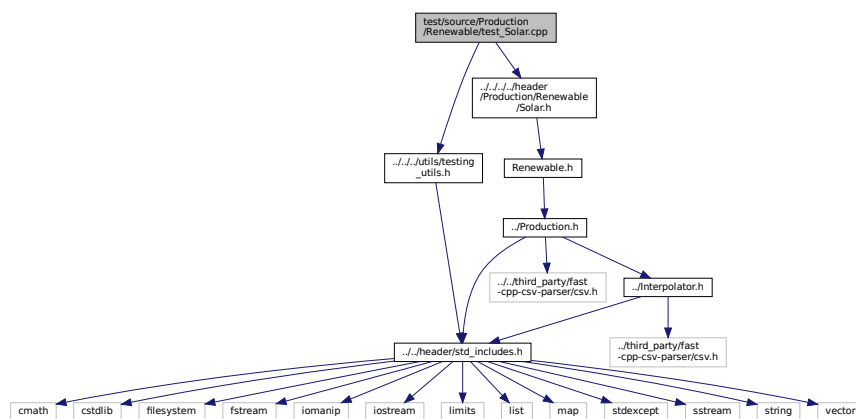
5.59 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for [Solar](#) class.

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

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

Include dependency graph for test_Solar.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.59.1 Detailed Description

Testing suite for [Solar](#) class.

A suite of tests for the [Solar](#) class.

5.59.2 Function Documentation

5.59.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Solar");
33
34     srand(time(NULL));
35
36     Renewable* test_solar_ptr;
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         bool error_flag = true;
43
44         try {
45             SolarInputs bad_solar_inputs;
46             bad_solar_inputs.derating = -1;
47
48             Solar bad_solar(8760, 1, bad_solar_inputs);
49
50             error_flag = false;
51         } catch (...) {
52             // Task failed successfully! =P
53         }
54         if (not error_flag) {
55             expectedErrorNotDetected(__FILE__, __LINE__);
56         }
57
58         SolarInputs solar_inputs;
59
60         test_solar_ptr = new Solar(8760, 1, solar_inputs);
61
62         // ===== END CONSTRUCTION ===== //
63
64
65
66         // ===== ATTRIBUTES ===== //
67
68         testTruth(
69             not solar_inputs.renewable_inputs.production_inputs.print_flag,
70             __FILE__,
71             __LINE__
72 );
73
74         testFloatEquals(
75             test_solar_ptr->type,
76             RenewableType :: SOLAR,
77             __FILE__,
78             __LINE__
79 );
80
81         testTruth(
82             test_solar_ptr->type_str == "SOLAR",
83             __FILE__,
84             __LINE__
85 );
86
87         testFloatEquals(
88             test_solar_ptr->capital_cost,
89             350118.723363,
90             __FILE__,
91             __LINE__
92 );
93
94         testFloatEquals(
95             test_solar_ptr->operation_maintenance_cost_kWh,
96             0.01,
97             __FILE__,

```

```

98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_solar_ptr->computeProductionkW(0, 1, 2),
110     100,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_solar_ptr->computeProductionkW(0, 1, -1),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double solar_resource_kWm2 = 0;
136
137 for (int i = 0; i < 48; i++) {
138     roll = (double)rand() / RAND_MAX;
139
140     solar_resource_kWm2 = roll;
141
142     roll = (double)rand() / RAND_MAX;
143
144     if (roll <= 0.1) {
145         solar_resource_kWm2 = 0;
146     }
147
148     else if (roll >= 0.95) {
149         solar_resource_kWm2 = 1.25;
150     }
151
152     roll = (double)rand() / RAND_MAX;
153
154     if (roll >= 0.95) {
155         roll = 1.25;
156     }
157
158     load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
159     load_kW = load_vec_kW[i];
160
161     production_kW = test_solar_ptr->computeProductionkW(
162         i,
163         dt_vec_hrs[i],
164         solar_resource_kWm2
165     );
166
167     load_kW = test_solar_ptr->commit(
168         i,
169         dt_vec_hrs[i],
170         production_kW,
171         load_kW
172     );
173
174     // is running (or not) as expected
175     if (solar_resource_kWm2 > 0) {
176         testTruth(
177             test_solar_ptr->is_running,
178             __FILE__,
179             __LINE__
180         );
181     }
182
183     else {
184         testTruth(

```

```

185         not test_solar_ptr->is_running,
186         __FILE__,
187         __LINE__
188     );
189 }
190
191 // load_kW <= load_vec_kW (i.e., after vs before)
192 testLessThanOrEqualTo(
193     load_kW,
194     load_vec_kW[i],
195     __FILE__,
196     __LINE__
197 );
198
199 // production = dispatch + storage + curtailment
200 testFloatEquals(
201     test_solar_ptr->production_vec_kW[i] -
202     test_solar_ptr->dispatch_vec_kW[i] -
203     test_solar_ptr->storage_vec_kW[i] -
204     test_solar_ptr->curtailment_vec_kW[i],
205     0,
206     __FILE__,
207     __LINE__
208 );
209
210 // capacity constraint
211 if (solar_resource_kWm2 > 1) {
212     testFloatEquals(
213         test_solar_ptr->production_vec_kW[i],
214         test_solar_ptr->capacity_kW,
215         __FILE__,
216         __LINE__
217     );
218 }
219
220 // resource, O&M > 0 whenever solar is running (i.e., producing)
221 if (test_solar_ptr->is_running) {
222     testGreaterThan(
223         solar_resource_kWm2,
224         0,
225         __FILE__,
226         __LINE__
227     );
228
229     testGreaterThan(
230         test_solar_ptr->operation_maintenance_cost_vec[i],
231         0,
232         __FILE__,
233         __LINE__
234     );
235 }
236
237 // resource, O&M = 0 whenever solar is not running (i.e., not producing)
238 else {
239     testFloatEquals(
240         solar_resource_kWm2,
241         0,
242         __FILE__,
243         __LINE__
244     );
245
246     testFloatEquals(
247         test_solar_ptr->operation_maintenance_cost_vec[i],
248         0,
249         __FILE__,
250         __LINE__
251     );
252 }
253 }
254
255
256 // ===== END METHODS ===== //
257
258 } /* try */
259
260
261 catch (...) {
262     delete test_solar_ptr;
263
264     printGold(" ..... ");
265     printRed("FAIL");
266     std::cout << std::endl;
267     throw;
268 }
269
270
271 delete test_solar_ptr;

```

```

272
273 printGold(" ..... ");
274 printGreen("PASS");
275 std::cout << std::endl;
276 return 0;
277 } /* main() */

```

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

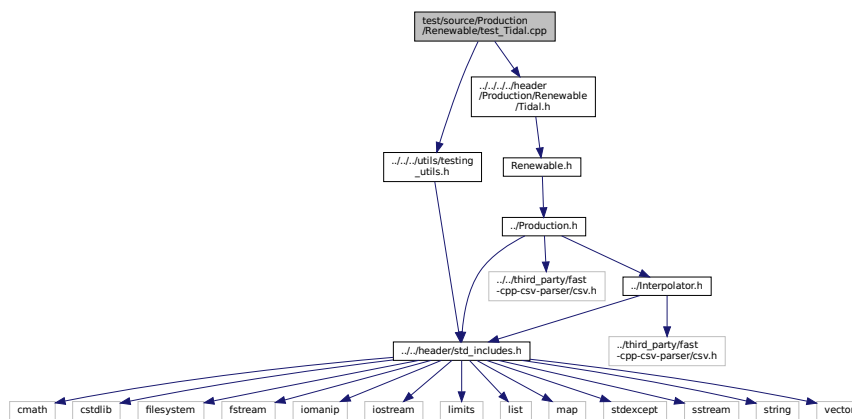
Testing suite for [Tidal](#) class.

```

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

```

Include dependency graph for test_Tidal.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.60.1 Detailed Description

Testing suite for [Tidal](#) class.

A suite of tests for the [Tidal](#) class.

5.60.2 Function Documentation

5.60.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Tidal");
33
34     srand(time(NULL));
35
36     Renewable* test_tidal_ptr;
37
38     try {
39
40     // ===== CONSTRUCTION ===== //
41
42     bool error_flag = true;
43
44     try {
45         TidalInputs bad_tidal_inputs;
46         bad_tidal_inputs.design_speed_ms = -1;
47
48         Tidal bad_tidal(8760, 1, bad_tidal_inputs);
49
50         error_flag = false;
51     } catch (...) {
52         // Task failed successfully! =P
53     }
54     if (not error_flag) {
55         expectedErrorNotDetected(__FILE__, __LINE__);
56     }
57
58     TidalInputs tidal_inputs;
59
60     test_tidal_ptr = new Tidal(8760, 1, tidal_inputs);
61
62     // ===== END CONSTRUCTION ===== //
63
64
65
66     // ===== ATTRIBUTES ===== //
67
68     testTruth(
69         not tidal_inputs.renewable_inputs.production_inputs.print_flag,
70         __FILE__,
71         __LINE__
72 );
73
74     testFloatEquals(
75         test_tidal_ptr->type,
76         RenewableType :: TIDAL,
77         __FILE__,
78         __LINE__
79 );
80
81     testTruth(
82         test_tidal_ptr->type_str == "TIDAL",
83         __FILE__,
84         __LINE__
85 );
86
87     testFloatEquals(
88         test_tidal_ptr->capital_cost,
89         500237.446725,
90         __FILE__,
91         __LINE__
92 );
93
94     testFloatEquals(
95         test_tidal_ptr->operation_maintenance_cost_kWh,
96         0.069905,
97         __FILE__,
98         __LINE__
99 );
100
101     // ===== END ATTRIBUTES ===== //
102
103
104
105     // ===== METHODS ===== //
106

```



```

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

```

```

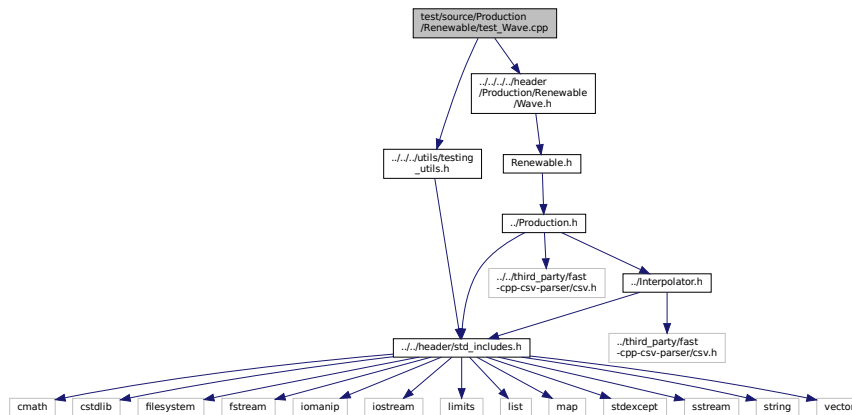
194     else {
195         testTruth(
196             not test_tidal_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_tidal_ptr->production_vec_kW[i] -
213         test_tidal_ptr->dispatch_vec_kW[i] -
214         test_tidal_ptr->storage_vec_kW[i] -
215         test_tidal_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220
221     // resource, O&M > 0 whenever tidal is running (i.e., producing)
222     if (test_tidal_ptr->is_running) {
223         testGreaterThan(
224             tidal_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229
230         testGreaterThan(
231             test_tidal_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // O&M = 0 whenever tidal is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_tidal_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248
249
250 // ===== END METHODS ===== //
251
252 } /* try */
253
254
255 catch (...) {
256     delete test_tidal_ptr;
257
258     printGold(" ..... ");
259     printRed("FAIL");
260     std::cout << std::endl;
261     throw;
262 }
263
264
265 delete test_tidal_ptr;
266
267 printGold(" ..... ");
268 printGreen("PASS");
269 std::cout << std::endl;
270 return 0;
271 } /* main() */

```

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

Testing suite for [Wave](#) class.

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



Functions

- int [main](#) (int argc, char **argv)

5.61.1 Detailed Description

Testing suite for [Wave](#) class.

A suite of tests for the [Wave](#) class.

5.61.2 Function Documentation

5.61.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Production <-- Renewable <-- Wave");
    33
    34     srand(time(NULL));
    35
    36     Renewable* test_wave_ptr;
    37
    38     try {
    39
    40     // ===== CONSTRUCTION ===== //
    41
    42     bool error_flag = true;
    43 }
```

```

44 try {
45     WaveInputs bad_wave_inputs;
46     bad_wave_inputs.design_significant_wave_height_m = -1;
47
48     Wave bad_wave(8760, 1, bad_wave_inputs);
49
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57
58 WaveInputs wave_inputs;
59
60 test_wave_ptr = new Wave(8760, 1, wave_inputs);
61
62
63 wave_inputs.power_model = WavePowerProductionModel :: WAVE_POWER_LOOKUP;
64 wave_inputs.path_2_normalized_performance_matrix =
65     "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
66
67 Wave test_wave_lookup(8760, 1, wave_inputs);
68
69 // ===== END CONSTRUCTION ===== //
70
71
72
73 // ===== ATTRIBUTES ===== //
74
75 testTruth(
76     not wave_inputs.renewable_inputs.production_inputs.print_flag,
77     __FILE__,
78     __LINE__
79 );
80
81 testFloatEquals(
82     test_wave_ptr->type,
83     RenewableType :: WAVE,
84     __FILE__,
85     __LINE__
86 );
87
88 testTruth(
89     test_wave_ptr->type_str == "WAVE",
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wave_ptr->capital_cost,
96     850831.063539,
97     __FILE__,
98     __LINE__
99 );
100
101 testFloatEquals(
102     test_wave_ptr->operation_maintenance_cost_kWh,
103     0.069905,
104     __FILE__,
105     __LINE__
106 );
107
108 // ===== END ATTRIBUTES ===== //
109
110
111
112 // ===== METHODS ===== //
113
114 // test production constraints
115 testFloatEquals(
116     test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 testFloatEquals(
123     test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
124     0,
125     __FILE__,
126     __LINE__
127 );
128
129 // test commit()
130 std::vector<double> dt_vec_hrs (48, 1);

```

```

131
132 std::vector<double> load_vec_kW = {
133     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
134     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
135     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
136     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
137 };
138
139 double load_kW = 0;
140 double production_kW = 0;
141 double roll = 0;
142 double significant_wave_height_m = 0;
143 double energy_period_s = 0;
144
145 for (int i = 0; i < 48; i++) {
146     roll = (double)rand() / RAND_MAX;
147
148     if (roll <= 0.05) {
149         roll = 0;
150     }
151
152     significant_wave_height_m = roll *
153         ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
154
155     roll = (double)rand() / RAND_MAX;
156
157     if (roll <= 0.05) {
158         roll = 0;
159     }
160
161     energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
162
163     roll = (double)rand() / RAND_MAX;
164
165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_wave_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         significant_wave_height_m,
176         energy_period_s
177     );
178
179     load_kW = test_wave_ptr->commit(
180         i,
181         dt_vec_hrs[i],
182         production_kW,
183         load_kW
184     );
185
186     // is running (or not) as expected
187     if (production_kW > 0) {
188         testTruth(
189             test_wave_ptr->is_running,
190             __FILE__,
191             __LINE__
192         );
193     }
194
195     else {
196         testTruth(
197             not test_wave_ptr->is_running,
198             __FILE__,
199             __LINE__
200         );
201     }
202
203     // load_kW <= load_vec_kW (i.e., after vs before)
204     testLessThanOrEqualTo(
205         load_kW,
206         load_vec_kW[i],
207         __FILE__,
208         __LINE__
209     );
210
211     // production = dispatch + storage + curtailment
212     testFloatEquals(
213         test_wave_ptr->production_vec_kW[i] -
214         test_wave_ptr->dispatch_vec_kW[i] -
215         test_wave_ptr->storage_vec_kW[i] -
216         test_wave_ptr->curtailment_vec_kW[i],
217         0,

```

```

218     __FILE__,
219     __LINE__
220 );
221
222 // resource, O&M > 0 whenever wave is running (i.e., producing)
223 if (test_wave_ptr->is_running) {
224     testGreaterThan(
225         significant_wave_height_m,
226         0,
227         __FILE__,
228         __LINE__
229     );
230
231     testGreaterThan(
232         energy_period_s,
233         0,
234         __FILE__,
235         __LINE__
236     );
237
238     testGreaterThan(
239         test_wave_ptr->operation_maintenance_cost_vec[i],
240         0,
241         __FILE__,
242         __LINE__
243     );
244 }
245
246 // O&M = 0 whenever wave is not running (i.e., not producing)
247 else {
248     testFloatEquals(
249         test_wave_ptr->operation_maintenance_cost_vec[i],
250         0,
251         __FILE__,
252         __LINE__
253     );
254 }
255 }
256
257 std::vector<double> significant_wave_height_vec_m = {
258     0.389211848822208,
259     0.836477431896843,
260     1.52738334015579,
261     1.92640601114508,
262     2.27297317532019,
263     2.87416589636605,
264     3.72275770908175,
265     3.95063175885536,
266     4.68097139867404,
267     4.97775020449812,
268     5.55184219980547,
269     6.06566629451658,
270     6.27927876785062,
271     6.96218133671013,
272     7.51754442460228
273 };
274
275 std::vector<double> energy_period_vec_s = {
276     5.45741899698926,
277     6.00101329139007,
278     7.50567689404182,
279     8.77681262912881,
280     9.45143678206774,
281     10.7767876462885,
282     11.4795760857165,
283     12.9430684577599,
284     13.303544885703,
285     14.5069863517863,
286     15.1487890438045,
287     16.086524049077,
288     17.176609978648,
289     18.4155153740256,
290     19.1704554940162
291 };
292
293 std::vector<std::vector<double>> expected_normalized_performance_matrix = {
294     {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
295     {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
296     {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
297     {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
298     {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
299

```

```

300     {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.7278111
301     {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.705111
302     {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
303     {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
304     {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
305     {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
306     {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
307     {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
308     {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
309     {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
309 };
310
311 for (size_t i = 0; i < energy_period_vec_s.size(); i++) {
312     for (size_t j = 0; j < significant_wave_height_vec_m.size(); j++) {
313         testFloatEquals(
314             test_wave_lookup.computeProductionkW(
315                 0,
316                 1,
317                 significant_wave_height_vec_m[j],
318                 energy_period_vec_s[i]
319             ),
320             expected_normalized_performance_matrix[i][j] *
321             test_wave_lookup.capacity_kW,
322             __FILE__,
323             __LINE__
324         );
325     }
326 }
327
328 // ===== END METHODS ===== //
329
330 } /* try */
331
332
333 catch (...) {
334     delete test_wave_ptr;
335
336     printGold(" ..... ");
337     printRed("FAIL");
338     std::cout << std::endl;
339     throw;
340 }
341
342
343 delete test_wave_ptr;
344
345 printGold(" ..... ");
346 printGreen("PASS");
347 std::cout << std::endl;
348 return 0;
349 } /* main() */

```

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

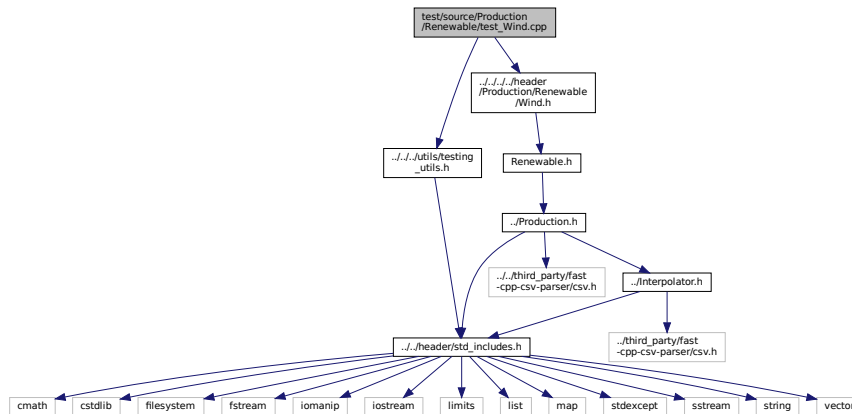
Testing suite for [Wind](#) class.

