PGMcpp: PRIMED Grid Modelling (in C++)

Generated by Doxygen 1.9.1

1 Hierarchical Index 1

1.1 Class Hierarchy 1

2 Class Index 3

2.1 Class List 3

3 File Index 5

3.1 File List 5

4 Class Documentation 9

4.1 Combustion Class Reference 9

4.1.1 Detailed Description 12

4.1.2 Constructor & Destructor Documentation 12

4.1.2.1 Combustion() [1/2] 12

4.1.2.2 Combustion() [2/2] 12

4.1.2.3 ∼Combustion() 14

4.1.3 Member Function Documentation 14

4.1.3.1 __checkInputs() 14

4.1.3.2 __writeSummary() 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] . . . 39
4.3.3.12 __handleStorageDischarging() . . . 40
4.3.3.13 applyDispatchControl() . . . 41
4.3.3.14 clear() . . . 42
4.3.3.15 init() . . . 43
4.3.3.16 setControlMode() . . . 43
4.3.4 Member Data Documentation . . . 44
4.3.4.1 combustion_map . . . 44
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() 51
4.4.3.4 __getGenericFuelSlope() 51
4.4.3.5 __getGenericOpMaintCost() 52
4.4.3.6 __handleStartStop() 52
4.4.3.7 __writeSummary() 53
4.4.3.8 __writeTimeSeries() 55
4.4.3.9 commit() 56
4.4.3.10 handleReplacement() 57
4.4.3.11 requestProductionkW() 57
4.4.4 Member Data Documentation 58
4.4.4.1 minimum_load_ratio 58
4.4.4.2 minimum_runtime_hrs 58
4.4.4.3 time_since_last_start_hrs 58
4.5 DieselInputs Struct Reference 59
4.5.1 Detailed Description 60
4.5.2 Member Data Documentation 60
4.5.2.1 capital_cost 60
4.5.2.2 CH4_emissions_intensity_kgL 60
4.5.2.3 CO2_emissions_intensity_kgL 61
4.5.2.4 CO_emissions_intensity_kgL 61
4.5.2.5 combustion_inputs 61
4.5.2.6 fuel_cost_L 61
4.5.2.7 linear_fuel_intercept_LkWh 61
4.5.2.8 linear_fuel_slope_LkWh 61
4.5.2.9 minimum_load_ratio 62
4.5.2.10 minimum_runtime_hrs 62
4.5.2.11 NOx_emissions_intensity_kgL 62
4.5.2.12 operation_maintenance_cost_kWh 62
4.5.2.13 PM_emissions_intensity_kgL 62
4.5.2.14 replace_running_hrs 62

4.5.2.15 SOx_emissions_intensity_kgL . . . 63
4.6 ElectricalLoad Class Reference . . . 63
4.6.1 Detailed Description . . . 64
4.6.2 Constructor & Destructor Documentation . . . 64
4.6.2.1 ElectricalLoad() [1/2] . . . 64
4.6.2.2 ElectricalLoad() [2/2] . . . 64
4.6.2.3 ∼ElectricalLoad() . . . 64
4.6.3 Member Function Documentation . . . 64
4.6.3.1 clear() . . . 65
4.6.3.2 readLoadData() . . . 65
4.6.4 Member Data Documentation . . . 66
4.6.4.1 dt_vec_hrs . . . 66
4.6.4.2 load_vec_kW . . . 66
4.6.4.3 max_load_kW . . . 66
4.6.4.4 mean_load_kW . . . 67
4.6.4.5 min_load_kW . . . 67
4.6.4.6 n_points . . . 67
4.6.4.7 n_years . . . 67
4.6.4.8 path_2_electrical_load_time_series . . . 67
4.6.4.9 time_vec_hrs . . . 67
4.7 Emissions Struct Reference . . . 68
4.7.1 Detailed Description . . . 68
4.7.2 Member Data Documentation . . . 68
4.7.2.1 CH4_kg . . . 68
4.7.2.2 CO2_kg . . . 68
4.7.2.3 CO_kg . . . 69
4.7.2.4 NOx_kg . . . 69
4.7.2.5 PM_kg . . . 69
4.7.2.6 SOx_kg . . . 69
4.8 Hydro Class Reference . . . 70
4.8.1 Detailed Description . . . 72
4.8.2 Constructor & Destructor Documentation . . . 72
4.8.2.1 Hydro() [1/2] . . . 72
4.8.2.2 Hydro() [2/2] . . . 73
4.8.2.3 ∼Hydro() . . . 74
4.8.3 Member Function Documentation . . . 74
4.8.3.1 __checkInputs() . . . 74
4.8.3.2 __flowToPower() . . . 75
4.8.3.3 __getAcceptableFlow() . . . 75
4.8.3.4 __getAvailableFlow() . . . 76
4.8.3.5 __getEfficiencyFactor() . . . 76
4.8.3.6 __getGenericCapitalCost() . . . 77

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

4.10.3.1 __checkBounds1D() . . . 93
4.10.3.2 __checkBounds2D() . . . 94
4.10.3.3 __checkDataKey1D() . . . 95
4.10.3.4 __checkDataKey2D() . . . 96
4.10.3.5 __getDataStringMatrix() . . . 96
4.10.3.6 __getInterpolationIndex() . . . 97
4.10.3.7 __isNonNumeric() . . . 97
4.10.3.8 __readData1D() . . . 98
4.10.3.9 __readData2D() . . . 99
4.10.3.10 __splitCommaSeparatedString() . . . 100
4.10.3.11 __throwReadError() . . . 101
4.10.3.12 addData1D() . . . 101
4.10.3.13 addData2D() . . . 102
4.10.3.14 interp1D() . . . 102
4.10.3.15 interp2D() . . . 103
4.10.4 Member Data Documentation . . . 104
4.10.4.1 interp_map_1D . . . 104
4.10.4.2 interp_map_2D . . . 104
4.10.4.3 path_map_1D . . . 104
4.10.4.4 path_map_2D . . . 104
4.11 InterpolatorStruct1D Struct Reference . . . 105
4.11.1 Detailed Description . . . 105
4.11.2 Member Data Documentation . . . 105
4.11.2.1 max_x . . . 105
4.11.2.2 min_x . . . 105
4.11.2.3 n_points . . . 106
4.11.2.4 x_vec . . . 106
4.11.2.5 y_vec . . . 106
4.12 InterpolatorStruct2D Struct Reference . . . 106
4.12.1 Detailed Description . . . 107
4.12.2 Member Data Documentation . . . 107
4.12.2.1 max_x . . . 107
4.12.2.2 max_y . . . 107
4.12.2.3 min_x . . . 107
4.12.2.4 min_y . . . 107
4.12.2.5 n_cols . . . 107
4.12.2.6 n_rows . . . 108
4.12.2.7 x_vec . . . 108
4.12.2.8 y_vec . . . 108
4.12.2.9 z_matrix . . . 108
4.13 LiIon Class Reference . . . 109
4.13.1 Detailed Description . . . 111

4.13.2 Constructor & Destructor Documentation 111
4.13.2.1 LiIon() [1/2] 111
4.13.2.2 LiIon() [2/2] 111
4.13.2.3 ∼LiIon() 112
4.13.3 Member Function Documentation 113
4.13.3.1 __checkInputs() 113
4.13.3.2 __getBcal() 115
4.13.3.3 __getEacal() 116
4.13.3.4 __getGenericCapitalCost() 116
4.13.3.5 __getGenericOpMaintCost() 117
4.13.3.6 __handleDegradation() 117
4.13.3.7 __modelDegradation() 117
4.13.3.8 __toggleDepleted() 119
4.13.3.9 __writeSummary() 119
4.13.3.10 __writeTimeSeries() 121
4.13.3.11 commitCharge() 121
4.13.3.12 commitDischarge() 122
4.13.3.13 getAcceptablekW() 123
4.13.3.14 getAvailablekW() 124
4.13.3.15 handleReplacement() 124
4.13.4 Member Data Documentation 125
4.13.4.1 charging_efficiency 125
4.13.4.2 degradation_a_cal 125
4.13.4.3 degradation_alpha 125
4.13.4.4 degradation_B_hat_cal_0 125
4.13.4.5 degradation_beta 126
4.13.4.6 degradation_Ea_cal_0 126
4.13.4.7 degradation_r_cal 126
4.13.4.8 degradation_s_cal 126
4.13.4.9 discharging_efficiency 126
4.13.4.10 dynamic_energy_capacity_kWh 126
4.13.4.11 gas_constant_JmolK 127
4.13.4.12 hysteresis_SOC 127
4.13.4.13 init_SOC 127
4.13.4.14 max_SOC 127
4.13.4.15 min_SOC 127
4.13.4.16 replace_SOH 127
4.13.4.17 SOH 128
4.13.4.18 SOH_vec 128
4.13.4.19 temperature_K 128
4.14 LiIonInputs Struct Reference 128
4.14.1 Detailed Description 129

4.14.2 Member Data Documentation 130
4.14.2.1 capital_cost 130
4.14.2.2 charging_efficiency 130
4.14.2.3 degradation_a_cal 130
4.14.2.4 degradation_alpha 130
4.14.2.5 degradation_B_hat_cal_0 130
4.14.2.6 degradation_beta 131
4.14.2.7 degradation_Ea_cal_0 131
4.14.2.8 degradation_r_cal 131
4.14.2.9 degradation_s_cal 131
4.14.2.10 discharging_efficiency 131
4.14.2.11 gas_constant_JmolK 131
4.14.2.12 hysteresis_SOC 132
4.14.2.13 init_SOC 132
4.14.2.14 max_SOC 132
4.14.2.15 min_SOC 132
4.14.2.16 operation_maintenance_cost_kWh 132
4.14.2.17 replace_SOH 132
4.14.2.18 storage_inputs 133
4.14.2.19 temperature_K 133
4.15 Model Class Reference 133
4.15.1 Detailed Description 135
4.15.2 Constructor & Destructor Documentation 135
4.15.2.1 Model() [1/2] 136
4.15.2.2 Model() [2/2] 136
4.15.2.3 ∼Model() 136
4.15.3 Member Function Documentation 136
4.15.3.1 __checkInputs() 137
4.15.3.2 __computeEconomics() 137
4.15.3.3 __computeFuelAndEmissions() 137
4.15.3.4 __computeLevellizedCostOfEnergy() 138
4.15.3.5 __computeNetPresentCost() 138
4.15.3.6 __writeSummary() 139
4.15.3.7 __writeTimeSeries() 142
4.15.3.8 addDiesel() 143
4.15.3.9 addHydro() 143
4.15.3.10 addLiIon() 144
4.15.3.11 addResource() [1/2] 144
4.15.3.12 addResource() [2/2] 145
4.15.3.13 addSolar() 145
4.15.3.14 addTidal() 146
4.15.3.15 addWave() 146

4.15.3.16 addWind() . . . 146
4.15.3.17 clear() . . . 147
4.15.3.18 reset() . . . 147
4.15.3.19 run() . . . 148
4.15.3.20 writeResults() . . . 148
4.15.4 Member Data Documentation . . . 150
4.15.4.1 combustion_ptr_vec . . . 150
4.15.4.2 controller . . . 150
4.15.4.3 electrical_load . . . 150
4.15.4.4 levellized_cost_of_energy_kWh . . . 150
4.15.4.5 net_present_cost . . . 150
4.15.4.6 noncombustion_ptr_vec . . . 151
4.15.4.7 renewable_ptr_vec . . . 151
4.15.4.8 resources . . . 151
4.15.4.9 storage_ptr_vec . . . 151
4.15.4.10 total_dispatch_discharge_kWh . . . 151
4.15.4.11 total_emissions . . . 151
4.15.4.12 total_fuel_consumed_L . . . 152
4.15.4.13 total_renewable_dispatch_kWh . . . 152
4.16 ModelInputs Struct Reference . . . 152
4.16.1 Detailed Description . . . 152
4.16.2 Member Data Documentation . . . 152
4.16.2.1 control_mode . . . 153
4.16.2.2 path_2_electrical_load_time_series . . . 153
4.17 Noncombustion Class Reference . . . 153
4.17.1 Detailed Description . . . 155
4.17.2 Constructor & Destructor Documentation . . . 155
4.17.2.1 Noncombustion() [1/2] . . . 155
4.17.2.2 Noncombustion() [2/2] . . . 155
4.17.2.3 ∼Noncombustion() . . . 156
4.17.3 Member Function Documentation . . . 156
4.17.3.1 __checkInputs() . . . 156
4.17.3.2 __handleStartStop() . . . 156
4.17.3.3 __writeSummary() . . . 157
4.17.3.4 __writeTimeSeries() . . . 157
4.17.3.5 commit() [1/2] . . . 157
4.17.3.6 commit() [2/2] . . . 158
4.17.3.7 computeEconomics() . . . 158
4.17.3.8 handleReplacement() . . . 159
4.17.3.9 requestProductionkW() [1/2] . . . 159
4.17.3.10 requestProductionkW() [2/2] . . . 159
4.17.3.11 writeResults() . . . 160

4.17.4 Member Data Documentation 160
4.17.4.1 resource_key 160
4.17.4.2 type 161
4.18 NoncombustionInputs Struct Reference 161
4.18.1 Detailed Description 161
4.18.2 Member Data Documentation 161
4.18.2.1 production_inputs 162
4.19 Production Class Reference 162
4.19.1 Detailed Description 165
4.19.2 Constructor & Destructor Documentation 165
4.19.2.1 Production() [1/2] 165
4.19.2.2 Production() [2/2] 165
4.19.2.3 ∼Production() 166
4.19.3 Member Function Documentation 166
4.19.3.1 __checkInputs() 167
4.19.3.2 __checkNormalizedProduction() 167
4.19.3.3 __checkTimePoint() 168
4.19.3.4 __readNormalizedProductionData() 168
4.19.3.5 __throwLengthError() 169
4.19.3.6 commit() 170
4.19.3.7 computeEconomics() 171
4.19.3.8 computeRealDiscountAnnual() 171
4.19.3.9 getProductionkW() 172
4.19.3.10 handleReplacement() 172
4.19.4 Member Data Documentation 173
4.19.4.1 capacity_kW 173
4.19.4.2 capital_cost 173
4.19.4.3 capital_cost_vec 173
4.19.4.4 curtailment_vec_kW 173
4.19.4.5 dispatch_vec_kW 174
4.19.4.6 interpolator 174
4.19.4.7 is_running 174
4.19.4.8 is_running_vec 174
4.19.4.9 is_sunk 174
4.19.4.10 levellized_cost_of_energy_kWh 174
4.19.4.11 n_points 175
4.19.4.12 n_replacements 175
4.19.4.13 n_starts 175
4.19.4.14 n_years 175
4.19.4.15 net_present_cost 175
4.19.4.16 nominal_discount_annual 175
4.19.4.17 nominal_inflation_annual 176

4.19.4.18 normalized_production_series_given 176
4.19.4.19 normalized_production_vec 176
4.19.4.20 operation_maintenance_cost_kWh 176
4.19.4.21 operation_maintenance_cost_vec 176
4.19.4.22 path_2_normalized_production_time_series 176
4.19.4.23 print_flag 177
4.19.4.24 production_vec_kW 177
4.19.4.25 real_discount_annual 177
4.19.4.26 replace_running_hrs 177
4.19.4.27 running_hours 177
4.19.4.28 storage_vec_kW 177
4.19.4.29 total_dispatch_kWh 178
4.19.4.30 type_str 178
4.20 ProductionInputs Struct Reference 178
4.20.1 Detailed Description 178
4.20.2 Member Data Documentation 179
4.20.2.1 capacity_kW 179
4.20.2.2 is_sunk 179
4.20.2.3 nominal_discount_annual 179
4.20.2.4 nominal_inflation_annual 179
4.20.2.5 path_2_normalized_production_time_series 179
4.20.2.6 print_flag 180
4.20.2.7 replace_running_hrs 180
4.21 Renewable Class Reference 180
4.21.1 Detailed Description 182
4.21.2 Constructor & Destructor Documentation 182
4.21.2.1 Renewable() [1/2] 182
4.21.2.2 Renewable() [2/2] 182
4.21.2.3 ∼Renewable() 183
4.21.3 Member Function Documentation 183
4.21.3.1 __checkInputs() 183
4.21.3.2 __handleStartStop() 184
4.21.3.3 __writeSummary() 184
4.21.3.4 __writeTimeSeries() 184
4.21.3.5 commit() 184
4.21.3.6 computeEconomics() 185
4.21.3.7 computeProductionkW() [1/2] 185
4.21.3.8 computeProductionkW() [2/2] 186
4.21.3.9 handleReplacement() 186
4.21.3.10 writeResults() 186
4.21.4 Member Data Documentation 187
4.21.4.1 resource_key 187

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

4.24.3.5 __writeTimeSeries() 209
4.24.3.6 commit() 209
4.24.3.7 computeProductionkW() 210
4.24.3.8 handleReplacement() 211
4.24.4 Member Data Documentation 211
4.24.4.1 derating 211
4.25 SolarInputs Struct Reference 212
4.25.1 Detailed Description 212
4.25.2 Member Data Documentation 213
4.25.2.1 capital_cost 213
4.25.2.2 derating 213
4.25.2.3 operation_maintenance_cost_kWh 213
4.25.2.4 renewable_inputs 213
4.25.2.5 resource_key 213
4.26 Storage Class Reference 214
4.26.1 Detailed Description 216
4.26.2 Constructor & Destructor Documentation 216
4.26.2.1 Storage() [1/2] 216
4.26.2.2 Storage() [2/2] 216
4.26.2.3 ∼Storage() 217
4.26.3 Member Function Documentation 217
4.26.3.1 __checkInputs() 218
4.26.3.2 __computeRealDiscountAnnual() 218
4.26.3.3 __writeSummary() 219
4.26.3.4 __writeTimeSeries() 219
4.26.3.5 commitCharge() 219
4.26.3.6 commitDischarge() 220
4.26.3.7 computeEconomics() 220
4.26.3.8 getAcceptablekW() 221
4.26.3.9 getAvailablekW() 221
4.26.3.10 handleReplacement() 221
4.26.3.11 writeResults() 222
4.26.4 Member Data Documentation 222
4.26.4.1 capital_cost 222
4.26.4.2 capital_cost_vec 223
4.26.4.3 charge_kWh 223
4.26.4.4 charge_vec_kWh 223
4.26.4.5 charging_power_vec_kW 223
4.26.4.6 discharging_power_vec_kW 223
4.26.4.7 energy_capacity_kWh 224
4.26.4.8 interpolator 224
4.26.4.9 is_depleted 224

4.26.4.10 is_sunk . . . 224
4.26.4.11 levellized_cost_of_energy_kWh . . . 224
4.26.4.12 n_points . . . 224
4.26.4.13 n_replacements . . . 225
4.26.4.14 n_years . . . 225
4.26.4.15 net_present_cost . . . 225
4.26.4.16 nominal_discount_annual . . . 225
4.26.4.17 nominal_inflation_annual . . . 225
4.26.4.18 operation_maintenance_cost_kWh . . . 225
4.26.4.19 operation_maintenance_cost_vec . . . 226
4.26.4.20 power_capacity_kW . . . 226
4.26.4.21 power_kW . . . 226
4.26.4.22 print_flag . . . 226
4.26.4.23 real_discount_annual . . . 226
4.26.4.24 total_discharge_kWh . . . 226
4.26.4.25 type . . . 227
4.26.4.26 type_str . . . 227
4.27 StorageInputs Struct Reference . . . 227
4.27.1 Detailed Description . . . 227
4.27.2 Member Data Documentation . . . 228
4.27.2.1 energy_capacity_kWh . . . 228
4.27.2.2 is_sunk . . . 228
4.27.2.3 nominal_discount_annual . . . 228
4.27.2.4 nominal_inflation_annual . . . 228
4.27.2.5 power_capacity_kW . . . 228
4.27.2.6 print_flag . . . 229
4.28 Tidal Class Reference . . . 229
4.28.1 Detailed Description . . . 231
4.28.2 Constructor & Destructor Documentation . . . 231
4.28.2.1 Tidal() [1/2] . . . 231
4.28.2.2 Tidal() [2/2] . . . 231
4.28.2.3 ∼Tidal() . . . 233
4.28.3 Member Function Documentation . . . 233
4.28.3.1 __checkInputs() . . . 233
4.28.3.2 __computeCubicProductionkW() . . . 233
4.28.3.3 __computeExponentialProductionkW() . . . 234
4.28.3.4 __computeLookupProductionkW() . . . 235
4.28.3.5 __getGenericCapitalCost() . . . 235
4.28.3.6 __getGenericOpMaintCost() . . . 236
4.28.3.7 __writeSummary() . . . 236
4.28.3.8 __writeTimeSeries() . . . 237
4.28.3.9 commit() . . . 238