```

#include "../.../utils/testing_utils.h"
#include "../.../header/Production/Renewable/Wind.h"

```

Include dependency graph for test_Wind.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.62.1 Detailed Description

Testing suite for [Wind](#) class.

A suite of tests for the [Wind](#) class.

5.62.2 Function Documentation

5.62.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wind");
33
34     srand(time(NULL));
35
36     Renewable* test_wind_ptr;
37
38     try {
39
40     // ===== CONSTRUCTION ===== //
41
42     bool error_flag = true;
43
44     try {
45         WindInputs bad_wind_inputs;
46         bad_wind_inputs.design_speed_ms = -1;

```



```

47
48     Wind bad_wind(8760, 1, bad_wind_inputs);
49
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57
58 WindInputs wind_inputs;
59
60 test_wind_ptr = new Wind(8760, 1, wind_inputs);
61
62 // ===== END CONSTRUCTION ===== //
63
64
65
66 // ===== ATTRIBUTES ===== //
67
68 testTruth(
69     not wind_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73
74 testFloatEquals(
75     test_wind_ptr->type,
76     RenewableType :: WIND,
77     __FILE__,
78     __LINE__
79 );
80
81 testTruth(
82     test_wind_ptr->type_str == "WIND",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_wind_ptr->capital_cost,
89     450356.170088,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wind_ptr->operation_maintenance_cost_kWh,
96     0.034953,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wind_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wind_ptr->computeProductionkW(
117         0,
118         1,
119         ((Wind*)test_wind_ptr)->design_speed_ms
120     ),
121     test_wind_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_wind_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()

```

```

134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double wind_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     wind_resource_ms = roll * ((Wind*)test_wind_ptr->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         wind_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         wind_resource_ms = 3 * ((Wind*)test_wind_ptr->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164
165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_wind_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         wind_resource_ms
176     );
177
178     load_kW = test_wind_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_wind_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193
194     else {
195         testTruth(
196             not test_wind_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_wind_ptr->production_vec_kW[i] -
213         test_wind_ptr->dispatch_vec_kW[i] -
214         test_wind_ptr->storage_vec_kW[i] -
215         test_wind_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220

```

```

221 // resource, O&M > 0 whenever wind is running (i.e., producing)
222 if (test_wind_ptr->is_running) {
223     testGreaterThan(
224         wind_resource_ms,
225         0,
226         __FILE__,
227         __LINE__
228     );
229
230     testGreaterThan(
231         test_wind_ptr->operation_maintenance_cost_vec[i],
232         0,
233         __FILE__,
234         __LINE__
235     );
236 }
237
238 // O&M = 0 whenever wind is not running (i.e., not producing)
239 else {
240     testFloatEquals(
241         test_wind_ptr->operation_maintenance_cost_vec[i],
242         0,
243         __FILE__,
244         __LINE__
245     );
246 }
247 }
248
249
250 // ===== END METHODS ===== //
251
252 } /* try */
253
254
255 catch (...) {
256     delete test_wind_ptr;
257
258     printGold(" ..... ");
259     printRed("FAIL");
260     std::cout << std::endl;
261     throw;
262 }
263
264 delete test_wind_ptr;
265
266 printGold(" ..... ");
267 printGreen("PASS");
268 std::cout << std::endl;
269 return 0;
270 } /* main() */

```

5.63 test/source/Production/test_Production.cpp File Reference

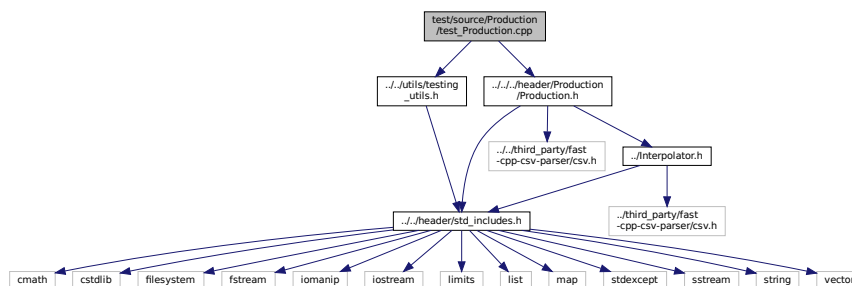
Testing suite for [Production](#) class.

```

#include "../utils/testing_utils.h"
#include "../header/Production/Production.h"

```

Include dependency graph for test_Production.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.63.1 Detailed Description

Testing suite for [Production](#) class.

A suite of tests for the [Production](#) class.

5.63.2 Function Documentation

5.63.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         bool error_flag = true;
42
43         try {
44             ProductionInputs production_inputs;
45
46             Production bad_production(0, 1, production_inputs);
47
48             error_flag = false;
49         } catch (...) {
50             // Task failed successfully! =P
51         }
52         if (not error_flag) {
53             expectedErrorNotDetected(__FILE__, __LINE__);
54         }
55
56         ProductionInputs production_inputs;
57
58         Production test_production(8760, 1, production_inputs);
59
60         // ===== END CONSTRUCTION ===== //
61
62
63
64         // ===== ATTRIBUTES ===== //
65
66         testTruth(
67             not production_inputs.print_flag,
68             __FILE__,
69             __LINE__
70 );
71
72         testFloatEquals(
73             production_inputs.nominal_inflation_annual,
74             0.02,
75             __FILE__,
76             __LINE__
77 );
```

```

78
79 testFloatEquals(
80     production_inputs.nominal_discount_annual,
81     0.04,
82     __FILE__,
83     __LINE__
84 );
85
86 testFloatEquals(
87     test_production.n_points,
88     8760,
89     __FILE__,
90     __LINE__
91 );
92
93 testFloatEquals(
94     test_production.capacity_kW,
95     100,
96     __FILE__,
97     __LINE__
98 );
99
100 testFloatEquals(
101     test_production.real_discount_annual,
102     0.0196078431372549,
103     __FILE__,
104     __LINE__
105 );
106
107 testFloatEquals(
108     test_production.production_vec_kW.size(),
109     8760,
110     __FILE__,
111     __LINE__
112 );
113
114 testFloatEquals(
115     test_production.dispatch_vec_kW.size(),
116     8760,
117     __FILE__,
118     __LINE__
119 );
120
121 testFloatEquals(
122     test_production.storage_vec_kW.size(),
123     8760,
124     __FILE__,
125     __LINE__
126 );
127
128 testFloatEquals(
129     test_production.curtailment_vec_kW.size(),
130     8760,
131     __FILE__,
132     __LINE__
133 );
134
135 testFloatEquals(
136     test_production.capital_cost_vec.size(),
137     8760,
138     __FILE__,
139     __LINE__
140 );
141
142 testFloatEquals(
143     test_production.operation_maintenance_cost_vec.size(),
144     8760,
145     __FILE__,
146     __LINE__
147 );
148
149 // ===== END ATTRIBUTES ===== //
150
151 } /* try */
152
153
154 catch (...) {
155     //...
156
157     printGold(" ..... ");
158     printRed("FAIL");
159     std::cout << std::endl;
160     throw;
161 }
162
163
164 printGold(" ..... ");

```

```

165 printGreen("PASS");
166 std::cout << std::endl;
167 return 0;
168
169 } /* main() */

```

5.64 test/source/Storage/test_Lilon.cpp File Reference

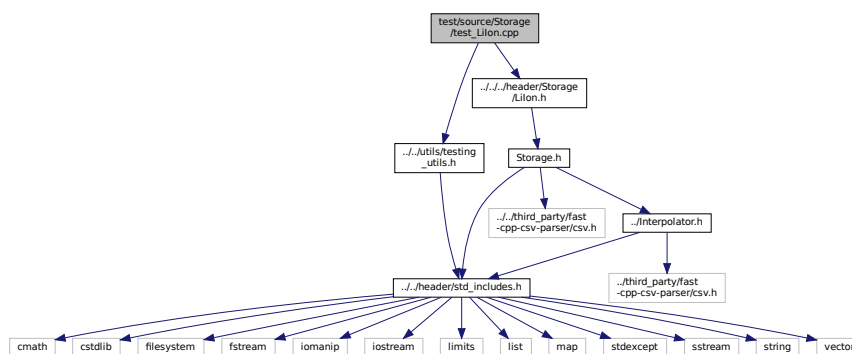
Testing suite for [Lilon](#) class.

```

#include "../..//utils/testing_utils.h"
#include "../..//header/Storage/LiIon.h"

```

Include dependency graph for test_Lilon.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.64.1 Detailed Description

Testing suite for [Lilon](#) class.

A suite of tests for the [Lilon](#) class.

5.64.2 Function Documentation

5.64.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Storage <-- LiIon");
33
34     srand(time(NULL));
35
36
37     try {
38
39 // ===== CONSTRUCTION ===== //
40
41     bool error_flag = true;
42
43     try {
44         LiIonInputs bad_liion_inputs;
45         bad_liion_inputs.min_SOC = -1;
46
47         LiIon bad_liion(8760, 1, bad_liion_inputs);
48
49         error_flag = false;
50     } catch (...) {
51         // Task failed successfully! =P
52     }
53     if (not error_flag) {
54         expectedErrorNotDetected(__FILE__, __LINE__);
55     }
56
57     LiIonInputs liion_inputs;
58
59     LiIon test_liion(8760, 1, liion_inputs);
60
61 // ===== END CONSTRUCTION ===== //
62
63
64
65 // ===== ATTRIBUTES ===== //
66
67     testTruth(
68         test_liion.type_str == "LIION",
69         __FILE__,
70         __LINE__
71 );
72
73     testFloatEquals(
74         test_liion.init_SOC,
75         0.5,
76         __FILE__,
77         __LINE__
78 );
79
80     testFloatEquals(
81         test_liion.min_SOC,
82         0.15,
83         __FILE__,
84         __LINE__
85 );
86
87     testFloatEquals(
88         test_liion.hysteresis_SOC,
89         0.5,
90         __FILE__,
91         __LINE__
92 );
93
94     testFloatEquals(
95         test_liion.max_SOC,
96         0.9,
97         __FILE__,
98         __LINE__
99 );
100
101     testFloatEquals(
102         test_liion.charging_efficiency,
103         0.9,
104         __FILE__,
105         __LINE__
106 );

```

```

107
108 testFloatEquals(
109     test_liion.discharging_efficiency,
110     0.9,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_liion.replace_SOH,
117     0.8,
118     __FILE__,
119     __LINE__
120 );
121
122 testFloatEquals(
123     test_liion.power_kW,
124     0,
125     __FILE__,
126     __LINE__
127 );
128
129 testFloatEquals(
130     test_liion.SOH_vec.size(),
131     8760,
132     __FILE__,
133     __LINE__
134 );
135
136 // ===== END ATTRIBUTES ===== //
137
138
139
140 // ===== METHODS ===== //
141
142 testFloatEquals(
143     test_liion.getAvailablekW(1),
144     100, // hits power capacity constraint
145     __FILE__,
146     __LINE__
147 );
148
149 testFloatEquals(
150     test_liion.getAcceptablekW(1),
151     100, // hits power capacity constraint
152     __FILE__,
153     __LINE__
154 );
155
156 test_liion.power_kW = 100;
157
158 testFloatEquals(
159     test_liion.getAvailablekW(1),
160     100, // hits power capacity constraint
161     __FILE__,
162     __LINE__
163 );
164
165 testFloatEquals(
166     test_liion.getAcceptablekW(1),
167     100, // hits power capacity constraint
168     __FILE__,
169     __LINE__
170 );
171
172 test_liion.power_kW = 1e6;
173
174 testFloatEquals(
175     test_liion.getAvailablekW(1),
176     0, // is already hitting power capacity constraint
177     __FILE__,
178     __LINE__
179 );
180
181 testFloatEquals(
182     test_liion.getAcceptablekW(1),
183     0, // is already hitting power capacity constraint
184     __FILE__,
185     __LINE__
186 );
187
188 test_liion.commitCharge(0, 1, 100);
189
190 testFloatEquals(
191     test_liion.power_kW,
192     0,
193     __FILE__,

```



```

194     __LINE__
195 );
196
197 // ===== END METHODS ===== //
198
199 } /* try */
200
201
202 catch (...) {
203     //...
204
205     printGold(" ..... ");
206     printRed("FAIL");
207     std::cout << std::endl;
208     throw;
209 }
210
211
212 printGold(" ..... ");
213 printGreen("PASS");
214 std::cout << std::endl;
215 return 0;
216 } /* main() */

```

5.65 test/source/Storage/test_Storage.cpp File Reference

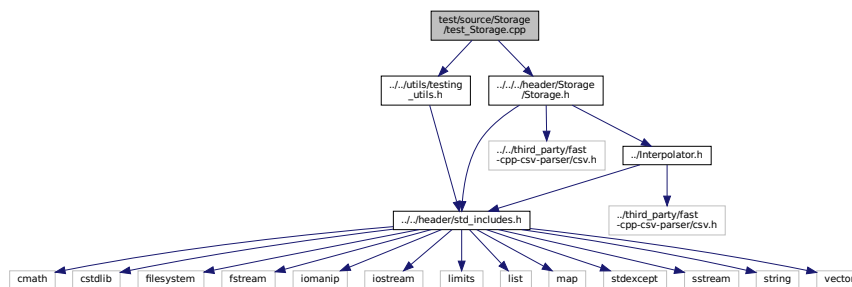
Testing suite for [Storage](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Storage/Storage.h"

```

Include dependency graph for test_Storage.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.65.1 Detailed Description

Testing suite for [Storage](#) class.

A suite of tests for the [Storage](#) class.

5.65.2 Function Documentation

5.65.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Storage");
33
34     srand(time(NULL));
35
36
37     try {
38
39 // ===== CONSTRUCTION ===== //
40
41     bool error_flag = true;
42
43     try {
44         StorageInputs bad_storage_inputs;
45         bad_storage_inputs.energy_capacity_kWh = 0;
46
47         Storage bad_storage(8760, 1, bad_storage_inputs);
48
49         error_flag = false;
50     } catch (...) {
51         // Task failed successfully! =P
52     }
53     if (not error_flag) {
54         expectedErrorNotDetected(__FILE__, __LINE__);
55     }
56
57     StorageInputs storage_inputs;
58
59     Storage test_storage(8760, 1, storage_inputs);
60
61 // ===== END CONSTRUCTION ===== //
62
63
64
65 // ===== ATTRIBUTES ===== //
66
67     testFloatEquals(
68         test_storage.power_capacity_kW,
69         100,
70         __FILE__,
71         __LINE__
72 );
73
74     testFloatEquals(
75         test_storage.energy_capacity_kWh,
76         1000,
77         __FILE__,
78         __LINE__
79 );
80
81     testFloatEquals(
82         test_storage.charge_vec_kWh.size(),
83         8760,
84         __FILE__,
85         __LINE__
86 );
87
88     testFloatEquals(
89         test_storage.charging_power_vec_kW.size(),
90         8760,
91         __FILE__,
92         __LINE__
93 );
94
95     testFloatEquals(
96         test_storage.discharging_power_vec_kW.size(),
97         8760,
98         __FILE__,
99         __LINE__
100 );
101
102     testFloatEquals(
103         test_storage.capital_cost_vec.size(),
104         8760,
105         __FILE__,
106         __LINE__

```

```

107 );
108
109 testFloatEquals(
110     test_storage.operation_maintenance_cost_vec.size(),
111     8760,
112     __FILE__,
113     __LINE__
114 );
115
116 // ===== END ATTRIBUTES ===== //
117
118
119
120 // ===== METHODS ===== //
121
122 //...
123
124 // ===== END METHODS ===== //
125
126 } /* try */
127
128
129 catch (...) {
130     //...
131
132     printGold(" ..... ");
133     printRed("FAIL");
134     std::cout << std::endl;
135     throw;
136 }
137
138
139 printGold(" ..... ");
140 printGreen("PASS");
141 std::cout << std::endl;
142 return 0;
143 } /* main() */

```

5.66 test/source/test_Controller.cpp File Reference

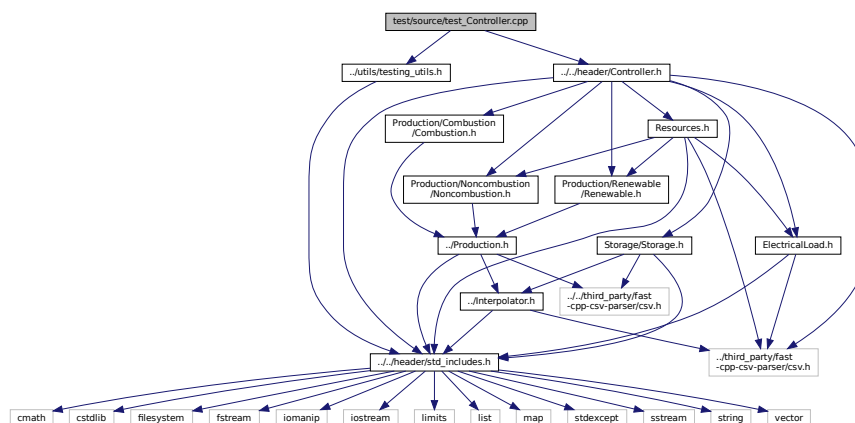
Testing suite for [Controller](#) class.

```

#include "../utils/testing_utils.h"
#include "../..//header/Controller.h"

```

Include dependency graph for test_Controller.cpp:



Functions

- [int main](#) (int argc, char **argv)

5.66.1 Detailed Description

Testing suite for [Controller](#) class.