4.28.3.10 computeProductionkW() 239
4.28.3.11 handleReplacement() 240
4.28.4 Member Data Documentation 240
4.28.4.1 design_speed_ms 241
4.28.4.2 power_model 241
4.28.4.3 power_model_string 241
4.29 TidalInputs Struct Reference 241
4.29.1 Detailed Description 242
4.29.2 Member Data Documentation 242
4.29.2.1 capital_cost 242
4.29.2.2 design_speed_ms 242
4.29.2.3 operation_maintenance_cost_kWh 243
4.29.2.4 power_model 243
4.29.2.5 renewable_inputs 243
4.29.2.6 resource_key 243
4.30 Wave Class Reference 244
4.30.1 Detailed Description 246
4.30.2 Constructor & Destructor Documentation 246
4.30.2.1 Wave() [1/2] 246
4.30.2.2 Wave() [2/2] 246
4.30.2.3 ∼Wave() 247
4.30.3 Member Function Documentation 248
4.30.3.1 __checkInputs() 248
4.30.3.2 __computeGaussianProductionkW() 248
4.30.3.3 __computeLookupProductionkW() 249
4.30.3.4 __computeParaboloidProductionkW() 250
4.30.3.5 __getGenericCapitalCost() 251
4.30.3.6 __getGenericOpMaintCost() 251
4.30.3.7 __writeSummary() 251
4.30.3.8 __writeTimeSeries() 253
4.30.3.9 commit() 254
4.30.3.10 computeProductionkW() 255
4.30.3.11 handleReplacement() 256
4.30.4 Member Data Documentation 256
4.30.4.1 design_energy_period_s 256
4.30.4.2 design_significant_wave_height_m 257
4.30.4.3 power_model 257
4.30.4.4 power_model_string 257
4.31 WaveInputs Struct Reference 257
4.31.1 Detailed Description 258
4.31.2 Member Data Documentation 258
4.31.2.1 capital_cost 258

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

5 File Documentation **275**

5.1 header/Controller.h File Reference 275
5.1.1 Detailed Description 276

5.1.2 Enumeration Type Documentation . . . 276
5.1.2.1 ControlMode . . . 276
5.2 header/doxygen_cite.h File Reference . . . 276
5.2.1 Detailed Description . . . 276
5.3 header/ElectricalLoad.h File Reference . . . 277
5.3.1 Detailed Description . . . 277
5.4 header/Interpolator.h File Reference . . . 277
5.4.1 Detailed Description . . . 278
5.5 header/Model.h File Reference . . . 278
5.5.1 Detailed Description . . . 279
5.6 header/Production/Combustion/Combustion.h File Reference . . . 279
5.6.1 Detailed Description . . . 280
5.6.2 Enumeration Type Documentation . . . 280
5.6.2.1 CombustionType . . . 280
5.6.2.2 FuelMode . . . 280
5.7 header/Production/Combustion/Diesel.h File Reference . . . 282
5.7.1 Detailed Description . . . 283
5.8 header/Production/Noncombustion/Hydro.h File Reference . . . 283
5.8.1 Detailed Description . . . 284
5.8.2 Enumeration Type Documentation . . . 284
5.8.2.1 HydroInterpKeys . . . 284
5.8.2.2 HydroTurbineType . . . 284
5.9 header/Production/Noncombustion/Noncombustion.h File Reference . . . 285
5.9.1 Enumeration Type Documentation . . . 286
5.9.1.1 NoncombustionType . . . 286
5.10 header/Production/Production.h File Reference . . . 286
5.10.1 Detailed Description . . . 287
5.11 header/Production/Renewable/Renewable.h File Reference . . . 287
5.11.1 Detailed Description . . . 288
5.11.2 Enumeration Type Documentation . . . 288
5.11.2.1 RenewableType . . . 288
5.12 header/Production/Renewable/Solar.h File Reference . . . 288
5.12.1 Detailed Description . . . 289
5.13 header/Production/Renewable/Tidal.h File Reference . . . 289
5.13.1 Detailed Description . . . 290
5.13.2 Enumeration Type Documentation . . . 290
5.13.2.1 TidalPowerProductionModel . . . 290
5.14 header/Production/Renewable/Wave.h File Reference . . . 291
5.14.1 Detailed Description . . . 292
5.14.2 Enumeration Type Documentation . . . 292
5.14.2.1 WavePowerProductionModel . . . 292
5.15 header/Production/Renewable/Wind.h File Reference . . . 292

5.15.1 Detailed Description 293
5.15.2 Enumeration Type Documentation 293
5.15.2.1 WindPowerProductionModel 293
5.16 header/Resources.h File Reference 294
5.16.1 Detailed Description 294
5.17 header/std_includes.h File Reference 295
5.17.1 Detailed Description 295
5.18 header/Storage/LiIon.h File Reference 295
5.18.1 Detailed Description 296
5.19 header/Storage/Storage.h File Reference 296
5.19.1 Detailed Description 297
5.19.2 Enumeration Type Documentation 297
5.19.2.1 StorageType 297
5.20 projects/example.cpp File Reference 298
5.20.1 Function Documentation 298
5.20.1.1 main() 298
5.21 pybindings/PYBIND11_PGM.cpp File Reference 302
5.21.1 Detailed Description 303
5.21.2 Function Documentation 303
5.21.2.1 PYBIND11_MODULE() 303
5.22 pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp File Reference 304
5.22.1 Detailed Description 304
5.22.2 Function Documentation 304
5.22.2.1 def_readwrite() 305
5.22.2.2 value() [1/2] 305
5.22.2.3 value() [2/2] 305
5.22.3 Variable Documentation 305
5.22.3.1 def_readwrite 305
5.23 pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp File Reference 305
5.23.1 Detailed Description 306
5.23.2 Function Documentation 307
5.23.2.1 def() 307
5.23.2.2 def_readwrite() [1/8] 307
5.23.2.3 def_readwrite() [2/8] 307
5.23.2.4 def_readwrite() [3/8] 307
5.23.2.5 def_readwrite() [4/8] 307
5.23.2.6 def_readwrite() [5/8] 308
5.23.2.7 def_readwrite() [6/8] 308
5.23.2.8 def_readwrite() [7/8] 308
5.23.2.9 def_readwrite() [8/8] 308
5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp File Reference 308
5.24.1 Detailed Description 309

5.24.2 Function Documentation . . . 309
5.24.2.1 def() . . . 309
5.24.2.2 def_readwrite() [1/9] . . . 310
5.24.2.3 def_readwrite() [2/9] . . . 310
5.24.2.4 def_readwrite() [3/9] . . . 310
5.24.2.5 def_readwrite() [4/9] . . . 310
5.24.2.6 def_readwrite() [5/9] . . . 310
5.24.2.7 def_readwrite() [6/9] . . . 310
5.24.2.8 def_readwrite() [7/9] . . . 311
5.24.2.9 def_readwrite() [8/9] . . . 311
5.24.2.10 def_readwrite() [9/9] . . . 311
5.24.2.11 value() [1/2] . . . 311
5.24.2.12 value() [2/2] . . . 311
5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp File Reference . . . 312
5.25.1 Detailed Description . . . 312
5.25.2 Function Documentation . . . 312
5.25.2.1 def() . . . 312
5.25.2.2 value() . . . 313
5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference . . . 313
5.26.1 Detailed Description . . . 314
5.26.2 Function Documentation . . . 314
5.26.2.1 def() . . . 315
5.26.2.2 def_readwrite() [1/17] . . . 315
5.26.2.3 def_readwrite() [2/17] . . . 315
5.26.2.4 def_readwrite() [3/17] . . . 315
5.26.2.5 def_readwrite() [4/17] . . . 315
5.26.2.6 def_readwrite() [5/17] . . . 316
5.26.2.7 def_readwrite() [6/17] . . . 316
5.26.2.8 def_readwrite() [7/17] . . . 316
5.26.2.9 def_readwrite() [8/17] . . . 316
5.26.2.10 def_readwrite() [9/17] . . . 316
5.26.2.11 def_readwrite() [10/17] . . . 317
5.26.2.12 def_readwrite() [11/17] . . . 317
5.26.2.13 def_readwrite() [12/17] . . . 317
5.26.2.14 def_readwrite() [13/17] . . . 317
5.26.2.15 def_readwrite() [14/17] . . . 317
5.26.2.16 def_readwrite() [15/17] . . . 318
5.26.2.17 def_readwrite() [16/17] . . . 318
5.26.2.18 def_readwrite() [17/17] . . . 318
5.27 pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp File Reference . . . 318
5.27.1 Detailed Description . . . 319
5.27.2 Function Documentation . . . 319

5.27.2.1 def() 319
5.27.2.2 value() [1/2] 319
5.27.2.3 value() [2/2] 319
5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference 320
5.28.1 Detailed Description 320
5.28.2 Function Documentation 320
5.28.2.1 def() 320
5.28.2.2 def_readwrite() [1/2] 321
5.28.2.3 def_readwrite() [2/2] 321
5.29 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference 321
5.29.1 Detailed Description 322
5.29.2 Function Documentation 322
5.29.2.1 def_readwrite() [1/2] 322
5.29.2.2 def_readwrite() [2/2] 322
5.29.2.3 value() [1/2] 322
5.29.2.4 value() [2/2] 322
5.29.3 Variable Documentation 323
5.29.3.1 def_readwrite 323
5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference 323
5.30.1 Detailed Description 324
5.30.2 Function Documentation 324
5.30.2.1 def_readwrite() [1/3] 324
5.30.2.2 def_readwrite() [2/3] 324
5.30.2.3 def_readwrite() [3/3] 324
5.30.2.4 value() [1/2] 325
5.30.2.5 value() [2/2] 325
5.30.3 Variable Documentation 325
5.30.3.1 def_readwrite 325
5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference 325
5.31.1 Detailed Description 326
5.31.2 Function Documentation 326
5.31.2.1 def_readwrite() [1/2] 326
5.31.2.2 def_readwrite() [2/2] 326
5.31.2.3 value() 326
5.31.3 Variable Documentation 327
5.31.3.1 def_readwrite 327
5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference 327
5.32.1 Detailed Description 328
5.32.2 Function Documentation 328
5.32.2.1 def() [1/3] 328
5.32.2.2 def() [2/3] 328
5.32.2.3 def() [3/3] 328

5.32.2.4 def_readwrite() [1/2] 328
5.32.2.5 def_readwrite() [2/2] 329
5.32.2.6 value() 329
5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference 329
5.33.1 Detailed Description 330
5.33.2 Function Documentation 330
5.33.2.1 def_readwrite() [1/4] 330
5.33.2.2 def_readwrite() [2/4] 330
5.33.2.3 def_readwrite() [3/4] 330
5.33.2.4 def_readwrite() [4/4] 330
5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference 331
5.34.1 Detailed Description 331
5.34.2 Function Documentation 331
5.34.2.1 def() 332
5.34.2.2 def_readwrite() [1/7] 332
5.34.2.3 def_readwrite() [2/7] 332
5.34.2.4 def_readwrite() [3/7] 332
5.34.2.5 def_readwrite() [4/7] 332
5.34.2.6 def_readwrite() [5/7] 332
5.34.2.7 def_readwrite() [6/7] 333
5.34.2.8 def_readwrite() [7/7] 333
5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference 333
5.35.1 Detailed Description 333
5.35.2 Variable Documentation 334
5.35.2.1 def_readwrite 334
5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference 334
5.36.1 Detailed Description 334
5.36.2 Function Documentation 335
5.36.2.1 def_readwrite() [1/2] 335
5.36.2.2 def_readwrite() [2/2] 335
5.37 pybindings/snippets/Storage/PYBIND11_LiIon.cpp File Reference 335
5.37.1 Detailed Description 336
5.37.2 Function Documentation 337
5.37.2.1 def() 337
5.37.2.2 def_readwrite() [1/18] 337
5.37.2.3 def_readwrite() [2/18] 337
5.37.2.4 def_readwrite() [3/18] 337
5.37.2.5 def_readwrite() [4/18] 337
5.37.2.6 def_readwrite() [5/18] 338
5.37.2.7 def_readwrite() [6/18] 338
5.37.2.8 def_readwrite() [7/18] 338
5.37.2.9 def_readwrite() [8/18] 338

5.37.2.10 def_readwrite() [9/18] . . . 338
5.37.2.11 def_readwrite() [10/18] . . . 339
5.37.2.12 def_readwrite() [11/18] . . . 339
5.37.2.13 def_readwrite() [12/18] . . . 339
5.37.2.14 def_readwrite() [13/18] . . . 339
5.37.2.15 def_readwrite() [14/18] . . . 339
5.37.2.16 def_readwrite() [15/18] . . . 340
5.37.2.17 def_readwrite() [16/18] . . . 340
5.37.2.18 def_readwrite() [17/18] . . . 340
5.37.2.19 def_readwrite() [18/18] . . . 340
5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference . . . 340
5.38.1 Detailed Description . . . 341
5.38.2 Function Documentation . . . 341
5.38.2.1 def_readwrite() [1/2] . . . 341
5.38.2.2 def_readwrite() [2/2] . . . 341
5.38.2.3 value() . . . 341
5.38.3 Variable Documentation . . . 342
5.38.3.1 def_readwrite . . . 342
5.39 source/Controller.cpp File Reference . . . 342
5.39.1 Detailed Description . . . 342
5.40 source/ElectricalLoad.cpp File Reference . . . 343
5.40.1 Detailed Description . . . 343
5.41 source/Interpolator.cpp File Reference . . . 343
5.41.1 Detailed Description . . . 343
5.42 source/Model.cpp File Reference . . . 344
5.42.1 Detailed Description . . . 344
5.43 source/Production/Combustion/Combustion.cpp File Reference . . . 344
5.43.1 Detailed Description . . . 345
5.44 source/Production/Combustion/Diesel.cpp File Reference . . . 345
5.44.1 Detailed Description . . . 345
5.45 source/Production/Noncombustion/Hydro.cpp File Reference . . . 345
5.45.1 Detailed Description . . . 346
5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference . . . 346
5.46.1 Detailed Description . . . 346
5.47 source/Production/Production.cpp File Reference . . . 347
5.47.1 Detailed Description . . . 347
5.48 source/Production/Renewable/Renewable.cpp File Reference . . . 347
5.48.1 Detailed Description . . . 347
5.49 source/Production/Renewable/Solar.cpp File Reference . . . 348
5.49.1 Detailed Description . . . 348
5.50 source/Production/Renewable/Tidal.cpp File Reference . . . 348
5.50.1 Detailed Description . . . 349

5.51 source/Production/Renewable/Wave.cpp File Reference 349
5.51.1 Detailed Description 349
5.52 source/Production/Renewable/Wind.cpp File Reference 349
5.52.1 Detailed Description 350
5.53 source/Resources.cpp File Reference 350
5.53.1 Detailed Description 350
5.54 source/Storage/LiIon.cpp File Reference 351
5.54.1 Detailed Description 351
5.55 source/Storage/Storage.cpp File Reference 351
5.55.1 Detailed Description 351
5.56 test/source/Production/Combustion/test_Combustion.cpp File Reference 352
5.56.1 Detailed Description 352
5.56.2 Function Documentation 352
5.56.2.1 main() 353
5.56.2.2 testConstruct_Combustion() 353
5.57 test/source/Production/Combustion/test_Diesel.cpp File Reference 354
5.57.1 Detailed Description 355
5.57.2 Function Documentation 356
5.57.2.1 main() 356
5.57.2.2 testBadConstruct_Diesel() 356
5.57.2.3 testCapacityConstraint_Diesel() 357
5.57.2.4 testCommit_Diesel() 357
5.57.2.5 testConstruct_Diesel() 359
5.57.2.6 testConstructLookup_Diesel() 360
5.57.2.7 testEconomics_Diesel() 360
5.57.2.8 testFuelConsumptionEmissions_Diesel() 361
5.57.2.9 testFuelLookup_Diesel() 363
5.57.2.10 testMinimumLoadRatioConstraint_Diesel() 364
5.57.2.11 testMinimumRuntimeConstraint_Diesel() 364
5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference 365
5.58.1 Detailed Description 366
5.58.2 Function Documentation 366
5.58.2.1 main() 366
5.58.2.2 testCommit_Hydro() 367
5.58.2.3 testConstruct_Hydro() 368
5.58.2.4 testEfficiencyInterpolation_Hydro() 369
5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference 370
5.59.1 Detailed Description 371
5.59.2 Function Documentation 371
5.59.2.1 main() 371
5.59.2.2 testConstruct_Noncombustion() 371
5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference 372

5.60.1 Detailed Description 373
5.60.2 Function Documentation 373
5.60.2.1 main() 373
5.60.2.2 testConstruct_Renewable() 373
5.61 test/source/Production/Renewable/test_Solar.cpp File Reference 374
5.61.1 Detailed Description 375
5.61.2 Function Documentation 375
5.61.2.1 main() 375
5.61.2.2 testBadConstruct_Solar() 376
5.61.2.3 testCommit_Solar() 376
5.61.2.4 testConstruct_Solar() 378
5.61.2.5 testEconomics_Solar() 379
5.61.2.6 testProductionConstraint_Solar() 379
5.61.2.7 testProductionOverride_Solar() 380
5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference 381
5.62.1 Detailed Description 382
5.62.2 Function Documentation 382
5.62.2.1 main() 382
5.62.2.2 testBadConstruct_Tidal() 383
5.62.2.3 testCommit_Tidal() 383
5.62.2.4 testConstruct_Tidal() 384
5.62.2.5 testEconomics_Tidal() 385
5.62.2.6 testProductionConstraint_Tidal() 386
5.63 test/source/Production/Renewable/test_Wave.cpp File Reference 386
5.63.1 Detailed Description 387
5.63.2 Function Documentation 387
5.63.2.1 main() 388
5.63.2.2 testBadConstruct_Wave() 388
5.63.2.3 testCommit_Wave() 389
5.63.2.4 testConstruct_Wave() 390
5.63.2.5 testConstructLookup_Wave() 391
5.63.2.6 testEconomics_Wave() 392
5.63.2.7 testProductionConstraint_Wave() 392
5.63.2.8 testProductionLookup_Wave() 392
5.64 test/source/Production/Renewable/test_Wind.cpp File Reference 394
5.64.1 Detailed Description 394
5.64.2 Function Documentation 395
5.64.2.1 main() 395
5.64.2.2 testBadConstruct_Wind() 395
5.64.2.3 testCommit_Wind() 396
5.64.2.4 testConstruct_Wind() 397
5.64.2.5 testEconomics_Wind() 398

5.64.2.6 testProductionConstraint_Wind() 399
5.65 test/source/Production/test_Production.cpp File Reference 399
5.65.1 Detailed Description 400
5.65.2 Function Documentation 400
5.65.2.1 main() 400
5.65.2.2 testBadConstruct_Production() 401
5.65.2.3 testConstruct_Production() 401
5.66 test/source/Storage/test_LiIon.cpp File Reference 402
5.66.1 Detailed Description 403
5.66.2 Function Documentation 403
5.66.2.1 main() 404
5.66.2.2 testBadConstruct_LiIon() 404
5.66.2.3 testCommitCharge_LiIon() 405
5.66.2.4 testCommitDischarge_LiIon() 405
5.66.2.5 testConstruct_LiIon() 406
5.67 test/source/Storage/test_Storage.cpp File Reference 407
5.67.1 Detailed Description 408
5.67.2 Function Documentation 408
5.67.2.1 main() 408
5.67.2.2 testBadConstruct_Storage() 409
5.67.2.3 testConstruct_Storage() 409
5.68 test/source/test_Controller.cpp File Reference 410
5.68.1 Detailed Description 411
5.68.2 Function Documentation 411
5.68.2.1 main() 411
5.68.2.2 testConstruct_Controller() 412
5.69 test/source/test_ElectricalLoad.cpp File Reference 412
5.69.1 Detailed Description 412
5.69.2 Function Documentation 413
5.69.2.1 main() 413
5.69.2.2 testConstruct_ElectricalLoad() 413
5.69.2.3 testDataRead_ElectricalLoad() 414
5.69.2.4 testPostConstructionAttributes_ElectricalLoad() 415
5.70 test/source/test_Interpolator.cpp File Reference 416
5.70.1 Detailed Description 416
5.70.2 Function Documentation 416
5.70.2.1 main() 417
5.70.2.2 testBadIndexing1D_Interpolator() 417
5.70.2.3 testConstruct_Interpolator() 418
5.70.2.4 testDataRead1D_Interpolator() 418
5.70.2.5 testDataRead2D_Interpolator() 420
5.70.2.6 testInterpolation1D_Interpolator() 422

5.70.2.7 testInterpolation2D_Interpolator() 423
5.70.2.8 testInvalidInterpolation1D_Interpolator() 424
5.70.2.9 testInvalidInterpolation2D_Interpolator() 425
5.71 test/source/test_Model.cpp File Reference 426
5.71.1 Detailed Description 427
5.71.2 Function Documentation 427
5.71.2.1 main() 428
5.71.2.2 testAddDiesel_Model() 429
5.71.2.3 testAddHydro_Model() 430
5.71.2.4 testAddHydroResource_Model() 431
5.71.2.5 testAddLiIon_Model() 432
5.71.2.6 testAddSolar_Model() 433
5.71.2.7 testAddSolar_productionOverride_Model() 433
5.71.2.8 testAddSolarResource_Model() 434
5.71.2.9 testAddTidal_Model() 435
5.71.2.10 testAddTidalResource_Model() 436
5.71.2.11 testAddWave_Model() 437
5.71.2.12 testAddWaveResource_Model() 437
5.71.2.13 testAddWind_Model() 439
5.71.2.14 testAddWindResource_Model() 440
5.71.2.15 testBadConstruct_Model() 441
5.71.2.16 testConstruct_Model() 442
5.71.2.17 testEconomics_Model() 442
5.71.2.18 testElectricalLoadData_Model() 442
5.71.2.19 testFuelConsumptionEmissions_Model() 444
5.71.2.20 testLoadBalance_Model() 444
5.71.2.21 testPostConstructionAttributes_Model() 446
5.72 test/source/test_Resources.cpp File Reference 446
5.72.1 Detailed Description 447
5.72.2 Function Documentation 448
5.72.2.1 main() 448
5.72.2.2 testAddHydroResource_Resources() 449
5.72.2.3 testAddSolarResource_Resources() 450
5.72.2.4 testAddTidalResource_Resources() 451
5.72.2.5 testAddWaveResource_Resources() 454
5.72.2.6 testAddWindResource_Resources() 455
5.72.2.7 testBadAdd_Resources() 457
5.72.2.8 testConstruct_Resources() 458
5.73 test/utils/testing_utils.cpp File Reference 458
5.73.1 Detailed Description 459
5.73.2 Function Documentation 459
5.73.2.1 expectedErrorNotDetected() 459

5.73.2.2 printGold() 460
5.73.2.3 printGreen() 460
5.73.2.4 printRed() 460
5.73.2.5 testFloatEquals() 461
5.73.2.6 testGreaterThan() 461
5.73.2.7 testGreaterThanOrEqualTo() 462
5.73.2.8 testLessThan() 463
5.73.2.9 testLessThanOrEqualTo() 463
5.73.2.10 testTruth() 464
5.74 test/utils/testing_utils.h File Reference 464
5.74.1 Detailed Description 465
5.74.2 Macro Definition Documentation 466
5.74.2.1 FLOAT_TOLERANCE 466
5.74.3 Function Documentation 466
5.74.3.1 expectedErrorNotDetected() 466
5.74.3.2 printGold() 466
5.74.3.3 printGreen() 467
5.74.3.4 printRed() 467
5.74.3.5 testFloatEquals() 467
5.74.3.6 testGreaterThan() 468
5.74.3.7 testGreaterThanOrEqualTo() 469
5.74.3.8 testLessThan() 469
5.74.3.9 testLessThanOrEqualTo() 470
5.74.3.10 testTruth() 470

Bibliography **474**

Index **475**

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

- CombustionInputs . . . 24
- Controller . . . 25
- DieselInputs . . . 59
- ElectricalLoad . . . 63
- Emissions . . . 68
- HydroInputs . . . 89
- Interpolator . . . 91
- InterpolatorStruct1D . . . 105
- InterpolatorStruct2D . . . 106
- LiIonInputs . . . 128
- Model . . . 133
- ModelInputs . . . 152
- NoncombustionInputs . . . 161
- Production . . . 162
  - Combustion . . . 9
    - Diesel . . . 45
  - Noncombustion . . . 153
    - Hydro . . . 70
  - Renewable . . . 180
    - Solar . . . 203
    - Tidal . . . 229
    - Wave . . . 244
    - Wind . . . 260
- ProductionInputs . . . 178
- RenewableInputs . . . 188
- Resources . . . 189
- SolarInputs . . . 212
- Storage . . . 214
  - LiIon . . . 109
- StorageInputs . . . 227
- TidalInputs . . . 241
- WaveInputs . . . 257
- WindInputs . . . 272

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

LiIonInputs
A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs 128
Model
A container class which forms the centre of PGMcpp. The Model class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes 133
ModelInputs
A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except path_2_electrical_load_time_series, for which a valid input must be provided) 152
Noncombustion
The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion 153
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 161
Production
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise 162
ProductionInputs
A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input 178
Renewable
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy 180
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 . . . 188
Resources
A class which contains renewable resource data. Intended to serve as a component class of Model 189
Solar
A derived class of the Renewable branch of Production which models solar production 203
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 212
Storage
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy 214
StorageInputs
A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input 227
Tidal
A derived class of the Renewable branch of Production which models tidal production 229
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 241
Wave
A derived class of the Renewable branch of Production which models wave production 244
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 257
Wind
A derived class of the Renewable branch of Production which models wind production 260
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 272

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 275
header/doxygen_cite.h
Header file which simply cites the doxygen tool 276
header/ElectricalLoad.h
Header file for the ElectricalLoad class 277
header/Interpolator.h
Header file for the Interpolator class 277
header/Model.h
Header file for the Model class 278
header/Resources.h
Header file for the Resources class 294
header/std_includes.h
Header file which simply batches together some standard includes 295
header/Production/Production.h
Header file for the Production class 286
header/Production/Combustion/Combustion.h
Header file for the Combustion class 279
header/Production/Combustion/Diesel.h
Header file for the Diesel class 282
header/Production/Noncombustion/Hydro.h
Header file for the Hydro class 283
header/Production/Noncombustion/Noncombustion.h 285
header/Production/Renewable/Renewable.h
Header file for the Renewable class 287
header/Production/Renewable/Solar.h
Header file for the Solar class 288
header/Production/Renewable/Tidal.h
Header file for the Tidal class 289
header/Production/Renewable/Wave.h
Header file for the Wave class 291
header/Production/Renewable/Wind.h
Header file for the Wind class 292
header/Storage/LiIon.h
Header file for the LiIon class 295

header/Storage/Storage.h
Header file for the Storage class 296
projects/example.cpp 298
pybindings/PYBIND11_PGM.cpp
Bindings file for PGMcpp 302
pybindings/snippets/PYBIND11_Controller.cpp
Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp 327
pybindings/snippets/PYBIND11_ElectricalLoad.cpp
Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp . . 329
pybindings/snippets/PYBIND11_Interpolator.cpp
Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp . . . 331
pybindings/snippets/PYBIND11_Model.cpp
Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp 333
pybindings/snippets/PYBIND11_Resources.cpp
Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp 334
pybindings/snippets/Production/PYBIND11_Production.cpp
Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp 313
pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp
Bindings file for the Combustion class. Intended to be #include'd in PYBIND11_PGM.cpp . . . 304
pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp
Bindings file for the Diesel class. Intended to be #include'd in PYBIND11_PGM.cpp 305
pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp
Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp 308
pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp
Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp . 312
pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp
Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp . . . 318
pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp
Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp 320
pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp
Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp 321
pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp
Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp 323
pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp
Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp 325
pybindings/snippets/Storage/PYBIND11_LiIon.cpp
Bindings file for the LiIon class. Intended to be #include'd in PYBIND11_PGM.cpp 335
pybindings/snippets/Storage/PYBIND11_Storage.cpp
Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp 340
source/Controller.cpp
Implementation file for the Controller class 342
source/ElectricalLoad.cpp
Implementation file for the ElectricalLoad class 343
source/Interpolator.cpp
Implementation file for the Interpolator class 343
source/Model.cpp
Implementation file for the Model class 344
source/Resources.cpp
Implementation file for the Resources class 350
source/Production/Production.cpp
Implementation file for the Production class 347
source/Production/Combustion/Combustion.cpp
Implementation file for the Combustion class 344
source/Production/Combustion/Diesel.cpp
Implementation file for the Diesel class 345
source/Production/Noncombustion/Hydro.cpp
Implementation file for the Hydro class 345

source/Production/Noncombustion/Noncombustion.cpp
Implementation file for the Noncombustion class . . . 346
source/Production/Renewable/Renewable.cpp
Implementation file for the Renewable class . . . 347
source/Production/Renewable/Solar.cpp
Implementation file for the Solar class . . . 348
source/Production/Renewable/Tidal.cpp
Implementation file for the Tidal class . . . 348
source/Production/Renewable/Wave.cpp
Implementation file for the Wave class . . . 349
source/Production/Renewable/Wind.cpp
Implementation file for the Wind class . . . 349
source/Storage/LiIon.cpp
Implementation file for the LiIon class . . . 351
source/Storage/Storage.cpp
Implementation file for the Storage class . . . 351
test/source/test_Controller.cpp
Testing suite for Controller class . . . 410
test/source/test_ElectricalLoad.cpp
Testing suite for ElectricalLoad class . . . 412
test/source/test_Interpolator.cpp
Testing suite for Interpolator class . . . 416
test/source/test_Model.cpp
Testing suite for Model class . . . 426
test/source/test_Resources.cpp
Testing suite for Resources class . . . 446
test/source/Production/test_Production.cpp
Testing suite for Production class . . . 399
test/source/Production/Combustion/test_Combustion.cpp
Testing suite for Combustion class . . . 352
test/source/Production/Combustion/test_Diesel.cpp
Testing suite for Diesel class . . . 354
test/source/Production/Noncombustion/test_Hydro.cpp
Testing suite for Hydro class . . . 365
test/source/Production/Noncombustion/test_Noncombustion.cpp
Testing suite for Noncombustion class . . . 370
test/source/Production/Renewable/test_Renewable.cpp
Testing suite for Renewable class . . . 372
test/source/Production/Renewable/test_Solar.cpp
Testing suite for Solar class . . . 374
test/source/Production/Renewable/test_Tidal.cpp
Testing suite for Tidal class . . . 381
test/source/Production/Renewable/test_Wave.cpp
Testing suite for Wave class . . . 386
test/source/Production/Renewable/test_Wind.cpp
Testing suite for Wind class . . . 394
test/source/Storage/test_LiIon.cpp
Testing suite for LiIon class . . . 402
test/source/Storage/test_Storage.cpp
Testing suite for Storage class . . . 407
test/utils/testing_utils.cpp
Implementation file for various PGMcpp testing utilities . . . 458
test/utils/testing_utils.h
Header file for various PGMcpp testing utilities . . . 464

Chapter 4

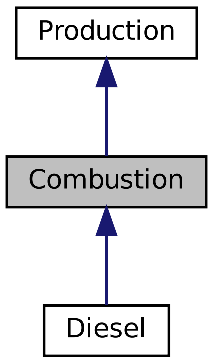
Class Documentation

4.1 Combustion Class Reference

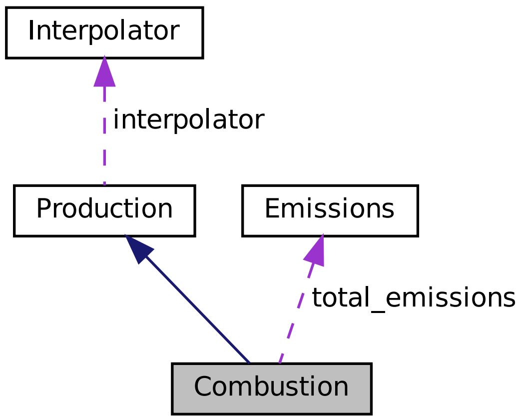
The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

- Combustion (void)

    Constructor (dummy) for the Combustion class.
- Combustion (int, double, CombustionInputs, std::vector< double > ∗)

    Constructor (intended) for the Combustion class.
- virtual void handleReplacement (int)

    Method to handle asset replacement and capital cost incursion, if applicable.
- void computeFuelAndEmissions (void)

    Helper method to compute the total fuel consumption and emissions over the Model run.
- void computeEconomics (std::vector< double > ∗)

    Helper method to compute key economic metrics for the Model run.
- virtual double requestProductionkW (int, double, double)
- virtual double commit (int, double, double, double)

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

    Method which takes in production and returns volume of fuel burned over the given interval of time.
- Emissions getEmissionskg (double)

    Method which takes in volume of fuel consumed and returns mass spectrum of resulting emissions.
- void writeResults (std::string, std::vector< double > ∗, int, int=-1)

    Method which writes Combustion results to an output directory.
- virtual ∼Combustion (void)

    Destructor for the Combustion class.

Public Attributes

- CombustionType type

  The type (CombustionType) of the asset.
- FuelMode fuel_mode

  The fuel mode to use in modelling fuel consumption.
- Emissions total_emissions

  An Emissions structure for holding total emissions [kg].
- double fuel_cost_L

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

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

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

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

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

```
{
    return;
}   /* Combustion() */
```

4.1.2.2 Combustion() [2/2]

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

Constructor (intended) for the Combustion class.

Parameters

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

```
  :
Production(
    n_points,
    n_years,
    combustion_inputs.production_inputs,
    time_vec_hrs_ptr
)
```

```
{
    //  1. check inputs
    this->__checkInputs(combustion_inputs);

    //  2. set attributes
    this->fuel_mode = combustion_inputs.fuel_mode;

    switch (this->fuel_mode) {
        case (FuelMode :: FUEL_MODE_LINEAR): {
            this->fuel_mode_str = "FUEL_MODE_LINEAR";

            break;
        }

        case (FuelMode :: FUEL_MODE_LOOKUP): {
            this->fuel_mode_str = "FUEL_MODE_LOOKUP";

            this->interpolator.addData1D(
                0,
                combustion_inputs.path_2_fuel_interp_data
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Combustion():  ";
            error_str += "fuel mode ";
            error_str += std::to_string(this->fuel_mode);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    this->fuel_cost_L = 0;
    this->nominal_fuel_escalation_annual =
        combustion_inputs.nominal_fuel_escalation_annual;

    this->real_fuel_escalation_annual = this->computeRealDiscountAnnual(
        combustion_inputs.nominal_fuel_escalation_annual,
        combustion_inputs.production_inputs.nominal_discount_annual
    );

    this->linear_fuel_slope_LkWh = 0;
    this->linear_fuel_intercept_LkWh = 0;

    this->CO2_emissions_intensity_kgL = 0;
    this->CO_emissions_intensity_kgL = 0;
    this->NOx_emissions_intensity_kgL = 0;
    this->SOx_emissions_intensity_kgL = 0;
    this->CH4_emissions_intensity_kgL = 0;
    this->PM_emissions_intensity_kgL = 0;

    this->total_fuel_consumed_L = 0;

    this->fuel_consumption_vec_L.resize(this->n_points, 0);
    this->fuel_cost_vec.resize(this->n_points, 0);

    this->CO2_emissions_vec_kg.resize(this->n_points, 0);
    this->CO_emissions_vec_kg.resize(this->n_points, 0);
    this->NOx_emissions_vec_kg.resize(this->n_points, 0);
    this->SOx_emissions_vec_kg.resize(this->n_points, 0);
    this->CH4_emissions_vec_kg.resize(this->n_points, 0);
    this->PM_emissions_vec_kg.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Combustion object constructed at " « this « std::endl;
    }

    return;
}   /* Combustion() */
```

4.1.2.3 ∼Combustion()

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

Destructor for the Combustion class.

```
{
    // 1. destruction print
    if (this->print_flag) {
        std::cout « "Combustion object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

combustion_inputs	A structure of Combustion constructor inputs.

```
{
    // 1. if FUEL_MODE_LOOKUP, check that path is given
    if (
        combustion_inputs.fuel_mode == FuelMode :: FUEL_MODE_LOOKUP and
        combustion_inputs.path_2_fuel_interp_data.empty()
    ) {
        std::string error_str = "ERROR:  Combustion()  fuel mode was set to ";
        error_str += "FuelMode::FUEL_MODE_LOOKUP, but no path to fuel interpolation ";
        error_str += "data was given";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.1.3.2 __writeSummary()

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

Reimplemented in Diesel.

```
{return;}
```

4.1.3.3 __writeTimeSeries()

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

Reimplemented in Diesel.

```
{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

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

Returns

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

Reimplemented from Production.

Reimplemented in Diesel.

```
{
    //  1. invoke base class method
    load_kW = Production :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    if (this->is_running) {
        //  2. compute and record fuel consumption
        double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
        this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;

        //  3. compute and record emissions
        Emissions emissions = this->getEmissionskg(fuel_consumed_L);
        this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
        this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
        this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
        this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
        this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
        this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;

        //  4. incur fuel costs
```

```
        this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
    }

    return load_kW;
}   /* 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

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
{
    //  1. account for fuel costs in net present cost
    double t_hrs = 0;
    double real_fuel_escalation_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_fuel_escalation_scalar = 1.0 / pow(
            1 + this->real_fuel_escalation_annual,
            t_hrs / 8760
        );

        this->net_present_cost += real_fuel_escalation_scalar * this->fuel_cost_vec[i];
    }

    //  2. invoke base class method
    Production :: computeEconomics(time_vec_hrs_ptr);

    return;
}   /* computeEconomics() */
```

4.1.3.6 computeFuelAndEmissions()

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

Helper method to compute the total fuel consumption and emissions over the Model run.

```
{
    for (int i = 0; i < n_points; i++) {
        this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];

        this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
        this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
        this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
        this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
        this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
        this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
    }

    return;
}   /* 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

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

Returns

A structure containing the mass spectrum of resulting emissions.

```
                                                              {
    Emissions emissions;

    emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
    emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
    emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
    emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
    emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
    emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;

    return emissions;
}   /* 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

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

Returns

The volume of fuel consumed [L].