A suite of tests for the [Controller](#) class.

5.66.2 Function Documentation

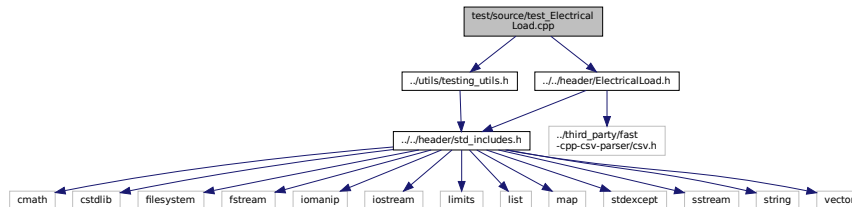
5.66.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Controller");
33
34     srand(time(NULL));
35
36
37 try {
38
39 // ===== CONSTRUCTION ===== //
40
41 Controller test_controller;
42
43 // ===== END CONSTRUCTION =====//
44
45
46
47 // ===== ATTRIBUTES ===== //
48
49 //...
50
51 // ===== END ATTRIBUTES ===== //
52
53
54
55 // ===== METHODS ===== //
56
57 //...
58
59 // ===== END METHODS ===== //
60
61 } /* try */
62
63
64 catch (...) {
65     //...
66
67     printGold(" ..... ");
68     printRed("FAIL");
69     std::cout << std::endl;
70     throw;
71 }
72
73
74 printGold(" ..... ");
75 printGreen("PASS");
76 std::cout << std::endl;
77 return 0;
78 } /* main() */
```

5.67 test/source/test_ElectricalLoad.cpp File Reference

Testing suite for [ElectricalLoad](#) class.

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



Functions

- int [main](#) (int argc, char **argv)

5.67.1 Detailed Description

Testing suite for [ElectricalLoad](#) class.

A suite of tests for the [ElectricalLoad](#) class.

5.67.2 Function Documentation

5.67.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting ElectricalLoad");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38
    39         // ===== CONSTRUCTION =====
    40
    41         std::string path_2_electrical_load_time_series =
    42             "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
    43
    44         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
    45
    46     }
```

```

46 // ===== END CONSTRUCTION ===== //
47
48
49
50 // ===== ATTRIBUTES ===== //
51
52 testTruth(
53     test_electrical_load.path_2_electrical_load_time_series ==
54     path_2_electrical_load_time_series,
55     __FILE__,
56     __LINE__
57 );
58
59 testFloatEquals(
60     test_electrical_load.n_points,
61     8760,
62     __FILE__,
63     __LINE__
64 );
65
66 testFloatEquals(
67     test_electrical_load.n_years,
68     0.999886,
69     __FILE__,
70     __LINE__
71 );
72
73 testFloatEquals(
74     test_electrical_load.min_load_kW,
75     82.1211213927802,
76     __FILE__,
77     __LINE__
78 );
79
80 testFloatEquals(
81     test_electrical_load.mean_load_kW,
82     258.373472633202,
83     __FILE__,
84     __LINE__
85 );
86
87
88 testFloatEquals(
89     test_electrical_load.max_load_kW,
90     500,
91     __FILE__,
92     __LINE__
93 );
94
95
96 std::vector<double> expected_dt_vec_hrs (48, 1);
97
98 std::vector<double> expected_time_vec_hrs = {
99     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
100    12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
101    24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
102    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
103 };
104
105 std::vector<double> expected_load_vec_kW = {
106    360.253836463674,
107    355.171277826775,
108    353.776453532298,
109    353.75405737934,
110    346.592867404975,
111    340.132411175118,
112    337.354867340578,
113    340.644115618736,
114    363.639028500678,
115    378.787797779238,
116    372.215798201712,
117    395.093925731298,
118    402.325427142659,
119    386.907725462306,
120    380.709170928091,
121    372.062070914977,
122    372.328646856954,
123    391.841444284136,
124    394.029351759596,
125    383.369407765254,
126    381.093099675206,
127    382.604158946193,
128    390.744843709034,
129    383.13949492437,
130    368.150393976985,
131    364.629744480226,
132    363.572736804082,

```

```

133     359.854924202248,
134     355.207590170267,
135     349.094656012401,
136     354.365935871597,
137     343.380608328546,
138     404.673065729266,
139     486.296896820126,
140     480.225974100847,
141     457.318764401085,
142     418.177339948609,
143     414.399018364126,
144     409.678420185754,
145     404.768766016563,
146     401.699589920585,
147     402.44339040654,
148     398.138372541906,
149     396.010498627646,
150     390.165117432277,
151     375.850429417013,
152     365.567100746484,
153     365.429624610923
154 };
155
156 for (int i = 0; i < 48; i++) {
157     testFloatEquals(
158         test_electrical_load.dt_vec_hrs[i],
159         expected_dt_vec_hrs[i],
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals(
165         test_electrical_load.time_vec_hrs[i],
166         expected_time_vec_hrs[i],
167         __FILE__,
168         __LINE__
169     );
170
171     testFloatEquals(
172         test_electrical_load.load_vec_kW[i],
173         expected_load_vec_kW[i],
174         __FILE__,
175         __LINE__
176     );
177 }
178
179 // ===== END ATTRIBUTES ===== //
180
181 } /* try */
182
183
184 catch (...) {
185     //...
186
187     printGold(" ..... ");
188     printRed("FAIL");
189     std::cout << std::endl;
190     throw;
191 }
192
193
194 printGold(" ..... ");
195 printGreen("PASS");
196 std::cout << std::endl;
197 return 0;
198 } /* main() */

```

5.68 test/source/test_Interpolator.cpp File Reference

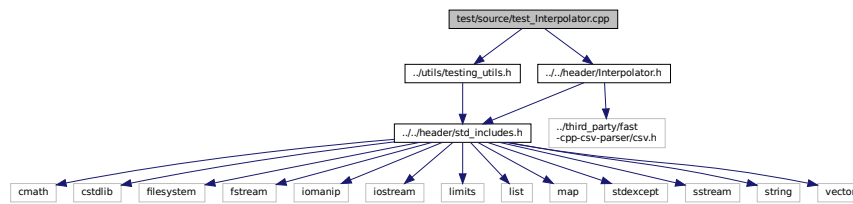
Testing suite for [Interpolator](#) class.

```

#include "../utils/testing_utils.h"
#include "../header/Interpolator.h"

```

Include dependency graph for test_Interpolator.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.68.1 Detailed Description

Testing suite for [Interpolator](#) class.

A suite of tests for the [Interpolator](#) class.

5.68.2 Function Documentation

5.68.2.1 main()

```

int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\n\tTesting Interpolator");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38
    39         // ===== CONSTRUCTION ===== //
    40
    41         Interpolator test_interpolator;
    42
    43         // ===== END CONSTRUCTION =====//
    44
    45
    46
    47         // ===== ATTRIBUTES ===== //
    48
    49         //...
    50
    51         // ===== END ATTRIBUTES ===== //
    52
    53
    54
    55         // ===== METHODS ===== //
    56

```



```

57 // 1. 1D interpolation
58
59 int data_key = 1;
60 std::string path_2_data = "data/test/interpolation/diesel_fuel_curve.csv";
61
62 test_interpolator.addData1D(data_key, path_2_data);
63
64 testTruth(
65     test_interpolator.path_map_1D[data_key] == path_2_data,
66     __FILE__,
67     __LINE__
68 );
69
70 testFloatEquals(
71     test_interpolator.interp_map_1D[data_key].n_points,
72     16,
73     __FILE__,
74     __LINE__
75 );
76
77 testFloatEquals(
78     test_interpolator.interp_map_1D[data_key].x_vec.size(),
79     16,
80     __FILE__,
81     __LINE__
82 );
83
84 std::vector<double> expected_x_vec = {
85     0,
86     0.3,
87     0.35,
88     0.4,
89     0.45,
90     0.5,
91     0.55,
92     0.6,
93     0.65,
94     0.7,
95     0.75,
96     0.8,
97     0.85,
98     0.9,
99     0.95,
100    1
101 };
102
103 std::vector<double> expected_y_vec = {
104     4.68079520372916,
105     11.1278522361839,
106     12.4787834830748,
107     13.7808847600209,
108     15.0417468303382,
109     16.277263,
110     17.4612831516442,
111     18.6279054806525,
112     19.7698039220515,
113     20.8893499214868,
114     21.955378,
115     23.0690535155297,
116     24.1323614374927,
117     25.1797231192866,
118     26.2122451458747,
119     27.254952
120 };
121
122 for (int i = 0; i < test_interpolator.interp_map_1D[data_key].n_points; i++) {
123     testFloatEquals(
124         test_interpolator.interp_map_1D[data_key].x_vec[i],
125         expected_x_vec[i],
126         __FILE__,
127         __LINE__
128     );
129
130     testFloatEquals(
131         test_interpolator.interp_map_1D[data_key].y_vec[i],
132         expected_y_vec[i],
133         __FILE__,
134         __LINE__
135     );
136 }
137
138 testFloatEquals(
139     test_interpolator.interp_map_1D[data_key].min_x,
140     expected_x_vec[0],
141     __FILE__,
142     __LINE__
143 );

```

```

144
145 testFloatEquals(
146     test_interpolator.interp_map_1D[data_key].max_x,
147     expected_x_vec[expected_x_vec.size() - 1],
148     __FILE__,
149     __LINE__
150 );
151
152 std::vector<double> interp_x_vec = {
153     0,
154     0.170812859791767,
155     0.322739274162545,
156     0.369750203682042,
157     0.443532869135929,
158     0.471567864244626,
159     0.536513734479662,
160     0.586125806988674,
161     0.601101175455075,
162     0.658356862575221,
163     0.70576929893201,
164     0.784069734739331,
165     0.805765927542453,
166     0.884747873186048,
167     0.930870496062112,
168     0.979415217694769,
169     1
170 };
171
172 std::vector<double> expected_interp_y_vec = {
173     4.68079520372916,
174     8.35159603357656,
175     11.7422361561399,
176     12.9931187917615,
177     14.8786636301325,
178     15.5746957307243,
179     17.1419229487141,
180     18.3041866133728,
181     18.6530540913696,
182     19.9569217633299,
183     21.012354614584,
184     22.7142305879957,
185     23.1916726441968,
186     24.8602332554707,
187     25.8172124624032,
188     26.8256741279932,
189     27.254952
190 };
191
192 for (size_t i = 0; i < interp_x_vec.size(); i++) {
193     testFloatEquals(
194         test_interpolator.interp1D(data_key, interp_x_vec[i]),
195         expected_interp_y_vec[i],
196         __FILE__,
197         __LINE__
198     );
199 }
200
201
202 // 2. 2D interpolation
203
204 data_key = 2;
205 path_2_data =
206     "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";
207
208 test_interpolator.addData2D(data_key, path_2_data);
209
210 testTruth(
211     test_interpolator.path_map_2D[data_key] == path_2_data,
212     __FILE__,
213     __LINE__
214 );
215
216 testFloatEquals(
217     test_interpolator.interp_map_2D[data_key].n_rows,
218     16,
219     __FILE__,
220     __LINE__
221 );
222
223 testFloatEquals(
224     test_interpolator.interp_map_2D[data_key].n_cols,
225     16,
226     __FILE__,
227     __LINE__
228 );
229
230 testFloatEquals(

```

```

231     test_interpolator.interp_map_2D[data_key].x_vec.size(),
232     16,
233     __FILE__,
234     __LINE__
235 );
236
237 testFloatEquals(
238     test_interpolator.interp_map_2D[data_key].y_vec.size(),
239     16,
240     __FILE__,
241     __LINE__
242 );
243
244 testFloatEquals(
245     test_interpolator.interp_map_2D[data_key].z_matrix.size(),
246     16,
247     __FILE__,
248     __LINE__
249 );
250
251 testFloatEquals(
252     test_interpolator.interp_map_2D[data_key].z_matrix[0].size(),
253     16,
254     __FILE__,
255     __LINE__
256 );
257
258 expected_x_vec = {
259     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
260 };
261
262 expected_y_vec = {
263     5,
264     6,
265     7,
266     8,
267     9,
268     10,
269     11,
270     12,
271     13,
272     14,
273     15,
274     16,
275     17,
276     18,
277     19,
278     20
279 };
280
281 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_cols; i++) {
282     testFloatEquals(
283         test_interpolator.interp_map_2D[data_key].x_vec[i],
284         expected_x_vec[i],
285         __FILE__,
286         __LINE__
287     );
288 }
289
290 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_rows; i++) {
291     testFloatEquals(
292         test_interpolator.interp_map_2D[data_key].y_vec[i],
293         expected_y_vec[i],
294         __FILE__,
295         __LINE__
296     );
297 }
298
299 testFloatEquals(
300     test_interpolator.interp_map_2D[data_key].min_x,
301     expected_x_vec[0],
302     __FILE__,
303     __LINE__
304 );
305
306 testFloatEquals(
307     test_interpolator.interp_map_2D[data_key].max_x,
308     expected_x_vec[expected_x_vec.size() - 1],
309     __FILE__,
310     __LINE__
311 );
312
313 testFloatEquals(
314     test_interpolator.interp_map_2D[data_key].min_y,
315     expected_y_vec[0],
316     __FILE__,
317     __LINE__

```

```

318 );
319
320 testFloatEquals(
321     test_interpolator.interp_map_2D[data_key].max_y,
322     expected_y_vec[expected_y_vec.size() - 1],
323     __FILE__,
324     __LINE__
325 );
326
327 std::vector<std::vector<double> expected_z_matrix = {
328     {0, 0.129128125, 0.268078125, 0.404253125, 0.537653125, 0.668278125, 0.796128125, 0.921203125, 1, 1,
329     1, 0, 0, 0, 0, 0},
330     {0, 0.11160375, 0.24944375, 0.38395375, 0.51513375, 0.64298375, 0.76750375, 0.88869375, 1, 1, 1, 1,
331     1, 1, 1},
332     {0, 0.094079375, 0.230809375, 0.363654375, 0.492614375, 0.617689375, 0.738879375, 0.856184375,
333     0.969604375, 1, 1, 1, 1, 1, 1, 1},
334     {0, 0.076555, 0.212175, 0.343355, 0.470095, 0.592395, 0.710255, 0.823675, 0.932655, 1, 1, 1, 1, 1,
335     1, 1},
336     {0, 0.059030625, 0.193540625, 0.323055625, 0.447575625, 0.567100625, 0.681630625, 0.791165625,
337     0.895705625, 0.995250625, 1, 1, 1, 1, 1, 1},
338     {0, 0.04150625, 0.17490625, 0.30275625, 0.42505625, 0.54180625, 0.65300625, 0.75865625, 0.85875625,
339     0.95330625, 1, 1, 1, 1, 1, 1},
340     {0, 0.023981875, 0.156271875, 0.282456875, 0.402536875, 0.516511875, 0.624381875, 0.726146875,
341     0.821806875, 0.911361875, 0.994811875, 1, 1, 1, 1, 1},
342     {0, 0.0064575, 0.1376375, 0.2621575, 0.3800175, 0.4912175, 0.5957575, 0.6936375, 0.7848575,
343     0.8694175, 0.9473175, 1, 1, 1, 1, 1},
344     {0, 0.119003125, 0.241858125, 0.357498125, 0.465923125, 0.567133125, 0.661128125, 0.747908125,
345     0.827473125, 0.899823125, 0.964958125, 1, 1, 1, 1, 1},
346     {0, 0.10036875, 0.22155875, 0.33497875, 0.44062875, 0.53850875, 0.62861875, 0.71095875,
347     0.78552875, 0.85232875, 0.91135875, 0.96261875, 1, 1, 1},
348     {0, 0.081734375, 0.201259375, 0.312459375, 0.415334375, 0.509884375, 0.596109375, 0.674009375,
349     0.743584375, 0.804834375, 0.857759375, 0.902359375, 0.938634375, 0.966584375, 0.986209375},
350     {0, 0.0631, 0.18096, 0.28994, 0.39004, 0.48126, 0.5636, 0.63706, 0.70164, 0.75734, 0.80416,
351     0.8421, 0.87116, 0.89134, 0.90264},
352     {0, 0.044465625, 0.160660625, 0.267420625, 0.364745625, 0.452635625, 0.531090625, 0.600110625,
353     0.659695625, 0.709845625, 0.750560625, 0.781840625, 0.803685624999999, 0.816095625, 0.819070625},
354     {0, 0.02583125, 0.14036125, 0.24490125, 0.33945125, 0.42401125, 0.49858125, 0.56316125,
355     0.61775125, 0.66235125, 0.69696125, 0.72158125, 0.73621125, 0.74085125, 0.73550125},
356     {0, 0.007196875, 0.120061875, 0.222381875, 0.314156875, 0.395386875, 0.466071875, 0.526211875,
357     0.575806875, 0.614856875, 0.643361875, 0.661321875, 0.668736875, 0.665606875, 0.651931875},
358     {0, 0, 0, 0.0997625, 0.1998625, 0.2888625, 0.3667625, 0.4335625, 0.4892625, 0.5338625, 0.5673625,
359     0.5897625, 0.6010625, 0.6012625, 0.5903625, 0.5683625}
360 };
361
362 for (int i = 0; i < test_interpolator.interp_map_2D[data_key].n_rows; i++) {
363     for (int j = 0; j < test_interpolator.interp_map_2D[data_key].n_cols; j++) {
364         testFloatEquals(
365             test_interpolator.interp_map_2D[data_key].z_matrix[i][j],
366             expected_z_matrix[i][j],
367             __FILE__,
368             __LINE__
369         );
370     }
371 }
372
373 interp_x_vec = {
374     0.389211848822208,
375     0.836477431896843,
376     1.52738334015579,
377     1.92640601114508,
378     2.27297317532019,
379     2.87416589636605,
380     3.72275770908175,
381     3.95063175885536,
382     4.68097139867404,
383     4.97775020449812,
384     5.55184219980547,
385     6.06566629451658,
386     6.27927876785062,
387     6.96218133671013,
388     7.51754442460228
389 };
390
391 std::vector<double> interp_y_vec = {
392     5.45741899698926,
393     6.00101329139007,
394     7.50567689404182,
395     8.77681262912881,
396     9.45143678206774,
397     10.7767876462885,
398     11.4795760857165,
399     12.9430684577599,
400     13.303544885703,
401     14.5069863517863,
402     15.1487890438045,
403     16.086524049077,
404     17.176609978648,

```

```

389     18.4155153740256,
390     19.1704554940162
391 };
392
393 std::vector<std::vector<double>> expected_interp_z_matrix = {
394     {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148
395     {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.882058
396     {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608
397     {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
398     {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
399     {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
400     {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
401     {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
402     {0,0.0196038727057393,0.18122235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
403     {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
404     {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
405     {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
406     {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
407     {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
408     {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
409 };
410
411 for (size_t i = 0; i < interp_y_vec.size(); i++) {
412     for (size_t j = 0; j < interp_x_vec.size(); j++) {
413         testFloatEquals(
414             test_interpolator.interp2D(data_key, interp_x_vec[j], interp_y_vec[i]),
415             expected_interp_z_matrix[i][j],
416             __FILE__,
417             __LINE__
418         );
419     }
420 }
421
422 // ===== END METHODS ===== //
423
424 } /* try */
425
426 catch (...) {
427     //...
428
429     printGold(" ..... ");
430     printRed("FAIL");
431     std::cout << std::endl;
432     throw;
433 }
434
435
436
437 printGold(" ..... ");
438 printGreen("PASS");
439 std::cout << std::endl;
440 return 0;
441 } /* main() */

```

5.69 test/source/test_Model.cpp File Reference

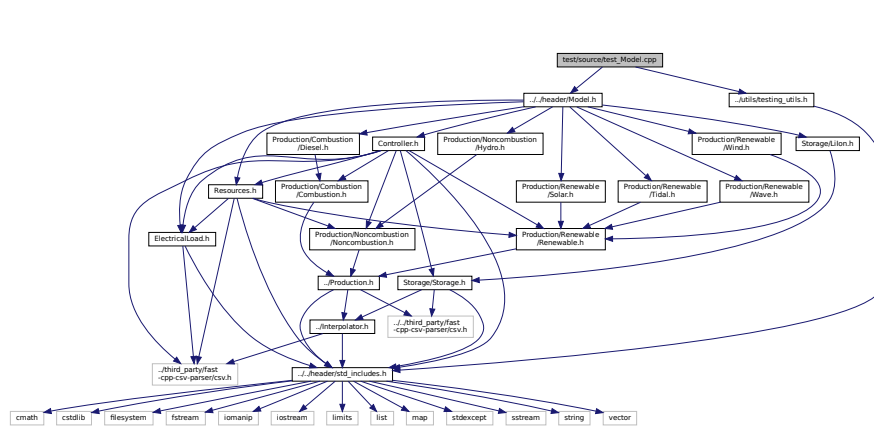
Testing suite for [Model](#) class.

```

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

```

Include dependency graph for test_Model.cpp:



Functions

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

5.69.1 Detailed Description

Testing suite for [Model](#) class.

A suite of tests for the [Model](#) class.

5.69.2 Function Documentation

5.69.2.1 main()

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

```

50     // Task failed successfully! =P
51 }
52 if (not error_flag) {
53     expectedErrorNotDetected(__FILE__, __LINE__);
54 }
55
56
57 try {
58     ModelInputs bad_model_inputs;
59     bad_model_inputs.path_2_electrical_load_time_series =
60         "data/test/electrical_load/bad_path_240984069830.csv";
61
62     Model bad_model(bad_model_inputs);
63
64     error_flag = false;
65 } catch (...) {
66     // Task failed successfully! =P
67 }
68 if (not error_flag) {
69     expectedErrorNotDetected(__FILE__, __LINE__);
70 }
71
72
73 std::string path_2_electrical_load_time_series =
74     "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
75
76 ModelInputs test_model_inputs;
77 test_model_inputs.path_2_electrical_load_time_series =
78     path_2_electrical_load_time_series;
79
80 Model test_model(test_model_inputs);
81
82 // ===== END CONSTRUCTION ===== //
83
84
85 // ===== ATTRIBUTES ===== //
86
87 testTruth(
88     test_model.electrical_load.path_2_electrical_load_time_series ==
89     path_2_electrical_load_time_series,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_model.electrical_load.n_points,
96     8760,
97     __FILE__,
98     __LINE__
99 );
100
101 testFloatEquals(
102     test_model.electrical_load.n_years,
103     0.999886,
104     __FILE__,
105     __LINE__
106 );
107
108 testFloatEquals(
109     test_model.electrical_load.min_load_kW,
110     82.1211213927802,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_model.electrical_load.mean_load_kW,
117     258.373472633202,
118     __FILE__,
119     __LINE__
120 );
121
122
123 testFloatEquals(
124     test_model.electrical_load.max_load_kW,
125     500,
126     __FILE__,
127     __LINE__
128 );
129
130
131 std::vector<double> expected_dt_vec_hrs (48, 1);
132
133 std::vector<double> expected_time_vec_hrs = {
134     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
135     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
136     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,

```

```

137     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
138 };
139
140 std::vector<double> expected_load_vec_kW = {
141     360.253836463674,
142     355.171277826775,
143     353.776453532298,
144     353.75405737934,
145     346.592867404975,
146     340.132411175118,
147     337.354867340578,
148     340.644115618736,
149     363.639028500678,
150     378.787797779238,
151     372.215798201712,
152     395.093925731298,
153     402.325427142659,
154     386.907725462306,
155     380.709170928091,
156     372.062070914977,
157     372.328646856954,
158     391.841444284136,
159     394.029351759596,
160     383.369407765254,
161     381.093099675206,
162     382.604158946193,
163     390.744843709034,
164     383.13949492437,
165     368.150393976985,
166     364.629744480226,
167     363.572736804082,
168     359.854924202248,
169     355.207590170267,
170     349.094656012401,
171     354.365935871597,
172     343.380608328546,
173     404.673065729266,
174     486.296896820126,
175     480.225974100847,
176     457.318764401085,
177     418.177339948609,
178     414.399018364126,
179     409.678420185754,
180     404.768766016563,
181     401.699589920585,
182     402.44339040654,
183     398.138372541906,
184     396.010498627646,
185     390.165117432277,
186     375.850429417013,
187     365.567100746484,
188     365.429624610923
189 };
190
191 for (int i = 0; i < 48; i++) {
192     testFloatEquals(
193         test_model.electrical_load.dt_vec_hrs[i],
194         expected_dt_vec_hrs[i],
195         __FILE__,
196         __LINE__
197     );
198
199     testFloatEquals(
200         test_model.electrical_load.time_vec_hrs[i],
201         expected_time_vec_hrs[i],
202         __FILE__,
203         __LINE__
204     );
205
206     testFloatEquals(
207         test_model.electrical_load.load_vec_kW[i],
208         expected_load_vec_kW[i],
209         __FILE__,
210         __LINE__
211     );
212 }
213
214 // ===== END ATTRIBUTES ===== //
215
216
217
218 // ===== METHODS ===== //
219
220 // add Solar resource
221 int solar_resource_key = 0;
222 std::string path_2_solar_resource_data =
223     "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

```



```

224
225 test_model.addResource(
226     RenewableType :: SOLAR,
227     path_2_solar_resource_data,
228     solar_resource_key
229 );
230
231 std::vector<double> expected_solar_resource_vec_kWm2 = {
232     0,
233     0,
234     0,
235     0,
236     0,
237     0,
238     8.51702662684015E-05,
239     0.000348341567045,
240     0.00213793728593,
241     0.004099863613322,
242     0.000997135230553,
243     0.009534527624657,
244     0.022927996790616,
245     0.0136071715294,
246     0.002535134127751,
247     0.005206897515821,
248     0.005627658648597,
249     0.000701186722215,
250     0.00017119827089,
251     0,
252     0,
253     0,
254     0,
255     0,
256     0,
257     0,
258     0,
259     0,
260     0,
261     0,
262     0,
263     0.000141055102242,
264     0.00084525014743,
265     0.024893647822702,
266     0.091245556190749,
267     0.158722176731637,
268     0.152859680515876,
269     0.149922903895116,
270     0.13049996570866,
271     0.03081254222795,
272     0.001218928911125,
273     0.000206092647423,
274     0,
275     0,
276     0,
277     0,
278     0,
279     0
280 };
281
282 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
283     testFloatEquals(
284         test_model.resources.resource_map_1D[solar_resource_key][i],
285         expected_solar_resource_vec_kWm2[i],
286         __FILE__,
287         __LINE__
288     );
289 }
290
291
292 // add Tidal resource
293 int tidal_resource_key = 1;
294 std::string path_2_tidal_resource_data =
295     "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
296
297 test_model.addResource(
298     RenewableType :: TIDAL,
299     path_2_tidal_resource_data,
300     tidal_resource_key
301 );
302
303
304 // add Wave resource
305 int wave_resource_key = 2;
306 std::string path_2_wave_resource_data =
307     "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
308
309 test_model.addResource(
310     RenewableType :: WAVE,

```

```

311     path_2_wave_resource_data,
312     wave_resource_key
313 );
314
315
316 // add Wind resource
317 int wind_resource_key = 3;
318 std::string path_2_wind_resource_data =
319     "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
320
321 test_model.addResource(
322     RenewableType :: WIND,
323     path_2_wind_resource_data,
324     wind_resource_key
325 );
326
327
328 // add Hydro resource
329 int hydro_resource_key = 4;
330 std::string path_2_hydro_resource_data =
331     "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
332
333 test_model.addResource(
334     NoncombustionType :: HYDRO,
335     path_2_hydro_resource_data,
336     hydro_resource_key
337 );
338
339
340 // add Hydro asset
341 HydroInputs hydro_inputs;
342 hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
343 hydro_inputs.reservoir_capacity_m3 = 10000;
344 hydro_inputs.init_reservoir_state = 0.5;
345 hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
346 hydro_inputs.resource_key = hydro_resource_key;
347
348 test_model.addHydro(hydro_inputs);
349
350 testFloatEquals(
351     test_model.noncombustion_ptr_vec.size(),
352     1,
353     __FILE__,
354     __LINE__
355 );
356
357 testFloatEquals(
358     test_model.noncombustion_ptr_vec[0]->type,
359     NoncombustionType :: HYDRO,
360     __FILE__,
361     __LINE__
362 );
363
364 testFloatEquals(
365     test_model.noncombustion_ptr_vec[0]->resource_key,
366     hydro_resource_key,
367     __FILE__,
368     __LINE__
369 );
370
371
372 // add Diesel assets
373 DieselInputs diesel_inputs;
374 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
375 diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;
376
377 test_model.addDiesel(diesel_inputs);
378
379 testFloatEquals(
380     test_model.combustion_ptr_vec.size(),
381     1,
382     __FILE__,
383     __LINE__
384 );
385
386 testFloatEquals(
387     test_model.combustion_ptr_vec[0]->type,
388     CombustionType :: DIESEL,
389     __FILE__,
390     __LINE__
391 );
392
393 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
394
395 test_model.addDiesel(diesel_inputs);
396
397 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;

```

```

398
399 test_model.addDiesel(diesel_inputs);
400
401 testFloatEquals(
402     test_model.combustion_ptr_vec.size(),
403     3,
404     __FILE__,
405     __LINE__
406 );
407
408 std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
409
410 for (int i = 0; i < 3; i++) {
411     testFloatEquals(
412         test_model.combustion_ptr_vec[i]->capacity_kW,
413         expected_diesel_capacity_vec_kW[i],
414         __FILE__,
415         __LINE__
416     );
417 }
418
419 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
420
421 for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
422     test_model.addDiesel(diesel_inputs);
423 }
424
425
426 // add Solar asset
427 SolarInputs solar_inputs;
428 solar_inputs.resource_key = solar_resource_key;
429
430 test_model.addSolar(solar_inputs);
431
432 testFloatEquals(
433     test_model.renewable_ptr_vec.size(),
434     1,
435     __FILE__,
436     __LINE__
437 );
438
439 testFloatEquals(
440     test_model.renewable_ptr_vec[0]->type,
441     RenewableType :: SOLAR,
442     __FILE__,
443     __LINE__
444 );
445
446
447 // add Tidal asset
448 TidalInputs tidal_inputs;
449 tidal_inputs.resource_key = tidal_resource_key;
450
451 test_model.addTidal(tidal_inputs);
452
453 testFloatEquals(
454     test_model.renewable_ptr_vec.size(),
455     2,
456     __FILE__,
457     __LINE__
458 );
459
460 testFloatEquals(
461     test_model.renewable_ptr_vec[1]->type,
462     RenewableType :: TIDAL,
463     __FILE__,
464     __LINE__
465 );
466
467
468 // add Wave asset
469 WaveInputs wave_inputs;
470 wave_inputs.resource_key = wave_resource_key;
471
472 test_model.addWave(wave_inputs);
473
474 testFloatEquals(
475     test_model.renewable_ptr_vec.size(),
476     3,
477     __FILE__,
478     __LINE__
479 );
480
481 testFloatEquals(
482     test_model.renewable_ptr_vec[2]->type,
483     RenewableType :: WAVE,
484     __FILE__,

```

```

485     __LINE__
486 );
487
488
489 // add Wind asset
490 WindInputs wind_inputs;
491 wind_inputs.resource_key = wind_resource_key;
492
493 test_model.addWind(wind_inputs);
494
495 testFloatEquals(
496     test_model.renewable_ptr_vec.size(),
497     4,
498     __FILE__,
499     __LINE__
500 );
501
502 testFloatEquals(
503     test_model.renewable_ptr_vec[3]->type,
504     RenewableType :: WIND,
505     __FILE__,
506     __LINE__
507 );
508
509
510 // add LiIon asset
511 LiIonInputs liion_inputs;
512
513 test_model.addLiIon(liion_inputs);
514
515 testFloatEquals(
516     test_model.storage_ptr_vec.size(),
517     1,
518     __FILE__,
519     __LINE__
520 );
521
522 testFloatEquals(
523     test_model.storage_ptr_vec[0]->type,
524     StorageType :: LIION,
525     __FILE__,
526     __LINE__
527 );
528
529
530 // run
531 test_model.run();
532
533
534 // write results
535 test_model.writeResults("test/test_results/");
536
537
538 // test post-run attributes
539 double net_load_kW;
540
541 Combustion* combustion_ptr;
542 Noncombustion* noncombustion_ptr;
543 Renewable* renewable_ptr;
544 Storage* storage_ptr;
545
546 for (int i = 0; i < test_model.electrical_load.n_points; i++) {
547     net_load_kW = test_model.controller.net_load_vec_kW[i];
548
549     testLessThanOrEqualTo(
550         test_model.controller.net_load_vec_kW[i],
551         test_model.electrical_load.max_load_kW,
552         __FILE__,
553         __LINE__
554     );
555
556     for (size_t j = 0; j < test_model.combustion_ptr_vec.size(); j++) {
557         combustion_ptr = test_model.combustion_ptr_vec[j];
558
559         testFloatEquals(
560             combustion_ptr->production_vec_kW[i] -
561             combustion_ptr->dispatch_vec_kW[i] -
562             combustion_ptr->curtailment_vec_kW[i] -
563             combustion_ptr->storage_vec_kW[i],
564             0,
565             __FILE__,
566             __LINE__
567         );
568
569         net_load_kW -= combustion_ptr->production_vec_kW[i];
570     }
571 }

```

```

572     for (size_t j = 0; j < test_model.noncombustion_ptr_vec.size(); j++) {
573         noncombustion_ptr = test_model.noncombustion_ptr_vec[j];
574
575         testFloatEquals(
576             noncombustion_ptr->production_vec_kW[i] -
577             noncombustion_ptr->dispatch_vec_kW[i] -
578             noncombustion_ptr->curtailment_vec_kW[i] -
579             noncombustion_ptr->storage_vec_kW[i],
580             0,
581             __FILE__,
582             __LINE__
583         );
584
585         net_load_kW -= noncombustion_ptr->production_vec_kW[i];
586     }
587
588     for (size_t j = 0; j < test_model.renewable_ptr_vec.size(); j++) {
589         renewable_ptr = test_model.renewable_ptr_vec[j];
590
591         testFloatEquals(
592             renewable_ptr->production_vec_kW[i] -
593             renewable_ptr->dispatch_vec_kW[i] -
594             renewable_ptr->curtailment_vec_kW[i] -
595             renewable_ptr->storage_vec_kW[i],
596             0,
597             __FILE__,
598             __LINE__
599         );
600
601         net_load_kW -= renewable_ptr->production_vec_kW[i];
602     }
603
604     for (size_t j = 0; j < test_model.storage_ptr_vec.size(); j++) {
605         storage_ptr = test_model.storage_ptr_vec[j];
606
607         testTruth(
608             not (
609                 storage_ptr->charging_power_vec_kW[i] > 0 and
610                 storage_ptr->discharging_power_vec_kW[i] > 0
611             ),
612             __FILE__,
613             __LINE__
614         );
615
616         net_load_kW -= storage_ptr->discharging_power_vec_kW[i];
617     }
618
619     testLessThanOrEqualTo(
620         net_load_kW,
621         0,
622         __FILE__,
623         __LINE__
624     );
625 }
626
627 testGreaterThan(
628     test_model.net_present_cost,
629     0,
630     __FILE__,
631     __LINE__
632 );
633
634 testFloatEquals(
635     test_model.total_dispatch_discharge_kWh,
636     2263351.62026685,
637     __FILE__,
638     __LINE__
639 );
640
641 testGreaterThan(
642     test_model.levellized_cost_of_energy_kWh,
643     0,
644     __FILE__,
645     __LINE__
646 );
647
648 testGreaterThan(
649     test_model.total_fuel_consumed_L,
650     0,
651     __FILE__,
652     __LINE__
653 );
654
655 testGreaterThan(
656     test_model.total_emissions.CO2_kg,
657     0,
658     __FILE__,

```

```

659     __LINE__
660 );
661
662 testGreaterThan(
663     test_model.total_emissions.CO_kg,
664     0,
665     __FILE__,
666     __LINE__
667 );
668
669 testGreaterThan(
670     test_model.total_emissions.NOx_kg,
671     0,
672     __FILE__,
673     __LINE__
674 );
675
676 testGreaterThan(
677     test_model.total_emissions.SOx_kg,
678     0,
679     __FILE__,
680     __LINE__
681 );
682
683 testGreaterThan(
684     test_model.total_emissions.CH4_kg,
685     0,
686     __FILE__,
687     __LINE__
688 );
689
690 testGreaterThan(
691     test_model.total_emissions.PM_kg,
692     0,
693     __FILE__,
694     __LINE__
695 );
696
697 // ===== END METHODS ===== //
698
699 } /* try */
700
701
702 catch (...) {
703     //...
704
705     printGold(" ..... ");
706     printRed("FAIL");
707     std::cout << std::endl;
708     throw;
709 }
710
711
712 printGold(" ..... ");
713 printGreen("PASS");
714 std::cout << std::endl;
715 return 0;
716 } /* main() */

```

5.70 test/source/test_Resources.cpp File Reference

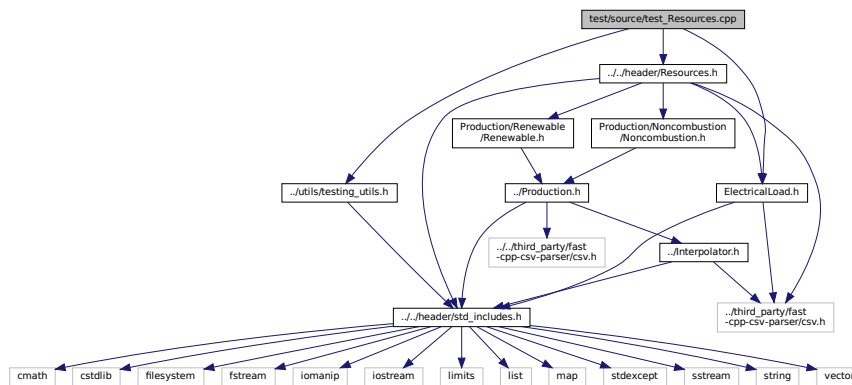
Testing suite for [Resources](#) class.

```

#include "../utils/testing_utils.h"
#include "../header/Resources.h"
#include "../header/ElectricalLoad.h"

```

Include dependency graph for test_Resources.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.70.1 Detailed Description

Testing suite for [Resources](#) class.