```
{
    double fuel_consumed_L = 0;

    switch (this->fuel_mode) {
        case (FuelMode :: FUEL_MODE_LINEAR): {
            fuel_consumed_L = (
                this->linear_fuel_slope_LkWh * production_kW +
                this->linear_fuel_intercept_LkWh * this->capacity_kW
            ) * dt_hrs;

            break;
        }

        case (FuelMode :: FUEL_MODE_LOOKUP): {
```

```
            double load_ratio = production_kW / this->capacity_kW;

            fuel_consumed_L = this->interpolator.interp1D(0, load_ratio) * dt_hrs;

            break;
        }

        default: {
            std::string error_str = "ERROR:  Combustion::getFuelConsumptionL():  ";
            error_str += "fuel mode ";
            error_str += std::to_string(this->fuel_mode);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return fuel_consumed_L;
} /* getFuelConsumptionL() */
```

4.1.3.9 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Diesel.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Production :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.1.3.10 requestProductionkW()

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

Reimplemented in Diesel.

```
{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

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
combustion_index	An integer which corresponds to the index of the Combustion asset in the Model.
max_lines	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.

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Production/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += "Combustion/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->capacity_kW)));
    write_path += "kW_idx";
    write_path += std::to_string(combustion_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
    }

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

4.1.4 Member Data Documentation

4.1.4.1 CH4_emissions_intensity_kgL

```
double Combustion::CH4_emissions_intensity_kgL
```

Methane (CH_4) 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:

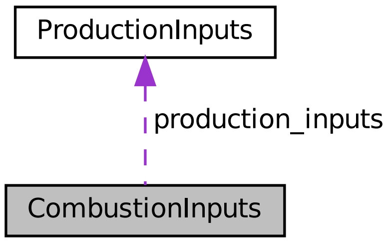
- header/Production/Combustion/Combustion.h
- source/Production/Combustion/Combustion.cpp

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.

```
{
    return;
}   /* Controller() */
```

4.3.2.2 ∼Controller()

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

Destructor for the Controller class.

```
{
    this->clear();

    return;
}   /* ~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

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

```
{
    //  1. default to load following
    this->__applyLoadFollowingControl_CHARGING(
        timestep,
        electrical_load_ptr,
        resources_ptr,
        combustion_ptr_vec_ptr,
        noncombustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr,
        storage_ptr_vec_ptr
    );

    return;
}   /* __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

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

curtailment

```
{
    //  1. get dt_hrs, net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = this->net_load_vec_kW[timestep];

    //  2. partition Storage assets into depleted and non-depleted
    std::list<Storage*> depleted_storage_ptr_list;
```

```
    std::list<Storage*> nondepleted_storage_ptr_list;

    Storage* storage_ptr;
    for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
        storage_ptr = storage_ptr_vec_ptr->at(i);

        if (storage_ptr->is_depleted) {
            depleted_storage_ptr_list.push_back(storage_ptr);
        }

        else {
            nondepleted_storage_ptr_list.push_back(storage_ptr);
        }
    }

    //  3. discharge non-depleted storage assets
    net_load_kW = this->__handleStorageDischarging(
        timestep,
        dt_hrs,
        net_load_kW,
        nondepleted_storage_ptr_list
    );

    //  4. request optimal production from all Noncombustion assets
    net_load_kW = this->__handleNoncombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        noncombustion_ptr_vec_ptr,
        resources_ptr
    );

    //  5. request optimal production from all Combustion assets
    //     default to load following if no depleted storage
    if (depleted_storage_ptr_list.empty()) {
        net_load_kW = this->__handleCombustionDispatch(
            timestep,
            dt_hrs,
            net_load_kW,
            combustion_ptr_vec_ptr,
            false   // is_cycle_charging
        );
    }

    else {
        net_load_kW = this->__handleCombustionDispatch(
            timestep,
            dt_hrs,
            net_load_kW,
            combustion_ptr_vec_ptr,
            true    // is_cycle_charging
        );
    }

    //  6. attempt to charge depleted Storage assets using any and all available
    //     charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        depleted_storage_ptr_list,
        combustion_ptr_vec_ptr,
        noncombustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    //  7. record any missed load
    if (net_load_kW > 1e-6) {
        this->missed_load_vec_kW[timestep] = net_load_kW;
    }

    return;
}   /* __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

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

```
{
    //  1. get dt_hrs, set net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = 0;

    //  2. request zero production from all Combustion assets
    this->__handleCombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        combustion_ptr_vec_ptr,
        false   // is_cycle_charging
    );

    //  3. request zero production from all Noncombustion assets
    this->__handleNoncombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        noncombustion_ptr_vec_ptr,
        resources_ptr
    );

    //  4. attempt to charge all Storage assets using any and all available curtailment
    //     charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        storage_ptr_vec_ptr,
        combustion_ptr_vec_ptr,
        noncombustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    return;
}   /* __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

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

curtailment

```
{
    //  1. get dt_hrs, net load
    double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
    double net_load_kW = this->net_load_vec_kW[timestep];

    //  2. partition Storage assets into depleted and non-depleted
    std::list<Storage*> depleted_storage_ptr_list;
    std::list<Storage*> nondepleted_storage_ptr_list;

    Storage* storage_ptr;
    for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
        storage_ptr = storage_ptr_vec_ptr->at(i);

        if (storage_ptr->is_depleted) {
            depleted_storage_ptr_list.push_back(storage_ptr);
        }

        else {
            nondepleted_storage_ptr_list.push_back(storage_ptr);
        }
    }

    //  3. discharge non-depleted storage assets
    net_load_kW = this->__handleStorageDischarging(
        timestep,
        dt_hrs,
        net_load_kW,
        nondepleted_storage_ptr_list
    );

    //  4. request optimal production from all Noncombustion assets
    net_load_kW = this->__handleNoncombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        noncombustion_ptr_vec_ptr,
        resources_ptr
    );

    //  5. request optimal production from all Combustion assets
    net_load_kW = this->__handleCombustionDispatch(
        timestep,
        dt_hrs,
        net_load_kW,
        combustion_ptr_vec_ptr,
        false   // is_cycle_charging
    );

    //  6. attempt to charge depleted Storage assets using any and all available
    //     charge priority is Combustion, then Renewable
    this->__handleStorageCharging(
        timestep,
        dt_hrs,
        depleted_storage_ptr_list,
        combustion_ptr_vec_ptr,
        noncombustion_ptr_vec_ptr,
        renewable_ptr_vec_ptr
    );

    //  7. record any missed load
    if (net_load_kW > 1e-6) {
        this->missed_load_vec_kW[timestep] = net_load_kW;
    }

    return;
}   /* __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

electrical_load_ptr	A pointer to the ElectricalLoad component of the Model.
renewable_ptr_vec_ptr	A pointer to the Renewable pointer vector of the Model.
resources_ptr	A pointer to the Resources component of the Model.

```
{
    //  1. init
    this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
    this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);

    //  2. populate net load vector
    double dt_hrs = 0;
    double load_kW = 0;
    double net_load_kW = 0;
    double production_kW = 0;

    Renewable* renewable_ptr;

    for (int i = 0; i < electrical_load_ptr->n_points; i++) {
        dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
        load_kW = electrical_load_ptr->load_vec_kW[i];
        net_load_kW = load_kW;

        for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {
            renewable_ptr = renewable_ptr_vec_ptr->at(j);

            production_kW = this->__getRenewableProduction(
                i,
                dt_hrs,
                renewable_ptr,
                resources_ptr
            );

            load_kW = renewable_ptr->commit(
                i,
                dt_hrs,
                production_kW,
                load_kW
            );

            net_load_kW -= production_kW;
        }

        this->net_load_vec_kW[i] = net_load_kW;
    }

    return;
}   /* __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

combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.

```
{
    //  1. get state table dimensions
    int n_cols = combustion_ptr_vec_ptr->size();
    int n_rows = pow(2, n_cols);

    //  2. init state table (all possible on/off combinations)
    std::vector<std::vector<bool» state_table;
    state_table.resize(n_rows, {});

    int x = 0;
    for (int i = 0; i < n_rows; i++) {
        state_table[i].resize(n_cols, false);

        x = i;
        for (int j = 0; j < n_cols; j++) {
            if (x % 2 == 0) {
                state_table[i][j] = true;
            }
            x /= 2;
        }
    }

    //  3. construct combustion map (handle duplicates by keeping rows with minimum
    //     trues)
    double total_capacity_kW = 0;
    int truth_count = 0;
    int current_truth_count = 0;

    for (int i = 0; i < n_rows; i++) {
        total_capacity_kW = 0;
        truth_count = 0;
        current_truth_count = 0;

        for (int j = 0; j < n_cols; j++) {
            if (state_table[i][j]) {
                total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
                truth_count++;
            }
        }

        if (this->combustion_map.count(total_capacity_kW) > 0) {
            for (int j = 0; j < n_cols; j++) {
                if (this->combustion_map[total_capacity_kW][j]) {
                    current_truth_count++;
                }
            }

            if (truth_count < current_truth_count) {
                this->combustion_map.erase(total_capacity_kW);
            }
        }

        this->combustion_map.insert(
            std::pair<double, std::vector<bool» (
                total_capacity_kW,
                state_table[i]
            )
        );
    }

    /*
    // ==== TEST PRINT ==== //
    std::cout « std::endl;

    std::cout « "\t\t";
    for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
        std::cout « combustion_ptr_vec_ptr->at(i)->capacity_kW « "\t";
    }
    std::cout « std::endl;

    std::map<double, std::vector<bool»::iterator iter;
    for (
        iter = this->combustion_map.begin();
        iter != this->combustion_map.end();
        iter++
    ) {
        std::cout « iter->first « ":\t{\t";

        for (size_t i = 0; i < iter->second.size(); i++) {
            std::cout « iter->second[i] « "\t";
        }
        std::cout « "}" « std::endl;
```

```
    }
    // ==== END TEST PRINT ==== //
    //*/

    return;
}   /* __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

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

Returns

The production [kW] of the Renewable asset.

```
{
    double production_kW = 0;

    switch (renewable_ptr->type) {
        case (RenewableType :: SOLAR): {
            double resource_value = 0;

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

            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resource_value
            );

            break;
        }

        case (RenewableType :: TIDAL): {
            double resource_value = 0;

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

            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resource_value
            );

            break;
        }

        case (RenewableType :: WAVE): {
            double significant_wave_height_m = 0;
```

```
            double energy_period_s = 0;

            if (not renewable_ptr->normalized_production_series_given) {
                significant_wave_height_m =
                    resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0];

                energy_period_s =
                    resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1];
            }

            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        case (RenewableType :: WIND): {
            double resource_value = 0;

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

            production_kW = renewable_ptr->computeProductionkW(
                timestep,
                dt_hrs,
                resource_value
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Controller::__getRenewableProduction():  ";
            error_str += "renewable type ";
            error_str += std::to_string(renewable_ptr->type);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* __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

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
net_load_kW	The net load [kW] before the dispatch is deducted from it.
combustion_ptr_vec_ptr	A pointer to the Combustion pointer vector of the Model.
is_cycle_charging	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.

```
{
    //  1. get minimal Combustion dispatch
    double target_production_kW = 1.2 * net_load_kW;
    double total_capacity_kW = 0;

    std::map<double, std::vector<bool>>::iterator iter = this->combustion_map.begin();
    while (iter != std::prev(this->combustion_map.end(), 1)) {
        if (target_production_kW <= total_capacity_kW) {
            break;
        }

        iter++;
        total_capacity_kW = iter->first;
    }

    //  2. share load proportionally (by rated capacity) over active diesels
    Combustion* combustion_ptr;
    double production_kW = 0;
    double request_kW = 0;
    double _net_load_kW = net_load_kW;

    for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
        combustion_ptr = combustion_ptr_vec_ptr->at(i);

        if (total_capacity_kW > 0) {
            request_kW =
                int(this->combustion_map[total_capacity_kW][i]) *
                net_load_kW *
                (combustion_ptr->capacity_kW / total_capacity_kW);
        }

        else {
            request_kW = 0;
        }

        if (is_cycle_charging and request_kW > 0) {
            if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
                request_kW = 0.85 * combustion_ptr->capacity_kW;
            }
        }

        production_kW = combustion_ptr->requestProductionkW(
            timestep,
            dt_hrs,
            request_kW
        );

        _net_load_kW = combustion_ptr->commit(
            timestep,
            dt_hrs,
            production_kW,
            _net_load_kW
        );
    }

    return _net_load_kW;
}   /* __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]
```

```
{
    Noncombustion* noncombustion_ptr;
    double production_kW = 0;

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

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

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

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

                net_load_kW = noncombustion_ptr->commit(
                    timestep,
                    dt_hrs,
                    production_kW,
                    net_load_kW,
                    resource_value
                );

                break;
            }

            default: {
                production_kW = noncombustion_ptr->requestProductionkW(
                    timestep,
                    dt_hrs,
                    net_load_kW
                );

                net_load_kW = noncombustion_ptr->commit(
                    timestep,
                    dt_hrs,
                    production_kW,
                    net_load_kW
                );

                break;
            }
        }
    }

    return net_load_kW;
}   /* __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

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

```
{
    double acceptable_kW = 0;
    double curtailment_kW = 0;

    Storage* storage_ptr;
    Combustion* combustion_ptr;
    Noncombustion* noncombustion_ptr;
    Renewable* renewable_ptr;

    std::list<Storage*>::iterator iter;
    for (
        iter = storage_ptr_list.begin();
        iter != storage_ptr_list.end();
        iter++
    ){
        storage_ptr = (*iter);

        //  1. attempt to charge from Combustion curtailment first
        for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
            combustion_ptr = combustion_ptr_vec_ptr->at(i);
            curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

            combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }

        //  2. attempt to charge from Noncombustion curtailment second
        for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
            noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
            curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

            noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }

        //  3. attempt to charge from Renewable curtailment third
        for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
            renewable_ptr = renewable_ptr_vec_ptr->at(i);
            curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

```

```
            renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }

        //  4. commit charge
        storage_ptr->commitCharge(
            timestep,
            dt_hrs,
            storage_ptr->power_kW
        );
    }

    return;
}   /* __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

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

```
{
    double acceptable_kW = 0;
    double curtailment_kW = 0;

    Storage* storage_ptr;
    Combustion* combustion_ptr;
    Noncombustion* noncombustion_ptr;
    Renewable* renewable_ptr;

    for (size_t j = 0; j < storage_ptr_vec_ptr->size(); j++) {
        storage_ptr = storage_ptr_vec_ptr->at(j);

        //  1. attempt to charge from Combustion curtailment first
        for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
            combustion_ptr = combustion_ptr_vec_ptr->at(i);
            curtailment_kW = combustion_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

            combustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            combustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }
```

```

        //  2. attempt to charge from Noncombustion curtailment second
        for (size_t i = 0; i < noncombustion_ptr_vec_ptr->size(); i++) {
            noncombustion_ptr = noncombustion_ptr_vec_ptr->at(i);
            curtailment_kW = noncombustion_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

            noncombustion_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            noncombustion_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }

        //  3. attempt to charge from Renewable curtailment third
        for (size_t i = 0; i < renewable_ptr_vec_ptr->size(); i++) {
            renewable_ptr = renewable_ptr_vec_ptr->at(i);
            curtailment_kW = renewable_ptr->curtailment_vec_kW[timestep];

            if (curtailment_kW <= 0) {
                continue;
            }

            acceptable_kW = storage_ptr->getAcceptablekW(dt_hrs);

            if (acceptable_kW > curtailment_kW) {
                acceptable_kW = curtailment_kW;
            }

            renewable_ptr->curtailment_vec_kW[timestep] -= acceptable_kW;
            renewable_ptr->storage_vec_kW[timestep] += acceptable_kW;
            storage_ptr->power_kW += acceptable_kW;
        }

        //  4. commit charge
        storage_ptr->commitCharge(
            timestep,
            dt_hrs,
            storage_ptr->power_kW
        );
    }

    return;
}   /* __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

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
storage_ptr_list	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.

```
{
    double discharging_kW = 0;

    Storage* storage_ptr;

    std::list<Storage*>::iterator iter;
    for (
        iter = storage_ptr_list.begin();
        iter != storage_ptr_list.end();
        iter++
    ){
        storage_ptr = (*iter);

        discharging_kW = storage_ptr->getAvailablekW(dt_hrs);

        if (discharging_kW > net_load_kW) {
            discharging_kW = net_load_kW;
        }

        net_load_kW = storage_ptr->commitDischarge(
            timestep,
            dt_hrs,
            discharging_kW,
            net_load_kW
        );
    }

    return net_load_kW;
}   /* __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

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

```
{
    for (int i = 0; i < electrical_load_ptr->n_points; i++) {
        switch (this->control_mode) {
            case (ControlMode :: LOAD_FOLLOWING): {
                if (this->net_load_vec_kW[i] <= 0) {
                    this->__applyLoadFollowingControl_CHARGING(
                        i,
                        electrical_load_ptr,
                        resources_ptr,
                        combustion_ptr_vec_ptr,
                        noncombustion_ptr_vec_ptr,
                        renewable_ptr_vec_ptr,
                        storage_ptr_vec_ptr
```

```
                );
            }

            else {
                this->__applyLoadFollowingControl_DISCHARGING(
                    i,
                    electrical_load_ptr,
                    resources_ptr,
                    combustion_ptr_vec_ptr,
                    noncombustion_ptr_vec_ptr,
                    renewable_ptr_vec_ptr,
                    storage_ptr_vec_ptr
                );
            }

            break;
        }

        case (ControlMode :: CYCLE_CHARGING): {
            if (this->net_load_vec_kW[i] <= 0) {
                this->__applyCycleChargingControl_CHARGING(
                    i,
                    electrical_load_ptr,
                    resources_ptr,
                    combustion_ptr_vec_ptr,
                    noncombustion_ptr_vec_ptr,
                    renewable_ptr_vec_ptr,
                    storage_ptr_vec_ptr
                );
            }

            else {
                this->__applyCycleChargingControl_DISCHARGING(
                    i,
                    electrical_load_ptr,
                    resources_ptr,
                    combustion_ptr_vec_ptr,
                    noncombustion_ptr_vec_ptr,
                    renewable_ptr_vec_ptr,
                    storage_ptr_vec_ptr
                );
            }

            break;
        }

        default: {
            std::string error_str = "ERROR:  Controller :: applyDispatchControl():  ";
            error_str += "control mode ";
            error_str += std::to_string(this->control_mode);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }
}

return;
}   /* applyDispatchControl() */
```

4.3.3.14 clear()

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

Method to clear all attributes of the Controller object.

```
{
    this->net_load_vec_kW.clear();
    this->missed_load_vec_kW.clear();
    this->combustion_map.clear();

    return;
}   /* 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

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

```
{
    //  1. compute net load
    this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);

    //  2. construct Combustion table
    this->__constructCombustionMap(combustion_ptr_vec_ptr);

    return;
}   /* init() */
```

4.3.3.16 setControlMode()

```
void Controller::setControlMode (
            ControlMode control_mode )
```

Parameters

control_mode	The ControlMode which is to be active in the Controller.