A suite of tests for the [Resources](#) class.

5.70.2 Function Documentation

5.70.2.1 main()

```

int main (
    int argc,
    char ** argv )

28 {
29     #ifdef _WIN32
30         activateVirtualTerminal();
31     #endif /* _WIN32 */
32
33     printGold("\tTesting Resources");
34
35     srand(time(NULL));
36
37
38     try {
39         // ===== CONSTRUCTION ===== //
40
41         std::string path_2_electrical_load_time_series =
42             "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
43
44         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
45
46         Resources test_resources;
47
48     }
  
```

```

49 // ===== END CONSTRUCTION ===== //
50
51
52
53 // ===== ATTRIBUTES ===== //
54
55 testFloatEquals(
56     test_resources.resource_map_1D.size(),
57     0,
58     __FILE__,
59     __LINE__
60 );
61
62 testFloatEquals(
63     test_resources.path_map_1D.size(),
64     0,
65     __FILE__,
66     __LINE__
67 );
68
69 testFloatEquals(
70     test_resources.resource_map_2D.size(),
71     0,
72     __FILE__,
73     __LINE__
74 );
75
76 testFloatEquals(
77     test_resources.path_map_2D.size(),
78     0,
79     __FILE__,
80     __LINE__
81 );
82
83 // ===== END ATTRIBUTES ===== //
84
85
86 // ===== METHODS ===== //
87
88 int solar_resource_key = 0;
89 std::string path_2_solar_resource_data =
90     "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
91
92 test_resources.addResource(
93     RenewableType::SOLAR,
94     path_2_solar_resource_data,
95     solar_resource_key,
96     &test_electrical_load
97 );
98
99 bool error_flag = true;
100 try {
101     test_resources.addResource(
102         RenewableType::SOLAR,
103         path_2_solar_resource_data,
104         solar_resource_key,
105         &test_electrical_load
106     );
107
108     error_flag = false;
109 } catch (...) {
110     // Task failed successfully! =P
111 }
112 if (not error_flag) {
113     expectedErrorNotDetected(__FILE__, __LINE__);
114 }
115
116
117 try {
118     std::string path_2_solar_resource_data_BAD_TIMES =
119         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";
120
121     test_resources.addResource(
122         RenewableType::SOLAR,
123         path_2_solar_resource_data_BAD_TIMES,
124         -1,
125         &test_electrical_load
126     );
127
128     error_flag = false;
129 } catch (...) {
130     // Task failed successfully! =P
131 }
132 if (not error_flag) {
133     expectedErrorNotDetected(__FILE__, __LINE__);
134 }
135

```



```

136
137 try {
138     std::string path_2_solar_resource_data_BAD_LENGTH =
139         "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";
140
141     test_resources.addResource(
142         RenewableType::SOLAR,
143         path_2_solar_resource_data_BAD_LENGTH,
144         -2,
145         &test_electrical_load
146     );
147
148     error_flag = false;
149 } catch (...) {
150     // Task failed successfully! =P
151 }
152 if (not error_flag) {
153     expectedErrorNotDetected(__FILE__, __LINE__);
154 }
155
156 std::vector<double> expected_solar_resource_vec_kWm2 = {
157     0,
158     0,
159     0,
160     0,
161     0,
162     0,
163     8.51702662684015E-05,
164     0.000348341567045,
165     0.00213793728593,
166     0.004099863613322,
167     0.000997135230553,
168     0.009534527624657,
169     0.022927996790616,
170     0.0136071715294,
171     0.002535134127751,
172     0.005206897515821,
173     0.005627658648597,
174     0.000701186722215,
175     0.00017119827089,
176     0,
177     0,
178     0,
179     0,
180     0,
181     0,
182     0,
183     0,
184     0,
185     0,
186     0,
187     0,
188     0.000141055102242,
189     0.00084525014743,
190     0.024893647822702,
191     0.091245556190749,
192     0.158722176731637,
193     0.152859680515876,
194     0.149922903895116,
195     0.13049996570866,
196     0.03081254222795,
197     0.001218928911125,
198     0.000206092647423,
199     0,
200     0,
201     0,
202     0,
203     0,
204     0
205 };
206
207 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
208     testFloatEquals(
209         test_resources.resource_map_1D[solar_resource_key][i],
210         expected_solar_resource_vec_kWm2[i],
211         __FILE__,
212         __LINE__
213     );
214 }
215
216
217 int tidal_resource_key = 1;
218 std::string path_2_tidal_resource_data =
219     "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
220
221 test_resources.addResource(
222     RenewableType::TIDAL,

```

```

223     path_2_tidal_resource_data,
224     tidal_resource_key,
225     &test_electrical_load
226 );
227
228 std::vector<double> expected_tidal_resource_vec_ms = {
229     0.347439913040533,
230     0.770545522195602,
231     0.731352084836198,
232     0.293389814389542,
233     0.209959110813115,
234     0.610609623896497,
235     1.78067162013604,
236     2.53522775118089,
237     2.75966627832024,
238     2.52101111143895,
239     2.05389330201031,
240     1.3461515862445,
241     0.28909254878384,
242     0.897754086048563,
243     1.71406453837407,
244     1.85047408742869,
245     1.71507908595979,
246     1.33540349705416,
247     0.434586143463003,
248     0.500623815700637,
249     1.37172172646733,
250     1.68294125491228,
251     1.56101300975417,
252     1.04925834219412,
253     0.211395463930223,
254     1.03720048903385,
255     1.85059536356448,
256     1.85203242794517,
257     1.4091471616277,
258     0.767776539039899,
259     0.251464906990961,
260     1.47018469375652,
261     2.36260493698197,
262     2.46653750048625,
263     2.12851908739291,
264     1.62783753197988,
265     0.734594890957439,
266     0.441886297300355,
267     1.6574418350918,
268     2.0684558286637,
269     1.87717416992136,
270     1.58871262337931,
271     1.03451227609235,
272     0.193371305159817,
273     0.976400122458815,
274     1.6583227369707,
275     1.76690616570953,
276     1.54801328553115
277 };
278
279 for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
280     testFloatEquals(
281         test_resources.resource_map_1D[tidal_resource_key][i],
282         expected_tidal_resource_vec_ms[i],
283         __FILE__,
284         __LINE__
285     );
286 }
287
288
289 int wave_resource_key = 2;
290 std::string path_2_wave_resource_data =
291     "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
292
293 test_resources.addResource(
294     RenewableType::WAVE,
295     path_2_wave_resource_data,
296     wave_resource_key,
297     &test_electrical_load
298 );
299
300 std::vector<double> expected_significant_wave_height_vec_m = {
301     4.26175222125028,
302     4.25020976167872,
303     4.25656524330349,
304     4.27193854786718,
305     4.28744955711233,
306     4.29421815278154,
307     4.2839937266082,
308     4.25716982457976,
309     4.22419391611483,

```

```
310     4.19588925217606,  
311     4.17338788587412,  
312     4.14672746914214,  
313     4.10560041173665,  
314     4.05074966447193,  
315     3.9953696962433,  
316     3.95316976150866,  
317     3.92771018142378,  
318     3.91129562488595,  
319     3.89558312094911,  
320     3.87861093931749,  
321     3.86538307240754,  
322     3.86108961027929,  
323     3.86459448853189,  
324     3.86796474016882,  
325     3.86357412779993,  
326     3.85554872014731,  
327     3.86044266668675,  
328     3.89445961915999,  
329     3.95554798115731,  
330     4.02265508610476,  
331     4.07419587011404,  
332     4.10314247143958,  
333     4.11738045085928,  
334     4.12554995596708,  
335     4.12923992001675,  
336     4.1229292327442,  
337     4.10123955307441,  
338     4.06748827895363,  
339     4.0336230651344,  
340     4.01134236393876,  
341     4.00136570034559,  
342     3.99368787690411,  
343     3.97820924247644,  
344     3.95369335178055,  
345     3.92742545608532,  
346     3.90683362771686,  
347     3.89331520944006,  
348     3.88256045801583  
349 };  
350  
351 std::vector<double> expected_energy_period_vec_s = {  
352     10.4456008226821,  
353     10.4614151137651,  
354     10.4462827795433,  
355     10.4127692097884,  
356     10.3734397942723,  
357     10.3408599227669,  
358     10.32637292093,  
359     10.3245412676322,  
360     10.310409818185,  
361     10.2589529840966,  
362     10.1728100603103,  
363     10.0862908658929,  
364     10.03480243813,  
365     10.023673635806,  
366     10.0243418565116,  
367     10.0063487117653,  
368     9.96050302286607,  
369     9.9011999635568,  
370     9.84451822125472,  
371     9.79726875879626,  
372     9.75614594835158,  
373     9.7173447961368,  
374     9.68342904390577,  
375     9.66380508567062,  
376     9.6674009575699,  
377     9.68927134575103,  
378     9.70979984863046,  
379     9.70967357906908,  
380     9.68983025704562,  
381     9.6722855524805,  
382     9.67973599910003,  
383     9.71977125328293,  
384     9.78450442291421,  
385     9.86532355233449,  
386     9.96158937600019,  
387     10.0807018356507,  
388     10.2291022504937,  
389     10.39458528356,  
390     10.5464393581004,  
391     10.6553277500484,  
392     10.7245553190084,  
393     10.7893127285064,  
394     10.8846512240849,  
395     11.0148158739075,  
396     11.1544325654719,
```

```

397     11.2772785848343,
398     11.3744362756187,
399     11.4533643503183
400 };
401
402 for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
403     testFloatEquals(
404         test_resources.resource_map_2D[wave_resource_key][i][0],
405         expected_significant_wave_height_vec_m[i],
406         __FILE__,
407         __LINE__
408     );
409
410     testFloatEquals(
411         test_resources.resource_map_2D[wave_resource_key][i][1],
412         expected_energy_period_vec_s[i],
413         __FILE__,
414         __LINE__
415     );
416 }
417
418
419 int wind_resource_key = 3;
420 std::string path_2_wind_resource_data =
421     "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";
422
423 test_resources.addResource(
424     RenewableType::WIND,
425     path_2_wind_resource_data,
426     wind_resource_key,
427     &test_electrical_load
428 );
429
430 std::vector<double> expected_wind_resource_vec_ms = {
431     6.88566688469997,
432     5.02177105466549,
433     3.74211715899568,
434     5.67169579985362,
435     4.90670669971858,
436     4.29586955031368,
437     7.41155377205065,
438     10.2243290476943,
439     13.1258696725555,
440     13.7016198628274,
441     16.2481482330233,
442     16.5096744355418,
443     13.4354482206162,
444     14.0129230731609,
445     14.5554549260515,
446     13.4454539065912,
447     13.3447169512094,
448     11.7372615098554,
449     12.7200070078013,
450     10.6421127908149,
451     6.09869498990661,
452     5.66355596602321,
453     4.97316966910831,
454     3.48937138360567,
455     2.15917470979169,
456     1.29061103587027,
457     3.43475751425219,
458     4.11706326260927,
459     4.28905275747408,
460     5.75850263196241,
461     8.98293663055264,
462     11.7069822941315,
463     12.4031987075858,
464     15.4096570910089,
465     16.6210843829552,
466     13.3421219142573,
467     15.2112831900548,
468     18.350864533037,
469     15.8751799822971,
470     15.3921198799796,
471     15.9729192868434,
472     12.4728950178772,
473     10.177050481096,
474     10.7342247355551,
475     8.98846695631389,
476     4.14671169124739,
477     3.17256452697149,
478     3.40036336968628
479 };
480
481 for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
482     testFloatEquals(
483         test_resources.resource_map_1D[wind_resource_key][i],

```

```

484         expected_wind_resource_vec_ms[i],
485         __FILE__,
486         __LINE__
487     );
488 }
489
490
491 int hydro_resource_key = 4;
492 std::string path_2_hydro_resource_data =
493     "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";
494
495 test_resources.addResource(
496     NoncombustionType::HYDRO,
497     path_2_hydro_resource_data,
498     hydro_resource_key,
499     &test_electrical_load
500 );
501
502 std::vector<double> expected_hydro_resource_vec_m3hr = {
503     2167.91531556942,
504     2046.58261560569,
505     2007.85941123153,
506     2000.11477247929,
507     1917.50527264453,
508     1963.97311577093,
509     1908.46985899809,
510     1886.5267112678,
511     1965.26388854254,
512     1953.64692935289,
513     2084.01504296306,
514     2272.46796101188,
515     2520.29645627096,
516     2715.203242423,
517     2720.36633563203,
518     3130.83228077221,
519     3289.59741021591,
520     3981.45195965772,
521     5295.45929491303,
522     7084.47124360523,
523     7709.20557708454,
524     7436.85238642936,
525     7235.49173429668,
526     6710.14695517339,
527     6015.71085806577,
528     5279.97001316337,
529     4877.24870889801,
530     4421.60569340303,
531     3919.49483690424,
532     3498.70270322341,
533     3274.10813058883,
534     3147.61233529349,
535     2904.94693324343,
536     2805.55738101,
537     2418.32535637171,
538     2398.96375630723,
539     2260.85100182222,
540     2157.58912702878,
541     2019.47637254377,
542     1913.63295220712,
543     1863.29279076589,
544     1748.41395678279,
545     1695.49224555317,
546     1599.97501375715,
547     1559.96103873397,
548     1505.74855473274,
549     1438.62833664765,
550     1384.41585476901
551 };
552
553 for (size_t i = 0; i < expected_hydro_resource_vec_m3hr.size(); i++) {
554     testFloatEquals(
555         test_resources.resource_map_1D[hydro_resource_key][i],
556         expected_hydro_resource_vec_m3hr[i],
557         __FILE__,
558         __LINE__
559     );
560 }
561
562 // ===== END METHODS ===== //
563
564 } /* try */
565
566
567 catch (...) {
568     printGold(" ..... ");
569     printRed("FAIL");
570     std::cout << std::endl;

```

```

571     throw;
572 }
573
574
575 printGold(" ..... ");
576 printGreen("PASS");
577 std::cout << std::endl;
578 return 0;
579 } /* main() */

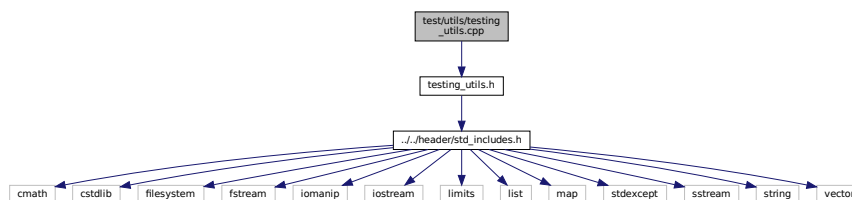
```

5.71 test/utls/testing_utils.cpp File Reference

Header file for various PGMcpp testing utilities.

```
#include "testing_utils.h"
```

Include dependency graph for testing_utils.cpp:



Functions

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

5.71.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.71.2 Function Documentation

5.71.2.1 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 "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

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

5.71.2.2 printGold()

```
void printGold (
    std::string input_str )
```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```
84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */
```

5.71.2.3 printGreen()

```
void printGreen (
    std::string input_str )
```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

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

5.71.2.4 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 std::cout.
------------------	---

```
104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */
```

5.71.2.5 testFloatEquals()

```
void testFloatEquals (
    double x,
    double y,
    std::string file,
    int line )
```

Tests for the equality of two floating point numbers *x* and *y* (to within `FLOAT_TOLERANCE`).

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```
138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
```



```

141     }
142
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154
155     #ifdef _WIN32
156         std::cout << error_str << std::endl;
157     #endif
158
159     throw std::runtime_error(error_str);
160     return;
161 } /* testFloatEquals() */

```

5.71.2.6 testGreaterThan()

```

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

```

Tests if $x > y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209
210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */

```

5.71.2.7 testGreaterThanOrEqualTo()

```

void testGreaterThanOrEqualTo (
    double x,

```

```
double y,
std::string file,
int line )
```

Tests if $x \geq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

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

5.71.2.8 testLessThan()

```
void testLessThan (
    double x,
    double y,
    std::string file,
    int line )
```

Tests if $x < y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
293 {
294     if (x < y) {
295         return;
296     }
297
298     std::string error_str = "ERROR: testLessThan():\t in ";
299     error_str += file;
300     error_str += "\tline ";
301     error_str += std::to_string(line);
302     error_str += ":\t\n";
```

```

303     error_str += std::to_string(x);
304     error_str += " is not less than ";
305     error_str += std::to_string(y);
306     error_str += "\n";
307
308     #ifdef _WIN32
309         std::cout << error_str << std::endl;
310     #endif
311
312     throw std::runtime_error(error_str);
313     return;
314 } /* testLessThan() */

```

5.71.2.9 testLessThanOrEqualTo()

```

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

```

Tests if $x \leq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.71.2.10 testTruth()

```

void testTruth (
    bool statement,
    std::string file,
    int line )

```

Tests if the given statement is true.

Parameters

<i>statement</i>	The statement whose truth is to be tested ("1 == 0", for example).
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

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

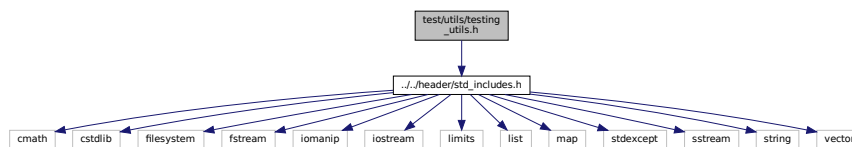
```

5.72 test/utlis/testing_utils.h File Reference

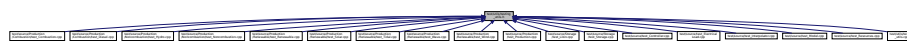
Header file for various PGMcpp testing utilities.

```
#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.

5.72.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.72.2 Macro Definition Documentation

5.72.2.1 `FLOAT_TOLERANCE`

```
#define FLOAT_TOLERANCE 1e-6
```

A tolerance for application to floating point equality tests.

5.72.3 Function Documentation

5.72.3.1 `expectedErrorNotDetected()`

```
void expectedErrorNotDetected (  
    std::string file,  
    int line )
```

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

Parameters

<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

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

```

5.72.3.2 printGold()

```

void printGold (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */

```

5.72.3.3 printGreen()

```

void printGreen (
    std::string input_str )

```

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

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

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

```

5.72.3.4 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 <code>std::cout</code> .
------------------	---

```
104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */
```

5.72.3.5 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.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```
138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154
155     #ifdef WIN32
156         std::cout << error_str << std::endl;
157     #endif
158
159     throw std::runtime_error(error_str);
160     return;
161 } /* testFloatEquals() */
```

5.72.3.6 testGreaterThan()

```
void testGreaterThan (
    double x,
```

```
double y,
std::string file,
int line )
```

Tests if $x > y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209
210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */
```

5.72.3.7 testGreaterThanOrEqualTo()

```
void testGreaterThanOrEqualTo (
    double x,
    double y,
    std::string file,
    int line )
```

Tests if $x \geq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
242 {
243     if (x >= y) {
244         return;
245     }
246
247     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
248     error_str += file;
249     error_str += "\tline ";
250     error_str += std::to_string(line);
251     error_str += ":\t\n";
```



```

252     error_str += std::to_string(x);
253     error_str += " is not greater than or equal to ";
254     error_str += std::to_string(y);
255     error_str += "\n";
256
257     #ifdef _WIN32
258         std::cout << error_str << std::endl;
259     #endif
260
261     throw std::runtime_error(error_str);
262     return;
263 } /* testGreaterThanOrEqualTo() */

```

5.72.3.8 testLessThan()

```

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

```

Tests if $x < y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

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

```

5.72.3.9 testLessThanOrEqualTo()

```

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

```

Tests if $x \leq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.72.3.10 testTruth()

```

void testTruth (
    bool statement,
    std::string file,
    int line )

```

Tests if the given statement is true.

Parameters

<i>statement</i>	The statement whose truth is to be tested ("1 == 0", for example).
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