```
{
    this->control_mode = control_mode;

    switch(control_mode) {
        case (ControlMode :: LOAD_FOLLOWING): {
            this->control_string = "LOAD_FOLLOWING";

            break;
        }

        case (ControlMode :: CYCLE_CHARGING): {
            this->control_string = "CYCLE_CHARGING";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Controller :: setControlMode():  ";
                error_str += "control mode ";
                error_str += std::to_string(control_mode);
                error_str += " not recognized";

                #ifdef _WIN32
                    std::cout « error_str « std::endl;
                #endif

                throw std::runtime_error(error_str);

            break;
```

```
        }
    }

    return;
}   /* setControlMode() */
```

4.3.4 Member Data Documentation

4.3.4.1 combustion_map

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

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

4.3.4.2 control_mode

`ControlMode Controller::control_mode`

The ControlMode that is active in the Model.

4.3.4.3 control_string

`std::string Controller::control_string`

A string describing the active ControlMode.

4.3.4.4 missed_load_vec_kW

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

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

4.3.4.5 net_load_vec_kW

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

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

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

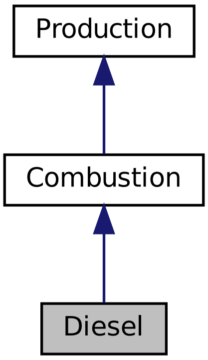
- header/Controller.h
- source/Controller.cpp

4.4 Diesel Class Reference

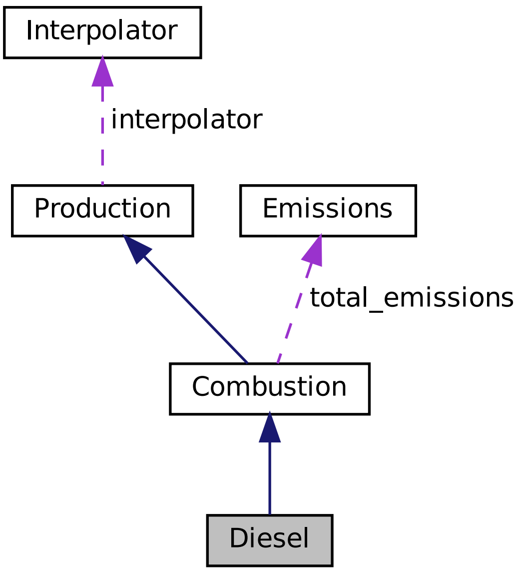
A derived class of the Combustion branch of Production which models production using a diesel generator.

`#include <Diesel.h>`

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

- Diesel (void)

  Constructor (dummy) for the Diesel class.
- Diesel (int, double, DieselInputs, std::vector$<$ double $>$ $*$)

  Constructor (intended) for the Diesel class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double requestProductionkW (int, double, double)

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

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

```
{
    return;
}   /* Diesel() */
```

4.4.2.2 Diesel() [2/2]

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

Constructor (intended) for the Diesel class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
diesel_inputs	A structure of Diesel constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Combustion(
    n_points,
    n_years,
    diesel_inputs.combustion_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(diesel_inputs);

    //  2. set attributes
    this->type = CombustionType :: DIESEL;
    this->type_str = "DIESEL";

    this->replace_running_hrs = diesel_inputs.replace_running_hrs;

    this->fuel_cost_L = diesel_inputs.fuel_cost_L;

    this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
    this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
    this->time_since_last_start_hrs = 0;
```

```
    this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
    this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
    this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
    this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
    this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
    this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;

    if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
        this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
    }
    else {
        this->linear_fuel_slope_LkWh = diesel_inputs.linear_fuel_slope_LkWh;
    }

    if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
        this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
    }
    else {
        this->linear_fuel_intercept_LkWh = diesel_inputs.linear_fuel_intercept_LkWh;
    }

    if (diesel_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = diesel_inputs.capital_cost;
    }

    if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            diesel_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Diesel object constructed at " « this « std::endl;
    }

    return;
}   /* Diesel() */
```

4.4.2.3 ∼Diesel()

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

Destructor for the Diesel class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Diesel object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

diesel_inputs	A structure of Diesel constructor inputs.

```
{
    //  1. check fuel_cost_L
    if (diesel_inputs.fuel_cost_L < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::fuel_cost_L must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check CO2_emissions_intensity_kgL
    if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check CO_emissions_intensity_kgL
        if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check NOx_emissions_intensity_kgL
    if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  5. check SOx_emissions_intensity_kgL
    if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  6. check CH4_emissions_intensity_kgL
    if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  7. check PM_emissions_intensity_kgL
    if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

```

```
        throw std::invalid_argument(error_str);
    }

    //  8. check minimum_load_ratio
    if (diesel_inputs.minimum_load_ratio < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::minimum_load_ratio must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  9. check minimum_runtime_hrs
    if (diesel_inputs.minimum_runtime_hrs < 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  10. check replace_running_hrs
    if (diesel_inputs.replace_running_hrs <= 0) {
        std::string error_str = "ERROR:  Diesel():  ";
        error_str += "DieselInputs::replace_running_hrs must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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].

```
{
    double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.425) + 800;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __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].

```
{
    double linear_fuel_intercept_LkWh = 0.0940 * pow(this->capacity_kW, -0.2735);

    return linear_fuel_intercept_LkWh;
}   /* __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].

```
{
    double linear_fuel_slope_LkWh = 0.4234 * pow(this->capacity_kW, -0.1012);

    return linear_fuel_slope_LkWh;
}   /* __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].

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __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

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

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

    if (this->is_running) {
        // handle stopping
        if (
            production_kW <= 0 and
            this->time_since_last_start_hrs >= this->minimum_runtime_hrs
        ) {
            this->is_running = false;
        }
    }

    else {
        // handle starting
        if (production_kW > 0) {
            this->is_running = true;
            this->n_starts++;
            this->time_since_last_start_hrs = 0;
        }
    }

```

```
    return;
} /* __handleStartStop() */
```

4.4.3.7 __writeSummary()

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

Helper method to write summary results for Diesel.

Parameters

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

Reimplemented from Combustion.

```
{
    // 1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    // 2. write to summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW DIESEL Summary Results\n";
    ofs « "\n--------\n\n";

    // 2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « " kW  \n";
    ofs « "\n";

    ofs « "Production Override: (N = 0 / Y = 1): "
        « this->normalized_production_series_given « "  \n";
    if (this->normalized_production_series_given) {
        ofs « "Path to Normalized Production Time Series: "
            « this->path_2_normalized_production_time_series « "  \n";
    }
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    // 2.2. Combustion attributes
    ofs « "## Combustion Attributes\n";
    ofs « "\n";

    ofs « "Fuel Cost: " « this->fuel_cost_L « " per L  \n";
    ofs « "Nominal Fuel Escalation Rate (annual): "
        « this->nominal_fuel_escalation_annual « "  \n";
    ofs « "Real Fuel Escalation Rate (annual): "
        « this->real_fuel_escalation_annual « "  \n";
    ofs « "\n";

    ofs « "Fuel Mode: " « this->fuel_mode_str « "  \n";
    switch (this->fuel_mode) {
        case (FuelMode :: FUEL_MODE_LINEAR): {
            ofs « "Linear Fuel Slope: " « this->linear_fuel_slope_LkWh
```

```
                « " L/kWh  \n";
            ofs « "Linear Fuel Intercept Coefficient: "
                « this->linear_fuel_intercept_LkWh « " L/kWh  \n";
            ofs « "\n";

            break;
        }

        case (FuelMode :: FUEL_MODE_LOOKUP): {
            ofs « "Fuel Consumption Data: " « this->interpolator.path_map_1D[0]
                « "  \n";

            break;
        }

        default: {
            // write nothing!

            break;
        }
    }

    ofs « "Carbon Dioxide (CO2) Emissions Intensity: "
        « this->CO2_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Carbon Monoxide (CO) Emissions Intensity: "
        « this->CO_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Nitrogen Oxides (NOx) Emissions Intensity: "
        « this->NOx_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Sulfur Oxides (SOx) Emissions Intensity: "
        « this->SOx_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Methane (CH4) Emissions Intensity: "
        « this->CH4_emissions_intensity_kgL « " kg/L  \n";

    ofs « "Particulate Matter (PM) Emissions Intensity: "
        « this->PM_emissions_intensity_kgL « " kg/L  \n";

    ofs « "\n--------\n\n";

    //  2.3. Diesel attributes
    ofs « "## Diesel Attributes\n";
    ofs « "\n";

    ofs « "Minimum Load Ratio: " « this->minimum_load_ratio « "  \n";
    ofs « "Minimum Runtime: " « this->minimum_runtime_hrs « " hrs  \n";

    ofs « "\n--------\n\n";

    //  2.4. Diesel Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Starts: " « this->n_starts « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
        « "(Annual Average: " « this->total_fuel_consumed_L / this->n_years
        « " L/yr)  \n";
    ofs « "\n";

    ofs « "Total Carbon Dioxide (CO2) Emissions: " «
        this->total_emissions.CO2_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.CO2_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Carbon Monoxide (CO) Emissions: " «
        this->total_emissions.CO_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.CO_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
        this->total_emissions.NOx_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.NOx_kg / this->n_years
```

```
        « " kg/yr)  \n";

    ofs « "Total Sulfur Oxides (SOx) Emissions: " «
        this->total_emissions.SOx_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.SOx_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.CH4_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "Total Particulate Matter (PM) Emissions: " «
        this->total_emissions.PM_kg « " kg "
        « "(Annual Average: " «  this->total_emissions.PM_kg / this->n_years
        « " kg/yr)  \n";

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __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

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

Reimplemented from Combustion.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Is Running (N = 0 / Y = 1),";
    ofs « "Fuel Consumption [L],";
    ofs « "Fuel Cost (actual),";
    ofs « "Carbon Dioxide (CO2) Emissions [kg],";
    ofs « "Carbon Monoxide (CO) Emissions [kg],";
    ofs « "Nitrogen Oxides (NOx) Emissions [kg],";
    ofs « "Sulfur Oxides (SOx) Emissions [kg],";
    ofs « "Methane (CH4) Emissions [kg],";
    ofs « "Particulate Matter (PM) Emissions [kg],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
```

```
        ofs « this->is_running_vec[i] « ",";
        ofs « this->fuel_consumption_vec_L[i] « ",";
        ofs « this->fuel_cost_vec[i] « ",";
        ofs « this->CO2_emissions_vec_kg[i] « ",";
        ofs « this->CO_emissions_vec_kg[i] « ",";
        ofs « this->NOx_emissions_vec_kg[i] « ",";
        ofs « this->SOx_emissions_vec_kg[i] « ",";
        ofs « this->CH4_emissions_vec_kg[i] « ",";
        ofs « this->PM_emissions_vec_kg[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __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

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

Returns

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

Reimplemented from Combustion.

```
{
    //  1. handle start/stop, enforce minimum runtime constraint
    this->__handleStartStop(timestep, dt_hrs, production_kW);

    //  2. invoke base class method
    load_kW = Combustion :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );

    if (this->is_running) {
        //  3. log time since last start
        this->time_since_last_start_hrs += dt_hrs;

        //  4. correct operation and maintenance costs (should be non-zero if idling)
        if (production_kW <= 0) {
            double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;

            double operation_maintenance_cost =
                this->operation_maintenance_cost_kWh * produced_kWh;
            this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
        }
```

```
    }

    return load_kW;
}   /* commit() */
```

4.4.3.10 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Combustion.

```
{
    //  1. reset attributes
    this->time_since_last_start_hrs = 0;

    //  2. invoke base class method
    Combustion :: handleReplacement(timestep);

    return;
}   /* __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

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

Returns

The production [kW] delivered by the diesel generator.

Reimplemented from Combustion.

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

```
        return production_kW;
    }

    //  1. return on request of zero
    if (request_kW <= 0) {
        return 0;
    }

    double deliver_kW = request_kW;

    //  2. enforce capacity constraint
    if (deliver_kW > this->capacity_kW) {
        deliver_kW = this->capacity_kW;
    }

    //  3. enforce minimum load ratio
    if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
        deliver_kW = this->minimum_load_ratio * this->capacity_kW;
    }

    return deliver_kW;
}   /* requestProductionkW() */
```

4.4.4 Member Data Documentation

4.4.4.1 minimum_load_ratio

`double Diesel::minimum_load_ratio`

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

4.4.4.2 minimum_runtime_hrs

`double Diesel::minimum_runtime_hrs`

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

4.4.4.3 time_since_last_start_hrs

`double Diesel::time_since_last_start_hrs`

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

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

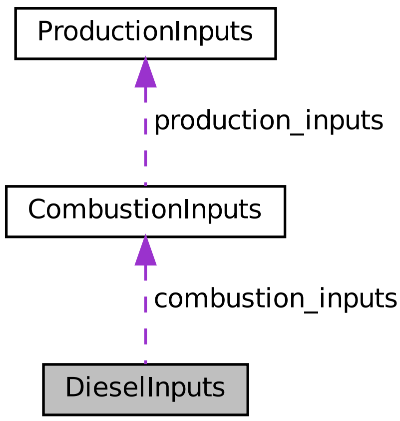
- header/Production/Combustion/Diesel.h
- source/Production/Combustion/Diesel.cpp

4.5 DieselInputs Struct Reference

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

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

Collaboration diagram for DieselInputs:



Public Attributes

- CombustionInputs combustion_inputs

  An encapsulated CombustionInputs instance.
- double replace_running_hrs = 30000

  The number of running hours after which the asset must be replaced. Overwrites the ProductionInputs attribute.
- double capital_cost = -1

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

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double fuel_cost_L = 1.70

  The cost of fuel [1/L] (undefined currency).
- double minimum_load_ratio = 0.2

  The minimum load ratio of the asset. That is, when the asset is producing, it must produce at least this ratio of its rated capacity.
- double minimum_runtime_hrs = 4

  The minimum runtime [hrs] of the asset. This is the minimum time that must elapse between successive starts and stops.
- double linear_fuel_slope_LkWh = -1

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

- double linear_fuel_intercept_LkWh = -1

  The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced. -1 is a sentinel value, which triggers a generic fuel consumption model on construction (in fact, any negative value here will trigger).
- double CO2_emissions_intensity_kgL = 2.7

  Carbon dioxide (CO2) emissions intensity [kg/L].
- double CO_emissions_intensity_kgL = 0.0178

  Carbon monoxide (CO) emissions intensity [kg/L].
- double NOx_emissions_intensity_kgL = 0.0014

  Nitrogen oxide (NOx) emissions intensity [kg/L].
- double SOx_emissions_intensity_kgL = 0.0042

  Sulfur oxide (SOx) emissions intensity [kg/L].
- double CH4_emissions_intensity_kgL = 0.0007

  Methane (CH4) emissions intensity [kg/L].
- double PM_emissions_intensity_kgL = 0.0001

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

4.5.1 Detailed Description

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

Ref: HOMER [2023c]
Ref: HOMER [2023d]
Ref: HOMER [2023e]
Ref: NRCan [2014]
Ref: CIMAC [2008]

4.5.2 Member Data Documentation

4.5.2.1 capital_cost

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

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

4.5.2.2 CH4_emissions_intensity_kgL

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

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

4.5.2.3 CO2_emissions_intensity_kgL

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

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

4.5.2.4 CO_emissions_intensity_kgL

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

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

4.5.2.5 combustion_inputs

```
CombustionInputs DieselInputs::combustion_inputs
```

An encapsulated CombustionInputs instance.

4.5.2.6 fuel_cost_L

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

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

4.5.2.7 linear_fuel_intercept_LkWh

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

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

4.5.2.8 linear_fuel_slope_LkWh

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

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

4.5.2.9 minimum_load_ratio

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

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

4.5.2.10 minimum_runtime_hrs

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

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

4.5.2.11 NOx_emissions_intensity_kgL

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

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

4.5.2.12 operation_maintenance_cost_kWh

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

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

4.5.2.13 PM_emissions_intensity_kgL

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

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

4.5.2.14 replace_running_hrs

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

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

4.5.2.15 SOx_emissions_intensity_kgL

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

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

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

- header/Production/Combustion/Diesel.h

4.6 ElectricalLoad Class Reference

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

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

Public Member Functions

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

Public Attributes

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

4.6.1 Detailed Description

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

4.6.2 Constructor & Destructor Documentation

4.6.2.1 ElectricalLoad() [1/2]

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

Constructor (dummy) for the ElectricalLoad class.

```
{
    return;
}   /* 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

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

```
{
    this->readLoadData(path_2_electrical_load_time_series);

    return;
}   /* ElectricalLoad() */
```

4.6.2.3 ∼ElectricalLoad()

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

Destructor for the ElectricalLoad class.

```
{
    this->clear();
    return;
}   /* ~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.

```
{
    this->n_points = 0;
    this->n_years = 0;
    this->min_load_kW = 0;
    this->mean_load_kW = 0;
    this->max_load_kW = 0;

    this->path_2_electrical_load_time_series.clear();
    this->time_vec_hrs.clear();
    this->dt_vec_hrs.clear();
    this->load_vec_kW.clear();

    return;
}   /* 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

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

```
{
    //  1. clear
    this->clear();

    //  2. init CSV reader, record path
    io::CSVReader<2> CSV(path_2_electrical_load_time_series);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Electrical Load [kW]"
    );

    this->path_2_electrical_load_time_series = path_2_electrical_load_time_series;

    //  3. read in time and load data, increment n_points, track min and max load
    double time_hrs = 0;
    double load_kW = 0;
    double load_sum_kW = 0;

    this->n_points = 0;

    this->min_load_kW = std::numeric_limits<double>::infinity();
    this->max_load_kW = -1 * std::numeric_limits<double>::infinity();

    while (CSV.read_row(time_hrs, load_kW)) {
        this->time_vec_hrs.push_back(time_hrs);
        this->load_vec_kW.push_back(load_kW);

        load_sum_kW += load_kW;

        this->n_points++;

        if (this->min_load_kW > load_kW) {
            this->min_load_kW = load_kW;
        }

```

```
        if (this->max_load_kW < load_kW) {
            this->max_load_kW = load_kW;
        }
    }

    //  4. compute mean load
    this->mean_load_kW = load_sum_kW / this->n_points;

    //  5. set number of years (assuming 8,760 hours per year)
    this->n_years = this->time_vec_hrs[this->n_points - 1] / 8760;

    //  6. populate dt_vec_hrs
    this->dt_vec_hrs.resize(n_points, 0);

    for (int i = 0; i < n_points; i++) {
        if (i == n_points - 1) {
            this->dt_vec_hrs[i] = this->dt_vec_hrs[i - 1];
        }

        else {
            double dt_hrs = this->time_vec_hrs[i + 1] - this->time_vec_hrs[i];

            this->dt_vec_hrs[i] = dt_hrs;
        }
    }

    return;
}   /* readLoadData() */
```

4.6.4 Member Data Documentation

4.6.4.1 dt_vec_hrs

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

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

4.6.4.2 load_vec_kW

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

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

4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

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

4.6.4.4 mean_load_kW

```
double ElectricalLoad::mean_load_kW
```

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

4.6.4.5 min_load_kW

```
double ElectricalLoad::min_load_kW
```

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

4.6.4.6 n_points

```
int ElectricalLoad::n_points
```

The number of points in the modelling time series.