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

```

Bibliography

- Dr. B. Buckham, Dr. C. Crawford, Dr. I. Beya Marshall, and Dr. B. Whitby. Wei Wai Kum Tidal Prefeasibility Study - Tidal Resource Assessment. Technical report, PRIMED, 2023. Internal: P2202E_BRKLYG+WEI WAI KUM_R01_V20230613v3. 225
- 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. 59
- HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 164, 212
- HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 16, 154, 164, 165, 210, 212
- HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 50, 51, 59
- HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 50, 59
- HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 51, 59
- 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. 202
- HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 164, 212
- HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 165, 210
- HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 164, 212
- W. Jakob. pybind11 — Seamless operability between C++11 and Python, 2023. URL <https://pybind11.readthedocs.io/en/stable/>. 290, 291, 293, 296, 297, 299, 300, 302, 305, 306, 308, 310, 312, 313, 315, 319
- 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. 227, 242
- Marks'. *Marks' Standard Handbook for Mechanical Engineers*. McGraw-Hill, 11 edition. ISBN: 978-0-07-142867-5. 74, 76, 77, 78
- NRCan. Auto\$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 59
- 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. 241

- A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023. Included: docs/refs/battery_degradation.pdf. [112](#), [114](#), [125](#)
- 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. [226](#), [239](#), [255](#)
- D. van Heesch. Doxygen: Generate documentation from source code, 2023. URL <https://www.doxygen.nl>. [266](#)

Index

- __applyCycleChargingControl_CHARGING
Controller, [27](#)
- __applyCycleChargingControl_DISCHARGING
Controller, [28](#)
- __applyLoadFollowingControl_CHARGING
Controller, [29](#)
- __applyLoadFollowingControl_DISCHARGING
Controller, [30](#)
- __checkBounds1D
Interpolator, [90](#)
- __checkBounds2D
Interpolator, [90](#)
- __checkDataKey1D
Interpolator, [91](#)
- __checkDataKey2D
Interpolator, [92](#)
- __checkInputs
Combustion, [14](#)
Diesel, [48](#)
Hydro, [73](#)
Lilon, [109](#)
Model, [132](#)
Noncombustion, [152](#)
Production, [162](#)
Renewable, [175](#)
Solar, [198](#)
Storage, [209](#)
Tidal, [224](#)
Wave, [238](#)
Wind, [254](#)
- __checkResourceKey1D
Resources, [183](#)
- __checkResourceKey2D
Resources, [184](#)
- __checkTimePoint
Resources, [185](#)
- __computeCubicProductionkW
Tidal, [225](#)
- __computeEconomics
Model, [133](#)
- __computeExponentialProductionkW
Tidal, [225](#)
Wind, [255](#)
- __computeFuelAndEmissions
Model, [133](#)
- __computeGaussianProductionkW
Wave, [239](#)
- __computeLevellizedCostOfEnergy
Model, [133](#)
- __computeLookupProductionkW
Tidal, [226](#)
Wave, [240](#)
Wind, [255](#)
- __computeNetLoad
Controller, [31](#)
- __computeNetPresentCost
Model, [134](#)
- __computeParaboloidProductionkW
Wave, [241](#)
- __computeRealDiscountAnnual
Storage, [210](#)
- __constructCombustionMap
Controller, [32](#)
- __flowToPower
Hydro, [74](#)
- __getAvailableFlow
Hydro, [75](#)
- __getBcal
Lilon, [112](#)
- __getDataStringMatrix
Interpolator, [92](#)
- __getEacal
Lilon, [112](#)
- __getGenericCapitalCost
Diesel, [50](#)
Hydro, [75](#)
Lilon, [113](#)
Solar, [198](#)
Tidal, [227](#)
Wave, [241](#)
Wind, [256](#)
- __getGenericFuelIntercept
Diesel, [50](#)
- __getGenericFuelSlope
Diesel, [51](#)
- __getGenericOpMaintCost
Diesel, [51](#)
Hydro, [76](#)
Lilon, [113](#)
Solar, [198](#)
Tidal, [227](#)
Wave, [242](#)
Wind, [256](#)
- __getInterpolationIndex
Interpolator, [93](#)
- __getMaximumFlowm3hr
Hydro, [76](#)
- __getMinimumFlowm3hr

- Hydro, 77
- __getRenewableProduction
 - Controller, 34
- __handleCombustionDispatch
 - Controller, 35
- __handleDegradation
 - Lilon, 113
- __handleNoncombustionDispatch
 - Controller, 36
- __handleStartStop
 - Diesel, 51
 - Noncombustion, 152
 - Renewable, 175
- __handleStorageCharging
 - Controller, 37, 38
- __handleStorageDischarging
 - Controller, 40
- __isNonNumeric
 - Interpolator, 93
- __modelDegradation
 - Lilon, 114
- __powerToFlow
 - Hydro, 77
- __readData1D
 - Interpolator, 94
- __readData2D
 - Interpolator, 95
- __readHydroResource
 - Resources, 185
- __readSolarResource
 - Resources, 186
- __readTidalResource
 - Resources, 187
- __readWaveResource
 - Resources, 188
- __readWindResource
 - Resources, 189
- __splitCommaSeparatedString
 - Interpolator, 97
- __throwLengthError
 - Resources, 190
- __throwReadError
 - Interpolator, 97
- __toggleDepleted
 - Lilon, 115
- __updateState
 - Hydro, 78
- __writeSummary
 - Combustion, 14
 - Diesel, 52
 - Hydro, 79
 - Lilon, 115
 - Model, 135
 - Noncombustion, 153
 - Renewable, 176
 - Solar, 199
 - Storage, 211
 - Tidal, 227
 - Wave, 242
 - Wind, 256
- __writeTimeSeries
 - Combustion, 14
 - Diesel, 54
 - Hydro, 80
 - Lilon, 116
 - Model, 138
 - Noncombustion, 153
 - Renewable, 176
 - Solar, 200
 - Storage, 211
 - Tidal, 228
 - Wave, 244
 - Wind, 258
- ~Combustion
 - Combustion, 13
- ~Controller
 - Controller, 27
- ~Diesel
 - Diesel, 48
- ~ElectricalLoad
 - ElectricalLoad, 64
- ~Hydro
 - Hydro, 73
- ~Interpolator
 - Interpolator, 89
- ~Lilon
 - Lilon, 109
- ~Model
 - Model, 132
- ~Noncombustion
 - Noncombustion, 152
- ~Production
 - Production, 162
- ~Renewable
 - Renewable, 175
- ~Resources
 - Resources, 182
- ~Solar
 - Solar, 197
- ~Storage
 - Storage, 209
- ~Tidal
 - Tidal, 224
- ~Wave
 - Wave, 238
- ~Wind
 - Wind, 254
- addData1D
 - Interpolator, 98
- addData2D
 - Interpolator, 98
- addDiesel
 - Model, 139
- addHydro
 - Model, 139
- addLilon

- Model, [139](#)
- addResource
 - Model, [140](#)
 - Resources, [191](#), [192](#)
- addSolar
 - Model, [142](#)
- addTidal
 - Model, [142](#)
- addWave
 - Model, [143](#)
- addWind
 - Model, [143](#)
- applyDispatchControl
 - Controller, [40](#)
- capacity_kW
 - Production, [166](#)
 - ProductionInputs, [171](#)
- capital_cost
 - DieselInputs, [59](#)
 - HydroInputs, [86](#)
 - LilongInputs, [126](#)
 - Production, [166](#)
 - SolarInputs, [204](#)
 - Storage, [214](#)
 - TidalInputs, [233](#)
 - WaveInputs, [249](#)
 - WindInputs, [264](#)
- capital_cost_vec
 - Production, [166](#)
 - Storage, [214](#)
- CH4_emissions_intensity_kgL
 - Combustion, [19](#)
 - DieselInputs, [60](#)
- CH4_emissions_vec_kg
 - Combustion, [19](#)
- CH4_kg
 - Emissions, [68](#)
- charge_kWh
 - Storage, [214](#)
- charge_vec_kWh
 - Storage, [215](#)
- charging_efficiency
 - Lilong, [121](#)
 - LilongInputs, [126](#)
- charging_power_vec_kW
 - Storage, [215](#)
- clear
 - Controller, [42](#)
 - ElectricalLoad, [64](#)
 - Model, [143](#)
 - Resources, [193](#)
- 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, [60](#)
- CO_emissions_vec_kg
 - Combustion, [20](#)
- CO_kg
 - Emissions, [68](#)
- Combustion, [9](#)
 - __checkInputs, [14](#)
 - __writeSummary, [14](#)
 - __writeTimeSeries, [14](#)
 - ~Combustion, [13](#)
 - CH4_emissions_intensity_kgL, [19](#)
 - CH4_emissions_vec_kg, [19](#)
 - 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](#)
 - fuel_consumption_vec_L, [20](#)
 - fuel_cost_L, [20](#)
 - fuel_cost_vec, [21](#)
 - fuel_mode, [21](#)
 - fuel_mode_str, [21](#)
 - getEmissionskg, [16](#)
 - getFuelConsumptionL, [17](#)
 - handleReplacement, [18](#)
 - linear_fuel_intercept_LkWh, [21](#)
 - linear_fuel_slope_LkWh, [21](#)
 - nominal_fuel_escalation_annual, [21](#)
 - NOx_emissions_intensity_kgL, [22](#)
 - NOx_emissions_vec_kg, [22](#)
 - PM_emissions_intensity_kgL, [22](#)
 - PM_emissions_vec_kg, [22](#)
 - real_fuel_escalation_annual, [22](#)
 - requestProductionkW, [18](#)
 - SOx_emissions_intensity_kgL, [22](#)
 - SOx_emissions_vec_kg, [23](#)
 - total_emissions, [23](#)
 - total_fuel_consumed_L, [23](#)
 - type, [23](#)
 - writeResults, [18](#)
- Combustion.h
 - CombustionType, [270](#)
 - DIESEL, [270](#)
 - FUEL_MODE_LINEAR, [272](#)
 - FUEL_MODE_LOOKUP, [272](#)
 - FuelMode, [270](#)
 - N_COMBUSTION_TYPES, [270](#)
 - N_FUEL_MODES, [272](#)
- combustion_inputs
 - DieselInputs, [60](#)
- combustion_map
 - Controller, [43](#)
- combustion_ptr_vec

- Model, 146
- CombustionInputs, 24
 - fuel_mode, 24
 - nominal_fuel_escalation_annual, 24
 - path_2_fuel_interp_data, 25
 - production_inputs, 25
- CombustionType
 - Combustion.h, 270
- commit
 - Combustion, 15
 - Diesel, 55
 - Hydro, 81
 - Noncombustion, 153, 154
 - Production, 163
 - Renewable, 176
 - Solar, 201
 - Tidal, 229
 - Wave, 244
 - Wind, 259
- commitCharge
 - Lilon, 117
 - Storage, 211
- commitDischarge
 - Lilon, 118
 - Storage, 211
- computeEconomics
 - Combustion, 16
 - Noncombustion, 154
 - Production, 164
 - Renewable, 177
 - Storage, 211
- computeFuelAndEmissions
 - Combustion, 16
- computeProductionkW
 - Renewable, 177, 178
 - Solar, 201
 - Tidal, 230
 - Wave, 245
 - Wind, 259
- computeRealDiscountAnnual
 - Production, 165
- control_mode
 - Controller, 43
 - ModelInputs, 149
- control_string
 - Controller, 44
- Controller, 25
 - __applyCycleChargingControl_CHARGING, 27
 - __applyCycleChargingControl_DISCHARGING, 28
 - __applyLoadFollowingControl_CHARGING, 29
 - __applyLoadFollowingControl_DISCHARGING, 30
 - __computeNetLoad, 31
 - __constructCombustionMap, 32
 - __getRenewableProduction, 34
 - __handleCombustionDispatch, 35
 - __handleNoncombustionDispatch, 36
 - __handleStorageCharging, 37, 38
 - __handleStorageDischarging, 40
 - ~Controller, 27
 - applyDispatchControl, 40
 - clear, 42
 - combustion_map, 43
 - control_mode, 43
 - control_string, 44
 - Controller, 27
 - init, 42
 - missed_load_vec_kW, 44
 - net_load_vec_kW, 44
 - setControlMode, 43
- controller
 - Model, 146
- Controller.h
 - ControlMode, 266
 - CYCLE_CHARGING, 266
 - LOAD_FOLLOWING, 266
 - N_CONTROL_MODES, 266
- ControlMode
 - Controller.h, 266
- curtailment_vec_kW
 - Production, 166
- CYCLE_CHARGING
 - Controller.h, 266
- def
 - PYBIND11_Controller.cpp, 306, 307
 - PYBIND11_Diesel.cpp, 294
 - PYBIND11_Interpolator.cpp, 310
 - PYBIND11_Lilon.cpp, 315
 - PYBIND11_Renewable.cpp, 298
 - PYBIND11_Solar.cpp, 299
- def_readwrite
 - PYBIND11_Combustion.cpp, 291, 292
 - PYBIND11_Controller.cpp, 307
 - PYBIND11_Diesel.cpp, 294, 295
 - PYBIND11_ElectricalLoad.cpp, 308, 309
 - PYBIND11_Interpolator.cpp, 310, 311
 - PYBIND11_Lilon.cpp, 315–318
 - PYBIND11_Model.cpp, 312
 - PYBIND11_Production.cpp, 296
 - PYBIND11_Resources.cpp, 313
 - PYBIND11_Solar.cpp, 299
 - PYBIND11_Storage.cpp, 320
 - PYBIND11_Tidal.cpp, 301
 - PYBIND11_Wave.cpp, 303
 - PYBIND11_Wind.cpp, 305
- degradation_a_cal
 - Lilon, 121
 - LilonInputs, 126
- degradation_alpha
 - Lilon, 121
 - LilonInputs, 126
- degradation_B_hat_cal_0
 - Lilon, 121
 - LilonInputs, 126
- degradation_beta
 - Lilon, 121
 - LilonInputs, 126

- degradation_Ea_cal_0
 - Lilon, [121](#)
 - LilonInputs, [127](#)
- degradation_r_cal
 - Lilon, [122](#)
 - LilonInputs, [127](#)
- degradation_s_cal
 - Lilon, [122](#)
 - LilonInputs, [127](#)
- derating
 - Solar, [203](#)
 - SolarInputs, [204](#)
- design_energy_period_s
 - Wave, [247](#)
 - WaveInputs, [249](#)
- design_significant_wave_height_m
 - Wave, [247](#)
 - WaveInputs, [249](#)
- design_speed_ms
 - Tidal, [232](#)
 - TidalInputs, [233](#)
 - Wind, [262](#)
 - WindInputs, [264](#)
- DIESEL
 - Combustion.h, [270](#)
- Diesel, [45](#)
 - __checkInputs, [48](#)
 - __getGenericCapitalCost, [50](#)
 - __getGenericFuelIntercept, [50](#)
 - __getGenericFuelSlope, [51](#)
 - __getGenericOpMaintCost, [51](#)
 - __handleStartStop, [51](#)
 - __writeSummary, [52](#)
 - __writeTimeSeries, [54](#)
 - ~Diesel, [48](#)
 - commit, [55](#)
 - Diesel, [47](#)
 - handleReplacement, [56](#)
 - minimum_load_ratio, [57](#)
 - minimum_runtime_hrs, [57](#)
 - requestProductionkW, [56](#)
 - time_since_last_start_hrs, [57](#)
- DieselInputs, [58](#)
 - capital_cost, [59](#)
 - CH4_emissions_intensity_kgL, [60](#)
 - CO2_emissions_intensity_kgL, [60](#)
 - CO_emissions_intensity_kgL, [60](#)
 - combustion_inputs, [60](#)
 - fuel_cost_L, [60](#)
 - linear_fuel_intercept_LkWh, [60](#)
 - linear_fuel_slope_LkWh, [61](#)
 - minimum_load_ratio, [61](#)
 - minimum_runtime_hrs, [61](#)
 - NOx_emissions_intensity_kgL, [61](#)
 - operation_maintenance_cost_kWh, [61](#)
 - PM_emissions_intensity_kgL, [62](#)
 - replace_running_hrs, [62](#)
 - SOx_emissions_intensity_kgL, [62](#)
- discharging_efficiency
 - Lilon, [122](#)
 - LilonInputs, [127](#)
- discharging_power_vec_kW
 - Storage, [215](#)
- dispatch_vec_kW
 - Production, [166](#)
- dt_vec_hrs
 - ElectricalLoad, [66](#)
- dynamic_energy_capacity_kWh
 - Lilon, [122](#)
- electrical_load
 - Model, [146](#)
- ElectricalLoad, [62](#)
 - ~ElectricalLoad, [64](#)
 - clear, [64](#)
 - dt_vec_hrs, [66](#)
 - ElectricalLoad, [63](#), [64](#)
 - load_vec_kW, [66](#)
 - max_load_kW, [66](#)
 - mean_load_kW, [66](#)
 - min_load_kW, [66](#)
 - n_points, [67](#)
 - n_years, [67](#)
 - path_2_electrical_load_time_series, [67](#)
 - readLoadData, [65](#)
 - time_vec_hrs, [67](#)
- Emissions, [67](#)
 - CH4_kg, [68](#)
 - CO2_kg, [68](#)
 - CO_kg, [68](#)
 - NOx_kg, [68](#)
 - PM_kg, [69](#)
 - SOx_kg, [69](#)
- energy_capacity_kWh
 - Storage, [215](#)
 - StorageInputs, [219](#)
- example.cpp
 - main, [287](#)
- expectedErrorNotDetected
 - testing_utils.cpp, [397](#)
 - testing_utils.h, [403](#)
- FLOAT_TOLERANCE
 - testing_utils.h, [403](#)
- fluid_density_kgm3
 - Hydro, [83](#)
 - HydroInputs, [86](#)
- fuel_consumption_vec_L
 - Combustion, [20](#)
- fuel_cost_L
 - Combustion, [20](#)
 - DieselInputs, [60](#)
- fuel_cost_vec
 - Combustion, [21](#)
- fuel_mode
 - Combustion, [21](#)
 - CombustionInputs, [24](#)

- FUEL_MODE_LINEAR
 - Combustion.h, [272](#)
- FUEL_MODE_LOOKUP
 - Combustion.h, [272](#)
- fuel_mode_str
 - Combustion, [21](#)
- FuelMode
 - Combustion.h, [270](#)
- gas_constant_JmolK
 - Lilon, [122](#)
 - LilonInputs, [127](#)
- getAcceptablekW
 - Lilon, [119](#)
 - Storage, [212](#)
- getAvailablekW
 - Lilon, [119](#)
 - Storage, [212](#)
- getEmissionskg
 - Combustion, [16](#)
- getFuelConsumptionL
 - Combustion, [17](#)
- handleReplacement
 - Combustion, [18](#)
 - Diesel, [56](#)
 - Hydro, [82](#)
 - Lilon, [120](#)
 - Noncombustion, [155](#)
 - Production, [165](#)
 - Renewable, [178](#)
 - Solar, [202](#)
 - Storage, [213](#)
 - Tidal, [231](#)
 - Wave, [246](#)
 - Wind, [261](#)
- header/Controller.h, [265](#)
- header/doxygen_cite.h, [266](#)
- header/ElectricalLoad.h, [267](#)
- header/Interpolator.h, [267](#)
- header/Model.h, [268](#)
- header/Production/Combustion/Combustion.h, [269](#)
- header/Production/Combustion/Diesel.h, [272](#)
- header/Production/Noncombustion/Hydro.h, [273](#)
- header/Production/Noncombustion/Noncombustion.h, [274](#)
- header/Production/Production.h, [275](#)
- header/Production/Renewable/Renewable.h, [276](#)
- header/Production/Renewable/Solar.h, [278](#)
- header/Production/Renewable/Tidal.h, [279](#)
- header/Production/Renewable/Wave.h, [280](#)
- header/Production/Renewable/Wind.h, [282](#)
- header/Resources.h, [283](#)
- header/std_includes.h, [284](#)
- header/Storage/Lilon.h, [285](#)
- header/Storage/Storage.h, [286](#)
- HYDRO
 - Noncombustion.h, [275](#)
- Hydro, [69](#)
 - __checkInputs, [73](#)
 - __flowToPower, [74](#)
 - __getAvailableFlow, [75](#)
 - __getGenericCapitalCost, [75](#)
 - __getGenericOpMaintCost, [76](#)
 - __getMaximumFlowm3hr, [76](#)
 - __getMinimumFlowm3hr, [77](#)
 - __powerToFlow, [77](#)
 - __updateState, [78](#)
 - __writeSummary, [79](#)
 - __writeTimeSeries, [80](#)
 - ~Hydro, [73](#)
 - commit, [81](#)
 - fluid_density_kgm3, [83](#)
 - handleReplacement, [82](#)
 - Hydro, [71](#), [72](#)
 - init_reservoir_state, [83](#)
 - maximum_flow_m3hr, [84](#)
 - minimum_flow_m3hr, [84](#)
 - net_head_m, [84](#)
 - requestProductionkW, [82](#)
 - reservoir_capacity_m3, [84](#)
 - stored_volume_m3, [84](#)
 - stored_volume_vec_m3, [84](#)
 - turbine_flow_vec_m3hr, [85](#)
 - turbine_type, [85](#)
- Hydro.h
 - HYDRO_TURBINE_FRANCIS, [274](#)
 - HYDRO_TURBINE_PELTON, [274](#)
 - HydroTurbineType, [274](#)
 - N_HYDRO_TURBINES, [274](#)
- HYDRO_TURBINE_FRANCIS
 - Hydro.h, [274](#)
- HYDRO_TURBINE_PELTON
 - Hydro.h, [274](#)
- HydroInputs, [85](#)
 - capital_cost, [86](#)
 - fluid_density_kgm3, [86](#)
 - init_reservoir_state, [86](#)
 - net_head_m, [87](#)
 - noncombustion_inputs, [87](#)
 - operation_maintenance_cost_kWh, [87](#)
 - reservoir_capacity_m3, [87](#)
 - resource_key, [87](#)
 - turbine_type, [87](#)
- HydroTurbineType
 - Hydro.h, [274](#)
- hysteresis_SOC
 - Lilon, [122](#)
 - LilonInputs, [127](#)
- init
 - Controller, [42](#)
- init_reservoir_state
 - Hydro, [83](#)
 - HydroInputs, [86](#)
- init_SOC
 - Lilon, [123](#)
 - LilonInputs, [128](#)

- interp1D
 - Interpolator, 99
- interp2D
 - Interpolator, 99
- interp_map_1D
 - Interpolator, 100
- interp_map_2D
 - Interpolator, 100
- Interpolator, 88
 - __checkBounds1D, 90
 - __checkBounds2D, 90
 - __checkDataKey1D, 91
 - __checkDataKey2D, 92
 - __getDataStringMatrix, 92
 - __getInterpolationIndex, 93
 - __isNonNumeric, 93
 - __readData1D, 94
 - __readData2D, 95
 - __splitCommaSeparatedString, 97
 - __throwReadError, 97
 - ~Interpolator, 89
 - addData1D, 98
 - addData2D, 98
 - interp1D, 99
 - interp2D, 99
 - interp_map_1D, 100
 - interp_map_2D, 100
 - Interpolator, 89
 - path_map_1D, 101
 - path_map_2D, 101
- interpolator
 - Production, 167
 - Storage, 215
- InterpolatorStruct1D, 101
 - max_x, 102
 - min_x, 102
 - n_points, 102
 - x_vec, 102
 - y_vec, 102
- InterpolatorStruct2D, 103
 - max_x, 103
 - max_y, 103
 - min_x, 104
 - min_y, 104
 - n_cols, 104
 - n_rows, 104
 - x_vec, 104
 - y_vec, 104
 - z_matrix, 105
- is_depleted
 - Storage, 215
- is_running
 - Production, 167
- is_running_vec
 - Production, 167
- is_sunk
 - Production, 167
 - ProductionInputs, 171
- Storage, 216
- StorageInputs, 219
- levellized_cost_of_energy_kWh
 - Model, 147
 - Production, 167
 - Storage, 216
- LIION
 - Storage.h, 287
- Lilon, 105
 - __checkInputs, 109
 - __getBcal, 112
 - __getEacal, 112
 - __getGenericCapitalCost, 113
 - __getGenericOpMaintCost, 113
 - __handleDegradation, 113
 - __modelDegradation, 114
 - __toggleDepleted, 115
 - __writeSummary, 115
 - __writeTimeSeries, 116
 - ~Lilon, 109
 - charging_efficiency, 121
 - commitCharge, 117
 - commitDischarge, 118
 - degradation_a_cal, 121
 - degradation_alpha, 121
 - degradation_B_hat_cal_0, 121
 - degradation_beta, 121
 - degradation_Ea_cal_0, 121
 - degradation_r_cal, 122
 - degradation_s_cal, 122
 - discharging_efficiency, 122
 - dynamic_energy_capacity_kWh, 122
 - gas_constant_JmolK, 122
 - getAcceptablekW, 119
 - getAvailablekW, 119
 - handleReplacement, 120
 - hysteresis_SOC, 122
 - init_SOC, 123
 - Lilon, 108
 - max_SOC, 123
 - min_SOC, 123
 - replace_SOH, 123
 - SOH, 123
 - SOH_vec, 123
 - temperature_K, 124
- LilonInputs, 124
 - capital_cost, 126
 - charging_efficiency, 126
 - degradation_a_cal, 126
 - degradation_alpha, 126
 - degradation_B_hat_cal_0, 126
 - degradation_beta, 126
 - degradation_Ea_cal_0, 127
 - degradation_r_cal, 127
 - degradation_s_cal, 127
 - discharging_efficiency, 127
 - gas_constant_JmolK, 127
 - hysteresis_SOC, 127

- init_SOC, 128
- max_SOC, 128
- min_SOC, 128
- operation_maintenance_cost_kWh, 128
- replace_SOH, 128
- storage_inputs, 128
- temperature_K, 129
- linear_fuel_intercept_LkWh
 - Combustion, 21
 - DieselInputs, 60
- linear_fuel_slope_LkWh
 - Combustion, 21
 - DieselInputs, 61
- LOAD_FOLLOWING
 - Controller.h, 266
- load_vec_kW
 - ElectricalLoad, 66
- main
 - example.cpp, 287
 - test_Combustion.cpp, 331
 - test_Controller.cpp, 370
 - test_Diesel.cpp, 333
 - test_ElectricalLoad.cpp, 371
 - test_Hydro.cpp, 339
 - test_Interpolator.cpp, 374
 - test_Lilon.cpp, 364
 - test_Model.cpp, 380
 - test_Noncombustion.cpp, 342
 - test_Production.cpp, 362
 - test_Renewable.cpp, 344
 - test_Resources.cpp, 389
 - test_Solar.cpp, 346
 - test_Storage.cpp, 367
 - test_Tidal.cpp, 349
 - test_Wave.cpp, 353
 - test_Wind.cpp, 358
- max_load_kW
 - ElectricalLoad, 66
- max_SOC
 - Lilon, 123
 - LilonInputs, 128
- max_x
 - InterpolatorStruct1D, 102
 - InterpolatorStruct2D, 103
- max_y
 - InterpolatorStruct2D, 103
- maximum_flow_m3hr
 - Hydro, 84
- mean_load_kW
 - ElectricalLoad, 66
- min_load_kW
 - ElectricalLoad, 66
- min_SOC
 - Lilon, 123
 - LilonInputs, 128
- min_x
 - InterpolatorStruct1D, 102
 - InterpolatorStruct2D, 104
- min_y
 - InterpolatorStruct2D, 104
- minimum_flow_m3hr
 - Hydro, 84
- minimum_load_ratio
 - Diesel, 57
 - DieselInputs, 61
- minimum_runtime_hrs
 - Diesel, 57
 - DieselInputs, 61
- missed_load_vec_kW
 - Controller, 44
- Model, 129
 - __checkInputs, 132
 - __computeEconomics, 133
 - __computeFuelAndEmissions, 133
 - __computeLevellizedCostOfEnergy, 133
 - __computeNetPresentCost, 134
 - __writeSummary, 135
 - __writeTimeSeries, 138
 - ~Model, 132
 - addDiesel, 139
 - addHydro, 139
 - addLilon, 139
 - addResource, 140
 - addSolar, 142
 - addTidal, 142
 - addWave, 143
 - addWind, 143
 - clear, 143
 - combustion_ptr_vec, 146
 - controller, 146
 - electrical_load, 146
 - levellized_cost_of_energy_kWh, 147
 - Model, 131
 - net_present_cost, 147
 - noncombustion_ptr_vec, 147
 - renewable_ptr_vec, 147
 - reset, 144
 - resources, 147
 - run, 144
 - storage_ptr_vec, 147
 - total_dispatch_discharge_kWh, 148
 - total_emissions, 148
 - total_fuel_consumed_L, 148
 - writeResults, 145
- ModelInputs, 148
 - control_mode, 149
 - path_2_electrical_load_time_series, 149
- n_cols
 - InterpolatorStruct2D, 104
- N_COMBUSTION_TYPES
 - Combustion.h, 270
- N_CONTROL_MODES
 - Controller.h, 266
- N_FUEL_MODES
 - Combustion.h, 272
- N_HYDRO_TURBINES

- Hydro.h, 274
- N_NONCOMBUSTION_TYPES
 - Noncombustion.h, 275
- n_points
 - ElectricalLoad, 67
 - InterpolatorStruct1D, 102
 - Production, 167
 - Storage, 216
- N_RENEWABLE_TYPES
 - Renewable.h, 277
- n_replacements
 - Production, 168
 - Storage, 216
- n_rows
 - InterpolatorStruct2D, 104
- n_starts
 - Production, 168
- N_STORAGE_TYPES
 - Storage.h, 287
- N_TIDAL_POWER_PRODUCTION_MODELS
 - Tidal.h, 280
- N_WAVE_POWER_PRODUCTION_MODELS
 - Wave.h, 281
- N_WIND_POWER_PRODUCTION_MODELS
 - Wind.h, 283
- n_years
 - ElectricalLoad, 67
 - Production, 168
 - Storage, 216
- net_head_m
 - Hydro, 84
 - HydroInputs, 87
- net_load_vec_kW
 - Controller, 44
- net_present_cost
 - Model, 147
 - Production, 168
 - Storage, 216
- nominal_discount_annual
 - Production, 168
 - ProductionInputs, 171
 - Storage, 217
 - StorageInputs, 219
- nominal_fuel_escalation_annual
 - Combustion, 21
 - CombustionInputs, 24
- nominal_inflation_annual
 - Production, 168
 - ProductionInputs, 171
 - Storage, 217
 - StorageInputs, 220
- Noncombustion, 149
 - __checkInputs, 152
 - __handleStartStop, 152
 - __writeSummary, 153
 - __writeTimeSeries, 153
 - ~Noncombustion, 152
 - commit, 153, 154
 - computeEconomics, 154
 - handleReplacement, 155
 - Noncombustion, 151
 - requestProductionkW, 155
 - resource_key, 156
 - type, 156
 - writeResults, 155
- Noncombustion.h
 - HYDRO, 275
 - N_NONCOMBUSTION_TYPES, 275
 - NoncombustionType, 275
- noncombustion_inputs
 - HydroInputs, 87
- noncombustion_ptr_vec
 - Model, 147
- NoncombustionInputs, 157
 - production_inputs, 157
- NoncombustionType
 - Noncombustion.h, 275
- NOx_emissions_intensity_kgL
 - Combustion, 22
 - DiesellInputs, 61
- NOx_emissions_vec_kg
 - Combustion, 22
- NOx_kg
 - Emissions, 68
- operation_maintenance_cost_kWh
 - DiesellInputs, 61
 - HydroInputs, 87
 - LilonInputs, 128
 - Production, 169
 - SolarInputs, 204
 - Storage, 217
 - TidalInputs, 233
 - WaveInputs, 249
 - WindInputs, 264
- operation_maintenance_cost_vec
 - Production, 169
 - Storage, 217
- path_2_electrical_load_time_series
 - ElectricalLoad, 67
 - ModelInputs, 149
- path_2_fuel_interp_data
 - CombustionInputs, 25
- path_2_normalized_performance_matrix
 - WaveInputs, 249
- path_map_1D
 - Interpolator, 101
 - Resources, 193
- path_map_2D
 - Interpolator, 101
 - Resources, 193
- PM_emissions_intensity_kgL
 - Combustion, 22
 - DiesellInputs, 62
- PM_emissions_vec_kg
 - Combustion, 22

- PM_kg
 - Emissions, 69
- power_capacity_kW
 - Storage, 217
 - StorageInputs, 220
- power_kW
 - Storage, 217
- power_model
 - Tidal, 232
 - TidalInputs, 234
 - Wave, 247
 - WaveInputs, 250
 - Wind, 262
 - WindInputs, 264
- power_model_string
 - Tidal, 232
 - Wave, 247
 - Wind, 262
- print_flag
 - Production, 169
 - ProductionInputs, 172
 - Storage, 218
 - StorageInputs, 220
- printGold
 - testing_utils.cpp, 397
 - testing_utils.h, 404
- printGreen
 - testing_utils.cpp, 397
 - testing_utils.h, 404
- printRed
 - testing_utils.cpp, 398
 - testing_utils.h, 404
- Production, 158
 - __checkInputs, 162
 - ~Production, 162
 - capacity_kW, 166
 - capital_cost, 166
 - capital_cost_vec, 166
 - commit, 163
 - computeEconomics, 164
 - computeRealDiscountAnnual, 165
 - curtailment_vec_kW, 166
 - dispatch_vec_kW, 166
 - handleReplacement, 165
 - interpolator, 167
 - is_running, 167
 - is_running_vec, 167
 - is_sunk, 167
 - levellized_cost_of_energy_kWh, 167
 - n_points, 167
 - n_replacements, 168
 - n_starts, 168
 - n_years, 168
 - net_present_cost, 168
 - nominal_discount_annual, 168
 - nominal_inflation_annual, 168
 - operation_maintenance_cost_kWh, 169
 - operation_maintenance_cost_vec, 169
 - print_flag, 169
 - Production, 160, 161
 - production_vec_kW, 169
 - real_discount_annual, 169
 - replace_running_hrs, 169
 - running_hours, 170
 - storage_vec_kW, 170
 - total_dispatch_kWh, 170
 - type_str, 170
- production_inputs
 - CombustionInputs, 25
 - NoncombustionInputs, 157
 - RenewableInputs, 180
- production_vec_kW
 - Production, 169
- ProductionInputs, 170
 - capacity_kW, 171
 - is_sunk, 171
 - nominal_discount_annual, 171
 - nominal_inflation_annual, 171
 - print_flag, 172
 - replace_running_hrs, 172
- projects/example.cpp, 287
- PYBIND11_Combustion.cpp
 - def_readwrite, 291, 292
 - value, 292
- PYBIND11_Controller.cpp
 - def, 306, 307
 - def_readwrite, 307
 - value, 307
- PYBIND11_Diesel.cpp
 - def, 294
 - def_readwrite, 294, 295
- PYBIND11_ElectricalLoad.cpp
 - def_readwrite, 308, 309
- PYBIND11_Interpolator.cpp
 - def, 310
 - def_readwrite, 310, 311
- PYBIND11_Lilon.cpp
 - def, 315
 - def_readwrite, 315–318
- PYBIND11_Model.cpp
 - def_readwrite, 312
- PYBIND11_MODULE
 - PYBIND11_PGM.cpp, 290
- PYBIND11_PGM.cpp
 - PYBIND11_MODULE, 290
- PYBIND11_Production.cpp
 - def_readwrite, 296
- PYBIND11_Renewable.cpp
 - def, 298
 - value, 298
- PYBIND11_Resources.cpp
 - def_readwrite, 313
- PYBIND11_Solar.cpp
 - def, 299
 - def_readwrite, 299
- PYBIND11_Storage.cpp

- def_readwrite, 320
- value, 320
- PYBIND11_Tidal.cpp
 - def_readwrite, 301
 - value, 301
- PYBIND11_Wave.cpp
 - def_readwrite, 303
 - value, 303
- PYBIND11_Wind.cpp
 - def_readwrite, 305
 - value, 305
- pybindings/PYBIND11_PGM.cpp, 289
- pybindings/snippets/Production/Combustion/PYBIND11_Controller.cpp, 291
- pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp, 292
- pybindings/snippets/Production/PYBIND11_Production.cpp, 295
- pybindings/snippets/Production/Renewable/PYBIND11_Replacement.cpp, 297
- pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp, 298
- pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp, 300
- pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp, 302
- pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp, 304
- pybindings/snippets/PYBIND11_Controller.cpp, 306
- pybindings/snippets/PYBIND11_ElectricalLoad.cpp, 308
- pybindings/snippets/PYBIND11_Interpolator.cpp, 309
- pybindings/snippets/PYBIND11_Model.cpp, 312
- pybindings/snippets/PYBIND11_Resources.cpp, 313
- pybindings/snippets/Storage/PYBIND11_Lilon.cpp, 314
- pybindings/snippets/Storage/PYBIND11_Storage.cpp, 319
- readLoadData
 - ElectricalLoad, 65
- real_discount_annual
 - Production, 169
 - Storage, 218
- real_fuel_escalation_annual
 - Combustion, 22
- Renewable, 172
 - __checkInputs, 175
 - __handleStartStop, 175
 - __writeSummary, 176
 - __writeTimeSeries, 176
 - ~Renewable, 175
 - commit, 176
 - computeEconomics, 177
 - computeProductionkW, 177, 178
 - handleReplacement, 178
 - Renewable, 174
 - resource_key, 179
 - type, 179
 - writeResults, 178
- Renewable.h
 - N_RENEWABLE_TYPES, 277
 - RenewableType, 277
 - SOLAR, 277
 - TIDAL, 277
 - WAVE, 277
 - WIND, 277
- renewable_inputs
 - SolarInputs, 205
 - TidalInputs, 234
 - WaveInputs, 250
 - WindInputs, 264
- renewable_model_vec
 - Model, 147
- renewable_inputs
 - production_inputs, 180
- RenewableType
 - Renewable.h, 277
- replacement_hours
 - DieselInputs, 62
 - Production, 169
 - ProductionInputs, 172
- replace_SOH
 - Lilon, 123
- requestProductionkW
 - Combustion, 18
 - Diesel, 56
 - Hydro, 82
 - Noncombustion, 155
- reservoir_capacity_m3
 - Hydro, 84
 - HydroInputs, 87
- reset
 - Model, 144
- resource_key
 - HydroInputs, 87
 - Noncombustion, 156
 - Renewable, 179
 - SolarInputs, 205
 - TidalInputs, 234
 - WaveInputs, 250
 - WindInputs, 264
- resource_map_1D
 - Resources, 193
- resource_map_2D
 - Resources, 194
- Resources, 181
 - __checkResourceKey1D, 183
 - __checkResourceKey2D, 184
 - __checkTimePoint, 185
 - __readHydroResource, 185
 - __readSolarResource, 186
 - __readTidalResource, 187
 - __readWaveResource, 188
 - __readWindResource, 189
 - __throwLengthError, 190
 - ~Resources, 182

- addResource, [191](#), [192](#)
- clear, [193](#)
- path_map_1D, [193](#)
- path_map_2D, [193](#)
- resource_map_1D, [193](#)
- resource_map_2D, [194](#)
- Resources, [182](#)
- string_map_1D, [194](#)
- string_map_2D, [194](#)
- resources
 - Model, [147](#)
- run
 - Model, [144](#)
- running_hours
 - Production, [170](#)
- setControlMode
 - Controller, [43](#)
- SOH
 - Lilon, [123](#)
- SOH_vec
 - Lilon, [123](#)
- SOLAR
 - Renewable.h, [277](#)
- Solar, [195](#)
 - __checkInputs, [198](#)
 - __getGenericCapitalCost, [198](#)
 - __getGenericOpMaintCost, [198](#)
 - __writeSummary, [199](#)
 - __writeTimeSeries, [200](#)
 - ~Solar, [197](#)
 - commit, [201](#)
 - computeProductionkW, [201](#)
 - derating, [203](#)
 - handleReplacement, [202](#)
 - Solar, [196](#), [197](#)
- SolarInputs, [203](#)
 - capital_cost, [204](#)
 - derating, [204](#)
 - operation_maintenance_cost_kWh, [204](#)
 - renewable_inputs, [205](#)
 - resource_key, [205](#)
- source/Controller.cpp, [321](#)
- source/ElectricalLoad.cpp, [321](#)
- source/Interpolator.cpp, [322](#)
- source/Model.cpp, [322](#)
- source/Production/Combustion/Combustion.cpp, [323](#)
- source/Production/Combustion/Diesel.cpp, [323](#)
- source/Production/Noncombustion/Hydro.cpp, [324](#)
- source/Production/Noncombustion/Noncombustion.cpp, [324](#)
- source/Production/Production.cpp, [325](#)
- source/Production/Renewable/Renewable.cpp, [326](#)
- source/Production/Renewable/Solar.cpp, [326](#)
- source/Production/Renewable/Tidal.cpp, [327](#)
- source/Production/Renewable/Wave.cpp, [327](#)
- source/Production/Renewable/Wind.cpp, [328](#)
- source/Resources.cpp, [329](#)
- source/Storage/Lilon.cpp, [329](#)
- source/Storage/Storage.cpp, [330](#)
- SOx_emissions_intensity_kgL
 - Combustion, [22](#)
 - DieselInputs, [62](#)
- SOx_emissions_vec_kg
 - Combustion, [23](#)
- SOx_kg
 - Emissions, [69](#)
- Storage, [205](#)
 - __checkInputs, [209](#)
 - __computeRealDiscountAnnual, [210](#)
 - __writeSummary, [211](#)
 - __writeTimeSeries, [211](#)
 - ~Storage, [209](#)
 - capital_cost, [214](#)
 - capital_cost_vec, [214](#)
 - charge_kWh, [214](#)
 - charge_vec_kWh, [215](#)
 - charging_power_vec_kW, [215](#)
 - commitCharge, [211](#)
 - commitDischarge, [211](#)
 - computeEconomics, [211](#)
 - discharging_power_vec_kW, [215](#)
 - energy_capacity_kWh, [215](#)
 - getAcceptablekW, [212](#)
 - getAvailablekW, [212](#)
 - handleReplacement, [213](#)
 - interpolator, [215](#)
 - is_depleted, [215](#)
 - is_sunk, [216](#)
 - levellized_cost_of_energy_kWh, [216](#)
 - n_points, [216](#)
 - n_replacements, [216](#)
 - n_years, [216](#)
 - net_present_cost, [216](#)
 - nominal_discount_annual, [217](#)
 - nominal_inflation_annual, [217](#)
 - operation_maintenance_cost_kWh, [217](#)
 - operation_maintenance_cost_vec, [217](#)
 - power_capacity_kW, [217](#)
 - power_kW, [217](#)
 - print_flag, [218](#)
 - real_discount_annual, [218](#)
 - Storage, [208](#)
 - total_discharge_kWh, [218](#)
 - type, [218](#)
 - type_str, [218](#)
 - writeResults, [213](#)
- Storage.h
 - LIION, [287](#)
 - N_STORAGE_TYPES, [287](#)
 - StorageType, [287](#)
- storage_inputs
 - LilonInputs, [128](#)
- storage_ptr_vec
 - Model, [147](#)
- storage_vec_kW
 - Production, [170](#)

- StorageInputs, 219
 - energy_capacity_kWh, 219
 - is_sunk, 219
 - nominal_discount_annual, 219
 - nominal_inflation_annual, 220
 - power_capacity_kW, 220
 - print_flag, 220
- StorageType
 - Storage.h, 287
- stored_volume_m3
 - Hydro, 84
- stored_volume_vec_m3
 - Hydro, 84
- string_map_1D
 - Resources, 194
- string_map_2D
 - Resources, 194
- temperature_K
 - Lilon, 124
 - LilonInputs, 129
- test/source/Production/Combustion/test_Combustion.cpp, 330
- test/source/Production/Combustion/test_Diesel.cpp, 332
- test/source/Production/Noncombustion/test_Hydro.cpp, 339
- test/source/Production/Noncombustion/test_Noncombustion.cpp, 342
- test/source/Production/Renewable/test_Renewable.cpp, 343
- test/source/Production/Renewable/test_Solar.cpp, 345
- test/source/Production/Renewable/test_Tidal.cpp, 349
- test/source/Production/Renewable/test_Wave.cpp, 352
- test/source/Production/Renewable/test_Wind.cpp, 357
- test/source/Production/test_Production.cpp, 361
- test/source/Storage/test_Lilon.cpp, 364
- test/source/Storage/test_Storage.cpp, 367
- test/source/test_Controller.cpp, 369
- test/source/test_ElectricalLoad.cpp, 371
- test/source/test_Interpolator.cpp, 373
- test/source/test_Model.cpp, 379
- test/source/test_Resources.cpp, 388
- test/utills/testing_utils.cpp, 396
- test/utills/testing_utils.h, 402
- test_Combustion.cpp
 - main, 331
- test_Controller.cpp
 - main, 370
- test_Diesel.cpp
 - main, 333
- test_ElectricalLoad.cpp
 - main, 371
- test_Hydro.cpp
 - main, 339
- test_Interpolator.cpp
 - main, 374
- test_Lilon.cpp
 - main, 364
- test_Model.cpp
 - main, 380
- test_Noncombustion.cpp
 - main, 342
- test_Production.cpp
 - main, 362
- test_Renewable.cpp
 - main, 344
- test_Resources.cpp
 - main, 389
- test_Solar.cpp
 - main, 346
- test_Storage.cpp
 - main, 367
- test_Tidal.cpp
 - main, 349
- test_Wave.cpp
 - main, 353
- test_Wind.cpp
 - main, 358
- testFloatEquals
 - testing_utils.cpp, 398
 - testing_utils.h, 405
- testGreaterThan
 - testing_utils.cpp, 399
 - testing_utils.h, 405
- testGreaterThanOrEqualTo
 - testing_utils.cpp, 399
 - testing_utils.h, 406
- testing_utils.cpp
 - expectedErrorNotDetected, 397
 - printGold, 397
 - printGreen, 397
 - printRed, 398
 - testFloatEquals, 398
 - testGreaterThan, 399
 - testGreaterThanOrEqualTo, 399
 - testLessThan, 400
 - testLessThanOrEqualTo, 401
 - testTruth, 401
- testing_utils.h
 - expectedErrorNotDetected, 403
 - FLOAT_TOLERANCE, 403
 - printGold, 404
 - printGreen, 404
 - printRed, 404
 - testFloatEquals, 405
 - testGreaterThan, 405
 - testGreaterThanOrEqualTo, 406
 - testLessThan, 407
 - testLessThanOrEqualTo, 407
 - testTruth, 408
- testLessThan
 - testing_utils.cpp, 400
 - testing_utils.h, 407
- testLessThanOrEqualTo
 - testing_utils.cpp, 401
 - testing_utils.h, 407

- testTruth
 - testing_utils.cpp, 401
 - testing_utils.h, 408
- TIDAL
 - Renewable.h, 277
- Tidal, 221
 - __checkInputs, 224
 - __computeCubicProductionkW, 225
 - __computeExponentialProductionkW, 225
 - __computeLookupProductionkW, 226
 - __getGenericCapitalCost, 227
 - __getGenericOpMaintCost, 227
 - __writeSummary, 227
 - __writeTimeSeries, 228
 - ~Tidal, 224
 - commit, 229
 - computeProductionkW, 230
 - design_speed_ms, 232
 - handleReplacement, 231
 - power_model, 232
 - power_model_string, 232
 - Tidal, 223
- Tidal.h
 - N_TIDAL_POWER_PRODUCTION_MODELS, 280
 - TIDAL_POWER_CUBIC, 280
 - TIDAL_POWER_EXPONENTIAL, 280
 - TIDAL_POWER_LOOKUP, 280
 - TidalPowerProductionModel, 280
- TIDAL_POWER_CUBIC
 - Tidal.h, 280
- TIDAL_POWER_EXPONENTIAL
 - Tidal.h, 280
- TIDAL_POWER_LOOKUP
 - Tidal.h, 280
- TidalInputs, 232
 - capital_cost, 233
 - design_speed_ms, 233
 - operation_maintenance_cost_kWh, 233
 - power_model, 234
 - renewable_inputs, 234
 - resource_key, 234
- TidalPowerProductionModel
 - Tidal.h, 280
- time_since_last_start_hrs
 - Diesel, 57
- time_vec_hrs
 - ElectricalLoad, 67
- total_discharge_kWh
 - Storage, 218
- total_dispatch_discharge_kWh
 - Model, 148
- total_dispatch_kWh
 - Production, 170
- total_emissions
 - Combustion, 23
 - Model, 148
- total_fuel_consumed_L
 - Combustion, 23
 - Model, 148
- turbine_flow_vec_m3hr
 - Hydro, 85
- turbine_type
 - Hydro, 85
 - HydroInputs, 87
- type
 - Combustion, 23
 - Noncombustion, 156
 - Renewable, 179
 - Storage, 218
- type_str
 - Production, 170
 - Storage, 218
- value
 - PYBIND11_Combustion.cpp, 292
 - PYBIND11_Controller.cpp, 307
 - PYBIND11_Renewable.cpp, 298
 - PYBIND11_Storage.cpp, 320
 - PYBIND11_Tidal.cpp, 301
 - PYBIND11_Wave.cpp, 303
 - PYBIND11_Wind.cpp, 305
- WAVE
 - Renewable.h, 277
- Wave, 235
 - __checkInputs, 238
 - __computeGaussianProductionkW, 239
 - __computeLookupProductionkW, 240
 - __computeParaboloidProductionkW, 241
 - __getGenericCapitalCost, 241
 - __getGenericOpMaintCost, 242
 - __writeSummary, 242
 - __writeTimeSeries, 244
 - ~Wave, 238
 - commit, 244
 - computeProductionkW, 245
 - design_energy_period_s, 247
 - design_significant_wave_height_m, 247
 - handleReplacement, 246
 - power_model, 247
 - power_model_string, 247
 - Wave, 237
- Wave.h
 - N_WAVE_POWER_PRODUCTION_MODELS, 281
 - WAVE_POWER_GAUSSIAN, 281
 - WAVE_POWER_LOOKUP, 281
 - WAVE_POWER_PARABOLOID, 281
 - WavePowerProductionModel, 281
- WAVE_POWER_GAUSSIAN
 - Wave.h, 281
- WAVE_POWER_LOOKUP
 - Wave.h, 281
- WAVE_POWER_PARABOLOID
 - Wave.h, 281
- WaveInputs, 248

- capital_cost, [249](#)
- design_energy_period_s, [249](#)
- design_significant_wave_height_m, [249](#)
- operation_maintenance_cost_kWh, [249](#)
- path_2_normalized_performance_matrix, [249](#)
- power_model, [250](#)
- renewable_inputs, [250](#)
- resource_key, [250](#)
- WavePowerProductionModel
 - Wave.h, [281](#)
- WIND
 - Renewable.h, [277](#)
- Wind, [251](#)
 - __checkInputs, [254](#)
 - __computeExponentialProductionkW, [255](#)
 - __computeLookupProductionkW, [255](#)
 - __getGenericCapitalCost, [256](#)
 - __getGenericOpMaintCost, [256](#)
 - __writeSummary, [256](#)
 - __writeTimeSeries, [258](#)
 - ~Wind, [254](#)
 - commit, [259](#)
 - computeProductionkW, [259](#)
 - design_speed_ms, [262](#)
 - handleReplacement, [261](#)
 - power_model, [262](#)
 - power_model_string, [262](#)
 - Wind, [253](#)
- Wind.h
 - N_WIND_POWER_PRODUCTION_MODELS, [283](#)
 - WIND_POWER_EXPONENTIAL, [283](#)
 - WIND_POWER_LOOKUP, [283](#)
 - WindPowerProductionModel, [283](#)
- WIND_POWER_EXPONENTIAL
 - Wind.h, [283](#)
- WIND_POWER_LOOKUP
 - Wind.h, [283](#)
- WindInputs, [263](#)
 - capital_cost, [264](#)
 - design_speed_ms, [264](#)
 - operation_maintenance_cost_kWh, [264](#)
 - power_model, [264](#)
 - renewable_inputs, [264](#)
 - resource_key, [264](#)
- WindPowerProductionModel
 - Wind.h, [283](#)
- writeResults
 - Combustion, [18](#)
 - Model, [145](#)
 - Noncombustion, [155](#)
 - Renewable, [178](#)
 - Storage, [213](#)
- x_vec
 - InterpolatorStruct1D, [102](#)
 - InterpolatorStruct2D, [104](#)
- y_vec
 - InterpolatorStruct1D, [102](#)
- InterpolatorStruct2D, [104](#)
- z_matrix
 - InterpolatorStruct2D, [105](#)