4.6.4.7 n_years

```
double ElectricalLoad::n_years
```

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

4.6.4.8 path_2_electrical_load_time_series

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

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

4.6.4.9 time_vec_hrs

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

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

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

- header/ElectricalLoad.h
- source/ElectricalLoad.cpp

4.7 Emissions Struct Reference

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

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

Public Attributes

- double CO2_kg = 0

    The mass of carbon dioxide (CO2) emitted [kg].
- double CO_kg = 0

    The mass of carbon monoxide (CO) emitted [kg].
- double NOx_kg = 0

    The mass of nitrogen oxides (NOx) emitted [kg].
- double SOx_kg = 0

    The mass of sulfur oxides (SOx) emitted [kg].
- double CH4_kg = 0

    The mass of methane (CH4) emitted [kg].
- double PM_kg = 0

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

4.7.1 Detailed Description

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

4.7.2 Member Data Documentation

4.7.2.1 CH4_kg

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

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

4.7.2.2 CO2_kg

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

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

4.7.2.3 CO_kg

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

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

4.7.2.4 NOx_kg

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

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

4.7.2.5 PM_kg

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

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

4.7.2.6 SOx_kg

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

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

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

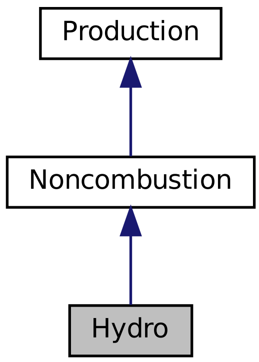
- header/Production/Combustion/Combustion.h

4.8 Hydro Class Reference

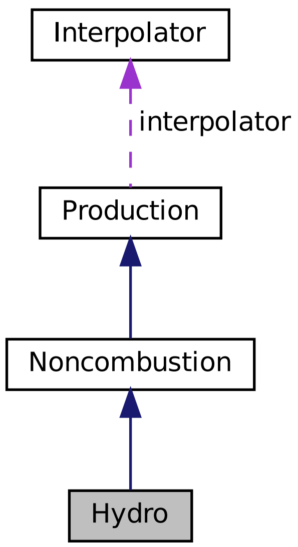
A derived class of the Noncombustion branch of Production which models production using a hydroelectric asset (either with reservoir or not).

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

Inheritance diagram for Hydro:



Collaboration diagram for Hydro:



Public Member Functions

- Hydro (void)

  Constructor (dummy) for the Hydro class.
- Hydro (int, double, HydroInputs, std::vector< double > *)

  Constructor (intended) for the Hydro class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double requestProductionkW (int, double, double, double)

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

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ∼Hydro (void)

  Destructor for the Hydro class.

Public Attributes

- HydroTurbineType turbine_type

  The type of hydroelectric turbine model to use.
- double fluid_density_kgm3

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

  The net head [m] of the asset.
- double reservoir_capacity_m3

  The capacity [m3] of the hydro reservoir.
- double init_reservoir_state

  The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- double stored_volume_m3

  The volume [m3] of stored fluid.
- double minimum_power_kW

  The minimum power [kW] that the asset can produce. Corresponds to minimum productive flow.
- double minimum_flow_m3hr

  The minimum required flow [m3/hr] for the asset to produce. Corresponds to minimum power.
- double maximum_flow_m3hr

  The maximum productive flow [m3/hr] that the asset can support.
- std::vector< double > turbine_flow_vec_m3hr

  A vector of the turbine flow [m3/hr] at each point in the modelling time series.
- std::vector< double > spill_rate_vec_m3hr

  A vector of the spill rate [m3/hr] at each point in the modelling time series.
- std::vector< double > stored_volume_vec_m3

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

Private Member Functions

- void __checkInputs (HydroInputs)

  Helper method to check inputs to the Hydro constructor.
- void __initInterpolator (void)

  Helper method to set up turbine and generator efficiency interpolation.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic hydroelectric capital cost.
- double __getGenericOpMaintCost (void)

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

  Helper method to compute the efficiency factor (product of turbine and generator efficiencies).
- double __getMinimumFlowm3hr (void)

  Helper method to compute and return the minimum required flow for production, based on turbine type.
- double __getMaximumFlowm3hr (void)

  Helper method to compute and return the maximum productive flow, based on turbine type.
- double __flowToPower (double)

  Helper method to translate a given flow into a corresponding power output.
- double __powerToFlow (double)

  Helper method to translate a given power output into a corresponding flow.
- double __getAvailableFlow (double, double)

  Helper method to determine what flow is currently available to the turbine.
- double __getAcceptableFlow (double)

  Helper method to determine what flow is currently acceptable by the reservoir.
- void __updateState (int, double, double, double)

  Helper method to update and log flow and reservoir state.
- void __writeSummary (std::string)

  Helper method to write summary results for Hydro.
- void __writeTimeSeries (std::string, std::vector$<$ double $>$ $*$, int=-1)

  Helper method to write time series results for Hydro.

4.8.1 Detailed Description

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

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Hydro() [1/2]

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

Constructor (dummy) for the Hydro class.

```
{
    return;
}   /* Hydro() */
```

4.8.2.2 Hydro() [2/2]

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

Constructor (intended) for the Hydro class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
hydro_inputs	A structure of Hydro constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Noncombustion(
    n_points,
    n_years,
    hydro_inputs.noncombustion_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(hydro_inputs);

    //  2. set attributes
    this->type = NoncombustionType :: HYDRO;
    this->type_str = "HYDRO";

    this->resource_key = hydro_inputs.resource_key;

    this->turbine_type = hydro_inputs.turbine_type;

    this->fluid_density_kgm3 = hydro_inputs.fluid_density_kgm3;
    this->net_head_m = hydro_inputs.net_head_m;

    this->reservoir_capacity_m3 = hydro_inputs.reservoir_capacity_m3;
    this->init_reservoir_state = hydro_inputs.init_reservoir_state;
    this->stored_volume_m3 =
        hydro_inputs.init_reservoir_state * hydro_inputs.reservoir_capacity_m3;

    this->minimum_power_kW = 0.1 * this->capacity_kW;   // <-- NEED TO DOUBLE CHECK THAT THIS MAKES
      SENSE IN GENERAL

    this->__initInterpolator();

    this->minimum_flow_m3hr = this->__getMinimumFlowm3hr();
    this->maximum_flow_m3hr = this->__getMaximumFlowm3hr();

    this->turbine_flow_vec_m3hr.resize(this->n_points, 0);
    this->spill_rate_vec_m3hr.resize(this->n_points, 0);
    this->stored_volume_vec_m3.resize(this->n_points, 0);

    if (hydro_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = hydro_inputs.capital_cost;
    }

    if (hydro_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            hydro_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    return;
}   /* Hydro() */
```

4.8.2.3 ∼Hydro()

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

Destructor for the Hydro class.

```
{
    // 1. destruction print
    if (this->print_flag) {
        std::cout « "Hydro object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

hydro_inputs	A structure of Hydro constructor inputs.

```
{
    // 1. check fluid_density_kgm3
    if (hydro_inputs.fluid_density_kgm3 <= 0) {
        std::string error_str = "ERROR: Hydro(): fluid_density_kgm3 must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    // 2. check net_head_m
    if (hydro_inputs.net_head_m <= 0) {
        std::string error_str = "ERROR: Hydro(): net_head_m must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    // 3. check reservoir_capacity_m3
    if (hydro_inputs.reservoir_capacity_m3 < 0) {
        std::string error_str = "ERROR: Hydro(): reservoir_capacity_m3 must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

```

```
    //  4. check init_reservoir_state
    if (
        hydro_inputs.init_reservoir_state < 0 or
        hydro_inputs.init_reservoir_state > 1
    ) {
        std::string error_str = "ERROR:  Hydro():  init_reservoir_state must be in ";
        error_str += "the closed interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.8.3.2 __flowToPower()

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

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

Ref: Truelove [2023b]

Parameters

flow_m3hr	The flow [m3/hr] through the turbine.

Returns

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

```
{
    //  1. return on less than minimum flow
    if (flow_m3hr < this->minimum_flow_m3hr) {
        return 0;
    }

    //  2. interpolate flow to power
    double power_kW = this->interpolator.interp1D(
        HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
        flow_m3hr
    );

    return power_kW;
}   /* __flowToPower() */
```

4.8.3.3 __getAcceptableFlow()

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

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

Parameters

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

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

```
{
    //  1. if no reservoir, return
    if (this->reservoir_capacity_m3 <= 0) {
        return 0;
    }

    //  2. compute acceptable based on room in reservoir
    double acceptable_m3hr = (this->reservoir_capacity_m3 - this->stored_volume_m3) /
        dt_hrs;

    return acceptable_m3hr;
}   /* __getAcceptableFlow() */
```

4.8.3.4 __getAvailableFlow()

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

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

Parameters

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

Returns

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

```
{
    //  1. init to flow available from stored volume in reservoir
    double flow_m3hr = this->stored_volume_m3 / dt_hrs;

    //  2. add flow available from resource
    flow_m3hr += hydro_resource_m3hr;

    //  3. cap at maximum flow
    if (flow_m3hr > this->maximum_flow_m3hr) {
        flow_m3hr = this->maximum_flow_m3hr;
    }

    return flow_m3hr;
}   /* __getAvailableFlow() */
```

4.8.3.5 __getEfficiencyFactor()

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

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

Ref: Truelove [2023b]

Parameters

power_kW	The power requested of the hydro plant.

Returns

The product of the turbine and generator efficiencies.

```
{
    //  1. return on zero
    if (power_kW <= 0) {
        return 0;
    }

    //  2. compute power ratio (clip to [0, 1])
    double power_ratio = power_kW / this->capacity_kW;

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

    else if (power_ratio > 1) {
        power_ratio = 1;
    }


    //  3. init efficiency factor to the turbine efficiency
    double efficiency_factor = this->interpolator.interp1D(
        HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
        power_ratio
    );

    //  4. include generator efficiency
    efficiency_factor *= this->interpolator.interp1D(
        HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
        power_ratio
    );

    return efficiency_factor;
}   /* __getEfficiencyFactor() */
```

4.8.3.6 __getGenericCapitalCost()

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

Helper method to generate a generic hydroelectric capital cost.

This model was obtained by way of ...

Returns

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

```
{
    double capital_cost_per_kW = 1000; //<-- WIP: need something better here!

    return capital_cost_per_kW * this->capacity_kW + 15000000; //<-- WIP: need something better here!
}   /* __getGenericCapitalCost() */
```

4.8.3.7 __getGenericOpMaintCost()

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

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

This model was obtained by way of ...

Returns

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

```
{
    double operation_maintenance_cost_kWh = 0.05;  //<-- WIP: need something better here!

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

4.8.3.8 __getMaximumFlowm3hr()

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

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

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

Ref: Truelove [2023b]

Returns

The maximum productive flow [m3/hr].

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

4.8.3.9 __getMinimumFlowm3hr()

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

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

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

Ref: Truelove [2023b]

Returns

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

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

4.8.3.10 __initInterpolator()

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

Helper method to set up turbine and generator efficiency interpolation.

Ref: Truelove [2023b]

```
{
    //  1. set up generator efficiency interpolation
    InterpolatorStruct1D generator_interp_struct_1D;

    generator_interp_struct_1D.n_points = 12;

    generator_interp_struct_1D.x_vec = {
        0,    0.1, 0.2,  0.3, 0.4, 0.5,
        0.6, 0.7, 0.75, 0.8, 0.9, 1
    };

    generator_interp_struct_1D.min_x = 0;
    generator_interp_struct_1D.max_x = 1;

    generator_interp_struct_1D.y_vec = {
        0.000, 0.800, 0.900, 0.913,
        0.925, 0.943, 0.947, 0.950,
        0.953, 0.954, 0.956, 0.958
    };

    this->interpolator.interp_map_1D.insert(
        std::pair<int, InterpolatorStruct1D>(
            HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
            generator_interp_struct_1D
        )
    );

    //  2. set up turbine efficiency interpolation
    InterpolatorStruct1D turbine_interp_struct_1D;

    turbine_interp_struct_1D.n_points = 11;

    turbine_interp_struct_1D.x_vec = {
        0,   0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9,
        1
    };

    turbine_interp_struct_1D.min_x = 0;
    turbine_interp_struct_1D.max_x = 1;

    std::vector<double> efficiency_vec;

    switch (this->turbine_type) {
        case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
            efficiency_vec = {
                0.000, 0.780, 0.855, 0.875, 0.890,
                0.900, 0.908, 0.913, 0.918, 0.908,
                0.880
            };

            break;
        }

        case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
            efficiency_vec = {
                0.000, 0.400, 0.625, 0.745, 0.810,
                0.845, 0.880, 0.900, 0.910, 0.900,
                0.850
            };

            break;
        }

        case(HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
            efficiency_vec = {
                0.000, 0.265, 0.460, 0.550, 0.650,
                0.740, 0.805, 0.845, 0.900, 0.880,
                0.850
            };

            break;
        }
```

```

        default: {
            std::string error_str = "ERROR:  Hydro():  turbine type ";
            error_str += std::to_string(this->turbine_type);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    turbine_interp_struct_1D.y_vec = efficiency_vec;

    this->interpolator.interp_map_1D.insert(
        std::pair<int, InterpolatorStruct1D>(
            HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
            turbine_interp_struct_1D
        )
    );

    //  3. set up flow to power interpolation
    InterpolatorStruct1D flow_to_power_interp_struct_1D;

    double power_ratio = 0.1;
    std::vector<double> power_ratio_vec (91, 0);

    for (size_t i = 0; i < power_ratio_vec.size(); i++) {
        power_ratio_vec[i] = power_ratio;

        power_ratio += 0.01;

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

        else if (power_ratio > 1) {
            power_ratio = 1;
        }
    }

    flow_to_power_interp_struct_1D.n_points = power_ratio_vec.size();

    std::vector<double> flow_vec_m3hr;
    std::vector<double> power_vec_kW;
    flow_vec_m3hr.resize(power_ratio_vec.size(), 0);
    power_vec_kW.resize(power_ratio_vec.size(), 0);

    for (size_t i = 0; i < power_ratio_vec.size(); i++) {
        flow_vec_m3hr[i] = this->__powerToFlow(power_ratio_vec[i] * this->capacity_kW);
        power_vec_kW[i] = power_ratio_vec[i] * this->capacity_kW;
        /*
        std::cout « flow_vec_m3hr[i] « "\t" « power_vec_kW[i] « " (" «
            power_ratio_vec[i] « ")" « std::endl;
        */
    }

    flow_to_power_interp_struct_1D.x_vec = flow_vec_m3hr;

    flow_to_power_interp_struct_1D.min_x = flow_vec_m3hr[0];
    flow_to_power_interp_struct_1D.max_x = flow_vec_m3hr[flow_vec_m3hr.size() - 1];

    flow_to_power_interp_struct_1D.y_vec = power_vec_kW;

    this->interpolator.interp_map_1D.insert(
        std::pair<int, InterpolatorStruct1D>(
            HydroInterpKeys :: FLOW_TO_POWER_INTERP_KEY,
            flow_to_power_interp_struct_1D
        )
    );

    return;
}   /* __initInterpolator() */
```

4.8.3.11 __powerToFlow()

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

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

Ref: Truelove [2023b]

Parameters

power_kW	The power output [kW] of the hydroelectric generator.

Returns

```
{
    //  1. return on zero power
    if (power_kW <= 0) {
        return 0;
    }

    //  2. get efficiency factor
    double efficiency_factor = this->__getEfficiencyFactor(power_kW);

    //  3. compute flow
    double flow_m3hr = 3600 * 1000 * power_kW;
    flow_m3hr /= efficiency_factor * this->fluid_density_kgm3 * 9.81 * this->net_head_m;

    return flow_m3hr;
}   /* __powerToFlow() */
```

4.8.3.12 __updateState()

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

Helper method to update and log flow and reservoir state.

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
production_kW	The production [kW] of the asset in this timestep.
hydro_resource_m3hr	The currently available hydro flow resource [m3/hr].

```
{
    //  1. get turbine flow, log
    double flow_m3hr = 0;

    if (production_kW >= this->minimum_power_kW) {
        flow_m3hr = this->__powerToFlow(production_kW);
    }

    double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);

    if (flow_m3hr > available_flow_m3hr) {
        flow_m3hr = available_flow_m3hr;
    }

    this->turbine_flow_vec_m3hr[timestep] = flow_m3hr;

    //  3. compute net reservoir flow
```

```
    double net_flow_m3hr = hydro_resource_m3hr - flow_m3hr;

    //  4. compute flow acceptable by reservoir
    double acceptable_flow_m3hr = this->__getAcceptableFlow(dt_hrs);

    //  5. compute spill, update net flow (if applicable), log
    double spill_m3hr = 0;

    if (acceptable_flow_m3hr < net_flow_m3hr) {
        spill_m3hr = net_flow_m3hr - acceptable_flow_m3hr;
        net_flow_m3hr = acceptable_flow_m3hr;
    }

    this->spill_rate_vec_m3hr[timestep] = spill_m3hr;

    //  6. update reservoir state, log
    this->stored_volume_m3 += net_flow_m3hr * dt_hrs;
    this->stored_volume_vec_m3[timestep] = this->stored_volume_m3;

    return;
}   /* __updateState() */
```

4.8.3.13 __writeSummary()

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

Helper method to write summary results for Hydro.

Parameters

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

Reimplemented from Noncombustion.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write to summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW HYDRO Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « " kW  \n";
    ofs « "\n";

    ofs « "Production Override: (N = 0 / Y = 1): "
        « this->normalized_production_series_given « "  \n";
    if (this->normalized_production_series_given) {
        ofs « "Path to Normalized Production Time Series: "
            « this->path_2_normalized_production_time_series « "  \n";
    }
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";
```

```
    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Noncombustion attributes
    ofs « "## Noncombustion Attributes\n";
    ofs « "\n";

    //...

    ofs « "\n--------\n\n";

    //  2.3. Hydro attributes
    ofs « "## Hydro Attributes\n";
    ofs « "\n";

    ofs « "Fluid Density: " « this->fluid_density_kgm3 « " kg/m3  \n";
    ofs « "Net Head: " « this->net_head_m « " m  \n";
    ofs « "\n";

    ofs « "Reservoir Volume: " « this->reservoir_capacity_m3 « " m3  \n";
    ofs « "Reservoir Initial State: " « this->init_reservoir_state « "  \n";
    ofs « "\n";

    ofs « "Turbine Type: ";
    switch(this->turbine_type) {
        case(HydroTurbineType :: HYDRO_TURBINE_PELTON): {
            ofs « "PELTON";

            break;
        }

        case(HydroTurbineType :: HYDRO_TURBINE_FRANCIS): {
            ofs « "FRANCIS";

            break;
        }

        case(HydroTurbineType :: HYDRO_TURBINE_KAPLAN): {
            ofs « "KAPLAN";

            break;
        }

        default: {
            // write nothing!

            break;
        }
    }
    ofs « "  \n";
    ofs « "\n";
    ofs « "Minimum Flow: " « this->minimum_flow_m3hr « " m3/hr  \n";
    ofs « "Maximum Flow: " « this->maximum_flow_m3hr « " m3/hr  \n";
    ofs « "\n";
    ofs « "Minimum Production: " « this->minimum_power_kW « " kW  \n";
    ofs « "\n";

    ofs « "\n--------\n\n";

    //  2.4. Hydro Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    //...

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __writeSummary() */
```

4.8.3.14 __writeTimeSeries()

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

Helper method to write time series results for Hydro.

Parameters

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

Reimplemented from Noncombustion.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Is Running (N = 0 / Y = 1),";
    ofs « "Turbine Flow [m3/hr],";
    ofs « "Spill Rate [m3/hr],";
    ofs « "Stored Volume [m3],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->is_running_vec[i] « ",";
        ofs « this->turbine_flow_vec_m3hr[i] « ",";
        ofs « this->spill_rate_vec_m3hr[i] « ",";
        ofs « this->stored_volume_vec_m3[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

4.8.3.15 commit()

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

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

Parameters

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

Returns

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

Reimplemented from Noncombustion.

```
{
    //  1. invoke base class method
    load_kW = Noncombustion :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );

    //  2. update state and record
    this->__updateState(
        timestep,
        dt_hrs,
        production_kW,
        hydro_resource_m3hr
    );

    return load_kW;
}   /* commit() */
```

4.8.3.16 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Noncombustion.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Noncombustion :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.8.3.17 requestProductionkW()

```
double Hydro::requestProductionkW (
            int timestep,
```

```
double dt_hrs,
double request_kW,
double hydro_resource_m3hr )  [virtual]
```

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

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
request_kW	The requested production [kW].
hydro_resource_m3hr	The currently available hydro flow resource [m3/hr].

Returns

The production [kW] delivered by the hydro generator.

Reimplemented from Noncombustion.

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

        return production_kW;
    }

    //  1. return on request of zero
    if (request_kW <= 0) {
        return 0;
    }

    //  2. if request is less than minimum power, set to minimum power
    if (request_kW < this->minimum_power_kW) {
        request_kW = this->minimum_power_kW;
    }

    //  3. check available flow, return if less than minimum flow
    double available_flow_m3hr = this->__getAvailableFlow(dt_hrs, hydro_resource_m3hr);

    if (available_flow_m3hr < this->minimum_flow_m3hr) {
        return 0;
    }

    //  4. init production to request, enforce capacity constraint (which also accounts
    //     for maximum flow constraint).
    double production_kW = request_kW;

    if (production_kW > this->capacity_kW) {
        production_kW = this->capacity_kW;
    }

    //  5. map production to flow
    double flow_m3hr = this->__powerToFlow(production_kW);

    //  6. if flow is in excess of available, then adjust production accordingly
    if (flow_m3hr > available_flow_m3hr) {
        production_kW = this->__flowToPower(available_flow_m3hr);
    }

    return production_kW;
}   /* requestProductionkW() */
```

4.8.4 Member Data Documentation

4.8.4.1 fluid_density_kgm3

```
double Hydro::fluid_density_kgm3
```

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

4.8.4.2 init_reservoir_state

```
double Hydro::init_reservoir_state
```

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

4.8.4.3 maximum_flow_m3hr

```
double Hydro::maximum_flow_m3hr
```

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

4.8.4.4 minimum_flow_m3hr

```
double Hydro::minimum_flow_m3hr
```

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

4.8.4.5 minimum_power_kW

```
double Hydro::minimum_power_kW
```

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

4.8.4.6 net_head_m

```
double Hydro::net_head_m
```

The net head [m] of the asset.

4.8.4.7 reservoir_capacity_m3

```
double Hydro::reservoir_capacity_m3
```

The capacity [m3] of the hydro reservoir.

4.8.4.8 spill_rate_vec_m3hr

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

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

4.8.4.9 stored_volume_m3

```
double Hydro::stored_volume_m3
```

The volume [m3] of stored fluid.

4.8.4.10 stored_volume_vec_m3

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

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

4.8.4.11 turbine_flow_vec_m3hr

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

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

4.8.4.12 turbine_type

```
HydroTurbineType Hydro::turbine_type
```

The type of hydroelectric turbine model to use.

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

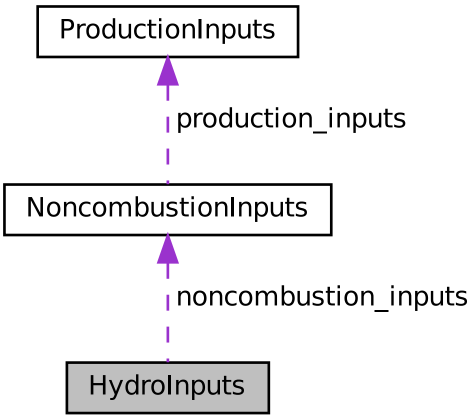
- header/Production/Noncombustion/Hydro.h
- source/Production/Noncombustion/Hydro.cpp

4.9 HydroInputs Struct Reference

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

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

Collaboration diagram for HydroInputs:



Public Attributes

- NoncombustionInputs noncombustion_inputs

  An encapsulated NoncombustionInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

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

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double fluid_density_kgm3 = 1000

  The density [kg/m3] of the hydroelectric working fluid.
- double net_head_m = 500

  The net head [m] of the asset.
- double reservoir_capacity_m3 = 0

  The capacity [m3] of the hydro reservoir.
- double init_reservoir_state = 0

  The initial state of the reservoir (where state is volume of stored fluid divided by capacity).
- HydroTurbineType turbine_type = HydroTurbineType :: HYDRO_TURBINE_PELTON

  The type of hydroelectric turbine model to use.

4.9.1 Detailed Description

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

4.9.2 Member Data Documentation

4.9.2.1 capital_cost

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

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

4.9.2.2 fluid_density_kgm3

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

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

4.9.2.3 init_reservoir_state

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

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

4.9.2.4 net_head_m

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

The net head [m] of the asset.

4.9.2.5 noncombustion_inputs

```
NoncombustionInputs HydroInputs::noncombustion_inputs
```

An encapsulated NoncombustionInputs instance.

4.9.2.6 operation_maintenance_cost_kWh

```
double HydroInputs::operation_maintenance_cost_kWh = -1
```

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

4.9.2.7 reservoir_capacity_m3

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

The capacity [m3] of the hydro reservoir.

4.9.2.8 resource_key

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

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

4.9.2.9 turbine_type

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

The type of hydroelectric turbine model to use.

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

- header/Production/Noncombustion/Hydro.h

4.10 Interpolator Class Reference

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

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

Public Member Functions

- Interpolator (void)

  Constructor for the Interpolator class.
- void addData1D (int, std::string)

  Method to add 1D interpolation data to the Interpolator.
- void addData2D (int, std::string)

  Method to add 2D interpolation data to the Interpolator.
- double interp1D (int, double)

  Method to perform a 1D interpolation.
- double interp2D (int, double, double)

  Method to perform a 2D interpolation.
- ∼Interpolator (void)

  Destructor for the Interpolator class.

Public Attributes

- std::map< int, InterpolatorStruct1D > interp_map_1D

  A map <int, InterpolatorStruct1D> of given 1D interpolation data.
- std::map< int, std::string > path_map_1D

  A map <int, string> of the paths (either relative or absolute) to the given 1D interpolation data.
- std::map< int, InterpolatorStruct2D > interp_map_2D

  A map <int, InterpolatorStruct2D> of given 2D interpolation data.
- std::map< int, std::string > path_map_2D

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

Private Member Functions

- void __checkDataKey1D (int)

  Helper method to check if given data key (1D) is already in use.
- void __checkDataKey2D (int)

  Helper method to check if given data key (2D) is already in use.
- void __checkBounds1D (int, double)

  Helper method to check that the given 1D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void __checkBounds2D (int, double, double)

  Helper method to check that the given 2D interpolation value is contained within the given corresponding data domain. Also checks that the data key has been registered.
- void __throwReadError (std::string, int)

  Helper method to throw a read error whenever non-numeric data is encountered where only numeric data should be.
- bool __isNonNumeric (std::string)

  Helper method to determine if given string is non-numeric (i.e., contains.
- int __getInterpolationIndex (double, std::vector< double > ∗)

  Helper method to get appropriate interpolation index into given vector.
- std::vector< std::string > __splitCommaSeparatedString (std::string, std::string="||")

  Helper method to split a comma-separated string into a vector of substrings.
- std::vector< std::vector< std::string > > __getDataStringMatrix (std::string)
- void __readData1D (int, std::string)

  Helper method to read the given 1D interpolation data into Interpolator.
- void __readData2D (int, std::string)

  Helper method to read the given 2D interpolation data into Interpolator.

4.10.1 Detailed Description

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

4.10.2 Constructor & Destructor Documentation

4.10.2.1 Interpolator()

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

Constructor for the Interpolator class.

```
{
    //...

    return;
}   /* Interpolator() */
```

4.10.2.2 ∼Interpolator()

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

Destructor for the Interpolator class.

```
{
    //...

    return;
}   /* ~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

data_key	A key associated with the given interpolation data.
interp←x	The query value to be interpolated.

```
{
    //  1. key error
    if (this->interp_map_1D.count(data_key) == 0) {
        std::string error_str = "ERROR:  Interpolator::interp1D()  ";
        error_str += "data key ";
        error_str += std::to_string(data_key);
        error_str += " has not been registered";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. bounds error
    if (
        interp_x < this->interp_map_1D[data_key].min_x or
        interp_x > this->interp_map_1D[data_key].max_x
    ) {
        std::string error_str = "ERROR:  Interpolator::interp1D()  ";
        error_str += "interpolation value ";
        error_str += std::to_string(interp_x);
        error_str += " is outside of the given interpolation data domain [";
        error_str += std::to_string(this->interp_map_1D[data_key].min_x);
        error_str += " , ";
        error_str += std::to_string(this->interp_map_1D[data_key].max_x);
        error_str += "]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

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

```
{
    //  1. key error
    if (this->interp_map_2D.count(data_key) == 0) {
        std::string error_str = "ERROR:  Interpolator::interp2D()  ";
        error_str += "data key ";
        error_str += std::to_string(data_key);
        error_str += " has not been registered";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }
```

```

    //  2. bounds error (x_interp)
    if (
        interp_x < this->interp_map_2D[data_key].min_x or
        interp_x > this->interp_map_2D[data_key].max_x
    ) {
        std::string error_str = "ERROR:  Interpolator::interp2D()  ";
        error_str += "interpolation value interp_x = ";
        error_str += std::to_string(interp_x);
        error_str += " is outside of the given interpolation data domain [";
        error_str += std::to_string(this->interp_map_2D[data_key].min_x);
        error_str += " , ";
        error_str += std::to_string(this->interp_map_2D[data_key].max_x);
        error_str += "]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. bounds error (y_interp)
    if (
        interp_y < this->interp_map_2D[data_key].min_y or
        interp_y > this->interp_map_2D[data_key].max_y
    ) {
        std::string error_str = "ERROR:  Interpolator::interp2D()  ";
        error_str += "interpolation value interp_y = ";
        error_str += std::to_string(interp_y);
        error_str += " is outside of the given interpolation data domain [";
        error_str += std::to_string(this->interp_map_2D[data_key].min_y);
        error_str += " , ";
        error_str += std::to_string(this->interp_map_2D[data_key].max_y);
        error_str += "]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

data_key	The key associated with the given 1D interpolation data.

```
{
    if (this->interp_map_1D.count(data_key) > 0) {
        std::string error_str = "ERROR:  Interpolator::addData1D()  ";
        error_str += "data key (1D) ";
        error_str += std::to_string(data_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

data_key	The key associated with the given 2D interpolation data.

```
{
    if (this->interp_map_2D.count(data_key) > 0) {
        std::string error_str = "ERROR:  Interpolator::addData2D()  ";
        error_str += "data key (2D) ";
        error_str += std::to_string(data_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkDataKey2D() */
```

4.10.3.5 __getDataStringMatrix()

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

```
{
    //  1. create input file stream
    std::ifstream ifs;
    ifs.open(path_2_data);

    //  2. check that open() worked
    if (not ifs.is_open()) {
        std::string error_str = "ERROR:  Interpolator::__getDataStringMatrix()  ";
        error_str += " failed to open ";
        error_str += path_2_data;

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. read file line by line
    bool is_header = true;
    std::string line;
    std::vector<std::string> line_split_vec;
    std::vector<std::vector<std::string» string_matrix;

    while (not ifs.eof()) {
        std::getline(ifs, line);

        if (is_header) {
            is_header = false;
            continue;
        }

        line_split_vec = this->__splitCommaSeparatedString(line);

        if (not line_split_vec.empty()) {
            string_matrix.push_back(line_split_vec);
        }
    }

    ifs.close();
    return string_matrix;
}   /* __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

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

Returns

The appropriate interpolation index into the given vector.

```
{
    int idx = 0;
    while (
        not (interp_x >= x_vec_ptr->at(idx) and interp_x <= x_vec_ptr->at(idx + 1))
    ) {
        idx++;
    }

    return idx;
}   /* __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

str	The string being tested.

Returns

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

```
{
    for (size_t i = 0; i < str.size(); i++) {;
        if (isalpha(str[i])) {
            return true;
        }
    }

    return false;
}   /* __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

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

```
{
    //  1. get string matrix
    std::vector<std::vector<std::string» string_matrix =
        this->__getDataStringMatrix(path_2_data);

    //  2. read string matrix contents into 1D interpolation struct
    InterpolatorStruct1D interp_struct_1D;

    interp_struct_1D.n_points = string_matrix.size();
    interp_struct_1D.x_vec.resize(interp_struct_1D.n_points, 0);
    interp_struct_1D.y_vec.resize(interp_struct_1D.n_points, 0);

    for (int i = 0; i < interp_struct_1D.n_points; i++) {
        try {
            interp_struct_1D.x_vec[i] = std::stod(string_matrix[i][0]);
            interp_struct_1D.y_vec[i] = std::stod(string_matrix[i][1]);
        }

        catch (...) {
            this->__throwReadError(path_2_data, 1);
        }
    }

    interp_struct_1D.min_x = interp_struct_1D.x_vec[0];
    interp_struct_1D.max_x = interp_struct_1D.x_vec[interp_struct_1D.n_points - 1];

    //  3. write struct to map
    this->interp_map_1D.insert(
        std::pair<int, InterpolatorStruct1D>(data_key, interp_struct_1D)
    );

    /*
    // ==== TEST PRINT ==== //
    std::cout « std::endl;
    std::cout « path_2_data « std::endl;
    std::cout « "--------" « std::endl;

    std::cout « "n_points: " « this->interp_map_1D[data_key].n_points « std::endl;

    std::cout « "x_vec: [";
    for (
        int i = 0;
        i < this->interp_map_1D[data_key].n_points;
        i++
    ) {
        std::cout « this->interp_map_1D[data_key].x_vec[i] « ", ";
    }
    std::cout « "]" « std::endl;

    std::cout « "y_vec: [";
    for (
        int i = 0;
        i < this->interp_map_1D[data_key].n_points;
        i++
    ) {
        std::cout « this->interp_map_1D[data_key].y_vec[i] « ", ";
    }
    std::cout « "]" « std::endl;

    std::cout « std::endl;
    // ==== END TEST PRINT ==== //
    //*/

    return;
}   /* __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

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

```
{
    //  1. get string matrix
    std::vector<std::vector<std::string» string_matrix =
        this->__getDataStringMatrix(path_2_data);

    //  2. read string matrix contents into 2D interpolation map
    InterpolatorStruct2D interp_struct_2D;

    interp_struct_2D.n_rows = string_matrix.size() - 1;
    interp_struct_2D.n_cols = string_matrix[0].size() - 1;

    interp_struct_2D.x_vec.resize(interp_struct_2D.n_cols, 0);
    interp_struct_2D.y_vec.resize(interp_struct_2D.n_rows, 0);

    interp_struct_2D.z_matrix.resize(interp_struct_2D.n_rows, {});

    for (int i = 0; i < interp_struct_2D.n_rows; i++) {
        interp_struct_2D.z_matrix[i].resize(interp_struct_2D.n_cols, 0);
    }

    for (size_t i = 1; i < string_matrix[0].size(); i++) {
        try {
            interp_struct_2D.x_vec[i - 1] = std::stod(string_matrix[0][i]);
        }

        catch (...) {
            this->__throwReadError(path_2_data, 2);
        }
    }

    interp_struct_2D.min_x = interp_struct_2D.x_vec[0];
    interp_struct_2D.max_x = interp_struct_2D.x_vec[interp_struct_2D.n_cols - 1];

    for (size_t i = 1; i < string_matrix.size(); i++) {
        try {
            interp_struct_2D.y_vec[i - 1] = std::stod(string_matrix[i][0]);
        }

        catch (...) {
            this->__throwReadError(path_2_data, 2);
        }
    }

    interp_struct_2D.min_y = interp_struct_2D.y_vec[0];
    interp_struct_2D.max_y = interp_struct_2D.y_vec[interp_struct_2D.n_rows - 1];

    for (size_t i = 1; i < string_matrix.size(); i++) {
        for (size_t j = 1; j < string_matrix[0].size(); j++) {
            try {
                interp_struct_2D.z_matrix[i - 1][j - 1] = std::stod(string_matrix[i][j]);
            }

            catch (...) {
                this->__throwReadError(path_2_data, 2);
            }
        }
    }

    //  3. write struct to map
    this->interp_map_2D.insert(
        std::pair<int, InterpolatorStruct2D>(data_key, interp_struct_2D)
    );

    /*
    // ==== TEST PRINT ==== //
    std::cout « std::endl;
    std::cout « path_2_data « std::endl;
```

```
    std::cout « "--------" « std::endl;

    std::cout « "n_rows: " « this->interp_map_2D[data_key].n_rows « std::endl;
    std::cout « "n_cols: " « this->interp_map_2D[data_key].n_cols « std::endl;

    std::cout « "x_vec: [";
    for (
        int i = 0;
        i < this->interp_map_2D[data_key].n_cols;
        i++
    ) {
        std::cout « this->interp_map_2D[data_key].x_vec[i] « ", ";
    }
    std::cout « "]" « std::endl;

    std::cout « "y_vec: [";
    for (
        int i = 0;
        i < this->interp_map_2D[data_key].n_rows;
        i++
    ) {
        std::cout « this->interp_map_2D[data_key].y_vec[i] « ", ";
    }
    std::cout « "]" « std::endl;

    std::cout « "z_matrix:" « std::endl;
    for (
        int i = 0;
        i < this->interp_map_2D[data_key].n_rows;
        i++
    ) {
        std::cout « "\t[";

        for (
            int j = 0;
            j < this->interp_map_2D[data_key].n_cols;
            j++
        ) {
            std::cout « this->interp_map_2D[data_key].z_matrix[i][j] « ", ";
        }

        std::cout « "]" « std::endl;
    }
    std::cout « std::endl;

    std::cout « std::endl;
    // ==== END TEST PRINT ==== //
    //*/

    return;
}   /* __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

str	The string to be split.
break_str	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.

```
{
    std::vector<std::string> str_split_vec;

    size_t idx = 0;
    std::string substr;

    while ((idx = str.find(',')) != std::string::npos) {
        substr = str.substr(0, idx);

        if (substr == break_str) {
            break;
        }

        str_split_vec.push_back(substr);

        str.erase(0, idx + 1);
    }

    return str_split_vec;
}   /* __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

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

```
{
    std::string error_str = "ERROR:  Interpolator::addData";
    error_str += std::to_string(dimensions);
    error_str += "D()  ";
    error_str += " failed to read ";
    error_str += path_2_data;
    error_str += " (this is probably a std::stod() error; is there non-numeric ";
    error_str += "data where only numeric data should be?)";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);

    return;
}   /* __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

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

```
{
    //  1. check key
    this->__checkDataKey1D(data_key);

    //  2. read data into map
    this->__readData1D(data_key, path_2_data);

    //  3. record path
    this->path_map_1D.insert(std::pair<int, std::string>(data_key, path_2_data));

    return;
}   /* 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

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

```
{
    //  1. check key
    this->__checkDataKey2D(data_key);

    //  2. read data into map
    this->__readData2D(data_key, path_2_data);

    //  3. record path
    this->path_map_2D.insert(std::pair<int, std::string>(data_key, path_2_data));

    return;
}   /* addData2D() */
```

4.10.3.14 interp1D()

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

Method to perform a 1D interpolation.

Parameters

data_key	A key used to index into the Interpolator.
interp←_x	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.

```
{
    //  1. check bounds
    this->__checkBounds1D(data_key, interp_x);

    //  2. get interpolation index
    int idx = this->__getInterpolationIndex(
        interp_x,
        &(this->interp_map_1D[data_key].x_vec)
    );

    //  3. perform interpolation
    double x_0 = this->interp_map_1D[data_key].x_vec[idx];
    double x_1 = this->interp_map_1D[data_key].x_vec[idx + 1];

    double y_0 = this->interp_map_1D[data_key].y_vec[idx];
    double y_1 = this->interp_map_1D[data_key].y_vec[idx + 1];

    double interp_y = ((y_1 - y_0) / (x_1 - x_0)) * (interp_x - x_0) + y_0;

    return interp_y;
}   /* 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

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

```
{
    //  1. check bounds
    this->__checkBounds2D(data_key, interp_x, interp_y);

    //  2. get interpolation indices
    int idx_x = this->__getInterpolationIndex(
        interp_x,
        &(this->interp_map_2D[data_key].x_vec)
    );

    int idx_y = this->__getInterpolationIndex(
        interp_y,
        &(this->interp_map_2D[data_key].y_vec)
    );

    //  3. perform first horizontal interpolation
    double x_0 = this->interp_map_2D[data_key].x_vec[idx_x];
    double x_1 = this->interp_map_2D[data_key].x_vec[idx_x + 1];

    double z_0 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x];
    double z_1 = this->interp_map_2D[data_key].z_matrix[idx_y][idx_x + 1];
```

```
    double interp_z_0 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;

    //  4. perform second horizontal interpolation
    z_0 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x];
    z_1 = this->interp_map_2D[data_key].z_matrix[idx_y + 1][idx_x + 1];

    double interp_z_1 = ((z_1 - z_0) / (x_1 - x_0)) * (interp_x - x_0) + z_0;

    //  5. perform vertical interpolation
    double y_0 = this->interp_map_2D[data_key].y_vec[idx_y];
    double y_1 = this->interp_map_2D[data_key].y_vec[idx_y + 1];

    double interp_z =
        ((interp_z_1 - interp_z_0) / (y_1 - y_0)) * (interp_y - y_0) + interp_z_0;

    return interp_z;
}   /* 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:

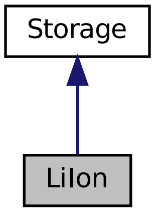
- header/Interpolator.h

4.13 LiIon Class Reference

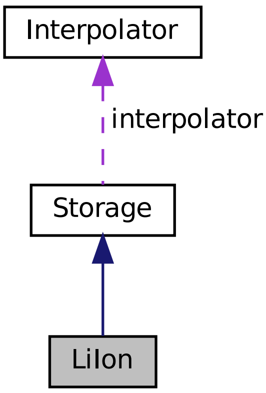
A derived class of Storage which models energy storage by way of lithium-ion batteries.

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

Inheritance diagram for LiIon:



Collaboration diagram for LiIon:



Public Member Functions

- LiIon (void)

    Constructor (dummy) for the LiIon class.
- LiIon (int, double, LiIonInputs)

    Constructor (intended) for the LiIon 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.
- ∼LiIon (void)

  Destructor for the LiIon 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 (LiIonInputs)

  Helper method to check inputs to the LiIon 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 LiIon.
- 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 LiIon.
- void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

  Helper method to write time series results for LiIon.

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 LiIon() [1/2]

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

Constructor (dummy) for the LiIon class.

```
645 {
646     return;
647 }   /* LiIon() */
```

4.13.2.2 LiIon() [2/2]

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

Constructor (intended) for the LiIon class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
liion_inputs	A structure of LiIon constructor inputs.

```
  :
Storage(
    n_points,
    n_years,
    liion_inputs.storage_inputs
)
{
    //  1. check inputs
    this->__checkInputs(liion_inputs);

    //  2. set attributes
    this->type = StorageType :: LIION;
    this->type_str = "LIION";

    this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
    this->SOH = 1;
    this->replace_SOH = liion_inputs.replace_SOH;

    this->degradation_alpha = liion_inputs.degradation_alpha;
    this->degradation_beta = liion_inputs.degradation_beta;
    this->degradation_B_hat_cal_0 = liion_inputs.degradation_B_hat_cal_0;
    this->degradation_r_cal = liion_inputs.degradation_r_cal;
    this->degradation_Ea_cal_0 = liion_inputs.degradation_Ea_cal_0;
    this->degradation_a_cal = liion_inputs.degradation_a_cal;
    this->degradation_s_cal = liion_inputs.degradation_s_cal;
    this->gas_constant_JmolK = liion_inputs.gas_constant_JmolK;
    this->temperature_K = liion_inputs.temperature_K;

    this->init_SOC = liion_inputs.init_SOC;
    this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;

    this->min_SOC = liion_inputs.min_SOC;
    this->hysteresis_SOC = liion_inputs.hysteresis_SOC;
    this->max_SOC = liion_inputs.max_SOC;

    this->charging_efficiency = liion_inputs.charging_efficiency;
    this->discharging_efficiency = liion_inputs.discharging_efficiency;

    if (liion_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = liion_inputs.capital_cost;
    }

    if (liion_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            liion_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    this->SOH_vec.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
        std::cout « "LiIon object constructed at " « this « std::endl;
    }

    return;
}   /* LiIon() */
```

4.13.2.3 ∼LiIon()

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

Destructor for the LiIon class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "LiIon object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

liion_inputs	A structure of LiIon constructor inputs.

```
{
    //  1. check replace_SOH
    if (liion_inputs.replace_SOH < 0 or liion_inputs.replace_SOH > 1) {
        std::string error_str = "ERROR:  LiIon():  replace_SOH must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check init_SOC
    if (liion_inputs.init_SOC < 0 or liion_inputs.init_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  init_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check min_SOC
    if (liion_inputs.min_SOC < 0 or liion_inputs.min_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  min_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check hysteresis_SOC
    if (liion_inputs.hysteresis_SOC < 0 or liion_inputs.hysteresis_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  hysteresis_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }
```

```

    //  5. check max_SOC
    if (liion_inputs.max_SOC < 0 or liion_inputs.max_SOC > 1) {
        std::string error_str = "ERROR:  LiIon():  max_SOC must be in the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

     //  6. check charging_efficiency
     if (liion_inputs.charging_efficiency <= 0 or liion_inputs.charging_efficiency > 1) {
         std::string error_str = "ERROR:  LiIon():  charging_efficiency must be in the ";
         error_str += "half-open interval (0, 1]";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  7. check discharging_efficiency
     if (
         liion_inputs.discharging_efficiency <= 0 or
         liion_inputs.discharging_efficiency > 1
     ) {
         std::string error_str = "ERROR:  LiIon():  discharging_efficiency must be in the ";
         error_str += "half-open interval (0, 1]";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  8. check degradation_alpha
     if (liion_inputs.degradation_alpha <= 0) {
         std::string error_str = "ERROR:  LiIon():  degradation_alpha must be > 0";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  9. check degradation_beta
     if (liion_inputs.degradation_beta <= 0) {
         std::string error_str = "ERROR:  LiIon():  degradation_beta must be > 0";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  10. check degradation_B_hat_cal_0
     if (liion_inputs.degradation_B_hat_cal_0 <= 0) {
         std::string error_str = "ERROR:  LiIon():  degradation_B_hat_cal_0 must be > 0";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  11. check degradation_r_cal
     if (liion_inputs.degradation_r_cal < 0) {
         std::string error_str = "ERROR:  LiIon():  degradation_r_cal must be >= 0";

         #ifdef _WIN32
             std::cout « error_str « std::endl;
         #endif

         throw std::invalid_argument(error_str);
     }

     //  12. check degradation_Ea_cal_0
     if (liion_inputs.degradation_Ea_cal_0 <= 0) {
         std::string error_str = "ERROR:  LiIon():  degradation_Ea_cal_0 must be > 0";
```

```
        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  13. check degradation_a_cal
    if (liion_inputs.degradation_a_cal < 0) {
        std::string error_str = "ERROR:  LiIon():  degradation_a_cal must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  14. check degradation_s_cal
    if (liion_inputs.degradation_s_cal < 0) {
        std::string error_str = "ERROR:  LiIon():  degradation_s_cal must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  15. check gas_constant_JmolK
    if (liion_inputs.gas_constant_JmolK <= 0) {
        std::string error_str = "ERROR:  LiIon():  gas_constant_JmolK must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  16. check temperature_K
    if (liion_inputs.temperature_K < 0) {
        std::string error_str = "ERROR:  LiIon():  temperature_K must be >= 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.13.3.2 __getBcal()

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

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

Ref: Truelove [2023a]

Parameters

SOC	The current state of charge of the asset.

Returns

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

```
{
    double B_cal = this->degradation_B_hat_cal_0 *
        exp(this->degradation_r_cal * SOC);

    return B_cal;
}   /* __getBcal() */
```

4.13.3.3 __getEacal()

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

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

Ref: Truelove [2023a]

Parameters

SOC	The current state of charge of the asset.

Returns

The activation energy value for the given state of charge.

```
{
    double Ea_cal = this->degradation_Ea_cal_0;

    Ea_cal -= this->degradation_a_cal *
        (exp(this->degradation_s_cal * SOC) - 1);

    return Ea_cal;
}   /* __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].

```
{
    double capital_cost_per_kWh = 250 * pow(this->energy_capacity_kWh, -0.15) + 650;

    return capital_cost_per_kWh * this->energy_capacity_kWh;
}   /* __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].

```
{
    return 0.01;
}   /* __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

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
charging_discharging_kW	The charging/discharging power [kw] being sent to the asset.

```
{
    //  1. model degradation
    this->__modelDegradation(dt_hrs, charging_discharging_kW);

    //  2. update and record
    this->SOH_vec[timestep] = this->SOH;
    this->dynamic_energy_capacity_kWh = this->SOH * this->energy_capacity_kWh;

    return;
}   /* __handleDegradation() */
```

4.13.3.7 __modelDegradation()

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

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

Ref: Truelove [2023a]

Parameters

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

```
{
    //  1. compute SOC
    double SOC = this->charge_kWh / this->energy_capacity_kWh;

    //  2. compute C-rate and corresponding acceleration factor
    double C_rate = charging_discharging_kW / this->power_capacity_kW;

    double C_acceleration_factor =
        1 + this->degradation_alpha * pow(C_rate, this->degradation_beta);

    //  3. compute dSOH / dt
    double B_cal = __getBcal(SOC);
    double Ea_cal = __getEacal(SOC);

    double dSOH_dt = B_cal *
        exp((-1 * Ea_cal) / (this->gas_constant_JmolK * this->temperature_K));

    dSOH_dt *= dSOH_dt;
    dSOH_dt *= 1 / (2 * this->SOH);
    dSOH_dt *= C_acceleration_factor;

    //  4. update state of health
    this->SOH -= dSOH_dt * dt_hrs;

    return;
}   /* __modelDegradation() */
```

4.13.3.8 __toggleDepleted()

```
void LiIon::__toggleDepleted (
              void  )  [private]
```

Helper method to toggle the is_depleted attribute of LiIon.

```
{
    if (this->is_depleted) {
        double hysteresis_charge_kWh = this->hysteresis_SOC * this->energy_capacity_kWh;

        if (hysteresis_charge_kWh > this->dynamic_energy_capacity_kWh) {
            hysteresis_charge_kWh = this->dynamic_energy_capacity_kWh;
        }

        if (this->charge_kWh >= hysteresis_charge_kWh) {
            this->is_depleted = false;
        }
    }

    else {
        double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;

        if (this->charge_kWh <= min_charge_kWh) {
            this->is_depleted = true;
        }
    }

    return;
}   /* __toggleDepleted() */
```

4.13.3.9 __writeSummary()

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

Helper method to write summary results for LiIon.

Parameters

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

Reimplemented from Storage.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->power_capacity_kW)));
    ofs « " kW ";
    ofs « std::to_string(int(ceil(this->energy_capacity_kWh)));
    ofs « " kWh LIION Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Storage attributes
    ofs « "## Storage Attributes\n";
    ofs « "\n";
    ofs « "Power Capacity: " « this->power_capacity_kW « " kW  \n";
    ofs « "Energy Capacity: " « this->energy_capacity_kWh « " kWh  \n";
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh charged/discharged  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";

    ofs « "\n--------\n\n";

    //  2.2. LiIon attributes
    ofs « "## LiIon Attributes\n";
    ofs « "\n";

    ofs « "Charging Efficiency: " « this->charging_efficiency « "  \n";
    ofs « "Discharging Efficiency: " « this->discharging_efficiency « "  \n";
    ofs « "\n";

    ofs « "Initial State of Charge: " « this->init_SOC « "  \n";
    ofs « "Minimum State of Charge: " « this->min_SOC « "  \n";
    ofs « "Hyteresis State of Charge: " « this->hysteresis_SOC « "  \n";
    ofs « "Maximum State of Charge: " « this->max_SOC « "  \n";
    ofs « "\n";

    ofs « "Replacement State of Health: " « this->replace_SOH « "  \n";
    ofs « "\n";

    ofs « "Degradation Acceleration Coeff.: " « this->degradation_alpha « "  \n";
    ofs « "Degradation Acceleration Exp.: " « this->degradation_beta « "  \n";
    ofs « "Degradation Base Pre-Exponential Factor: "
        « this->degradation_B_hat_cal_0 « " 1/sqrt(hrs)  \n";
    ofs « "Degradation Dimensionless Constant (r_cal): "
        « this->degradation_r_cal « "  \n";
    ofs « "Degradation Base Activation Energy: "
        « this->degradation_Ea_cal_0 « " J/mol  \n";
    ofs « "Degradation Pre-Exponential Factor: "
        « this->degradation_a_cal « " J/mol  \n";
    ofs « "Degradation Dimensionless Constant (s_cal): "
        « this->degradation_s_cal « "  \n";
    ofs « "Universal Gas Constant: " « this->gas_constant_JmolK
        « " J/mol.K  \n";
    ofs « "Absolute Environmental Temperature: " « this->temperature_K « " K  \n";

    ofs « "\n--------\n\n";

    //  2.3. LiIon Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Discharge: " « this->total_discharge_kWh
        « " kWh  \n";
```

```
    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";
    ofs.close();
    return;
}   /* __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 LiIon.

Parameters

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

Reimplemented from Storage.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Charging Power [kW],";
    ofs « "Discharging Power [kW],";
    ofs « "Charge (at end of timestep) [kWh],";
    ofs « "State of Health (at end of timestep) [ ],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";
        ofs « this->charging_power_vec_kW[i] « ",";
        ofs « this->discharging_power_vec_kW[i] « ",";
        ofs « this->charge_vec_kWh[i] « ",";
        ofs « this->SOH_vec[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __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

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
charging_kW	The charging power [kw] being sent to the asset.

Reimplemented from Storage.

```
{
    //  1. record charging power
    this->charging_power_vec_kW[timestep] = charging_kW;

    //  2. update charge and record
    this->charge_kWh += this->charging_efficiency * charging_kW * dt_hrs;
    this->charge_vec_kWh[timestep] = this->charge_kWh;

    //  3. toggle depleted flag (if applicable)
    this->__toggleDepleted();

    //  4. model degradation
    this->__handleDegradation(timestep, dt_hrs, charging_kW);

    //  5. trigger replacement (if applicable)
    if (this->SOH <= this->replace_SOH) {
        this->handleReplacement(timestep);
    }

    //  6. capture operation and maintenance costs (if applicable)
    if (charging_kW > 0) {
        this->operation_maintenance_cost_vec[timestep] = charging_kW * dt_hrs *
            this->operation_maintenance_cost_kWh;
    }

    this->power_kW= 0;
    return;
}   /* 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

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
discharging_kW	The discharging power [kw] being drawn from the asset.
load_kW	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.

```
{
    //  1. record discharging power, update total
    this->discharging_power_vec_kW[timestep] = discharging_kW;
    this->total_discharge_kWh += discharging_kW * dt_hrs;

    //  2. update charge and record
    this->charge_kWh -= (discharging_kW * dt_hrs) / this->discharging_efficiency;
    this->charge_vec_kWh[timestep] = this->charge_kWh;

    //  3. update load
    load_kW -= discharging_kW;

    //  4. toggle depleted flag (if applicable)
    this->__toggleDepleted();

    //  5. model degradation
    this->__handleDegradation(timestep, dt_hrs, discharging_kW);

    //  6. trigger replacement (if applicable)
    if (this->SOH <= this->replace_SOH) {
        this->handleReplacement(timestep);
    }

    //  7. capture operation and maintenance costs (if applicable)
    if (discharging_kW > 0) {
        this->operation_maintenance_cost_vec[timestep] = discharging_kW * dt_hrs *
            this->operation_maintenance_cost_kWh;
    }

    this->power_kW = 0;
    return load_kW;
}   /* 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

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

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

Reimplemented from Storage.

```
{
    //  1. get max charge
    double max_charge_kWh = this->max_SOC * this->energy_capacity_kWh;

    if (max_charge_kWh > this->dynamic_energy_capacity_kWh) {
        max_charge_kWh = this->dynamic_energy_capacity_kWh;
    }

    //  2. compute acceptable power
    //     (accounting for the power currently being charged/discharged by the asset)
    double acceptable_kW =
        (max_charge_kWh - this->charge_kWh) /
        (this->charging_efficiency * dt_hrs);
```

```
    acceptable_kW -= this->power_kW;

    if (acceptable_kW <= 0) {
        return 0;
    }

    //  3. apply power constraint
    if (acceptable_kW > this->power_capacity_kW) {
        acceptable_kW = this->power_capacity_kW;
    }

    return acceptable_kW;
}   /* 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

dt_hrs	The interval of time [hrs] associated with the timestep.

Returns

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

Reimplemented from Storage.

```
{
    //  1. get min charge
    double min_charge_kWh = this->min_SOC * this->energy_capacity_kWh;

    //  2. compute available power
    //     (accounting for the power currently being charged/discharged by the asset)
    double available_kW =
        ((this->charge_kWh - min_charge_kWh) * this->discharging_efficiency) /
        dt_hrs;

    available_kW -= this->power_kW;

    if (available_kW <= 0) {
        return 0;
    }

    //  3. apply power constraint
    if (available_kW > this->power_capacity_kW) {
        available_kW = this->power_capacity_kW;
    }

    return available_kW;
}   /* getAvailablekW() */
```

4.13.3.15 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Storage.

```
{
    //  1. reset attributes
    this->dynamic_energy_capacity_kWh = this->energy_capacity_kWh;
    this->SOH = 1;

    // 2. invoke base class method
    Storage::handleReplacement(timestep);

    //  3. correct attributes
    this->charge_kWh = this->init_SOC * this->energy_capacity_kWh;
    this->is_depleted = false;

    return;
}   /* __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:

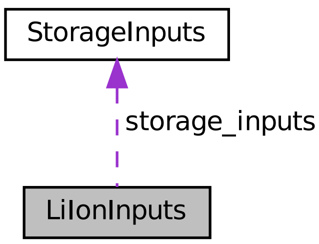
- header/Storage/LiIon.h
- source/Storage/LiIon.cpp

4.14 LiIonInputs Struct Reference

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

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

Collaboration diagram for LiIonInputs:



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 LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.

Ref: Truelove [2023a]

4.14.2 Member Data Documentation

4.14.2.1 capital_cost

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

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

4.14.2.2 charging_efficiency

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

The charging efficiency of the asset.

4.14.2.3 degradation_a_cal

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

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

4.14.2.4 degradation_alpha

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

A dimensionless acceleration coefficient used in modelling energy capacity degradation.

4.14.2.5 degradation_B_hat_cal_0

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

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

4.14.2.6 degradation_beta

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

A dimensionless acceleration exponent used in modelling energy capacity degradation.

4.14.2.7 degradation_Ea_cal_0

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

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

4.14.2.8 degradation_r_cal

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

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.9 degradation_s_cal

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

A dimensionless constant used in modelling energy capacity degradation.

4.14.2.10 discharging_efficiency

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

The discharging efficiency of the asset.

4.14.2.11 gas_constant_JmolK

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

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

4.14.2.12 hysteresis_SOC

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

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

4.14.2.13 init_SOC

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

The initial state of charge of the asset.

4.14.2.14 max_SOC

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

The maximum state of charge of the asset.

4.14.2.15 min_SOC

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

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

4.14.2.16 operation_maintenance_cost_kWh

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

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

4.14.2.17 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:

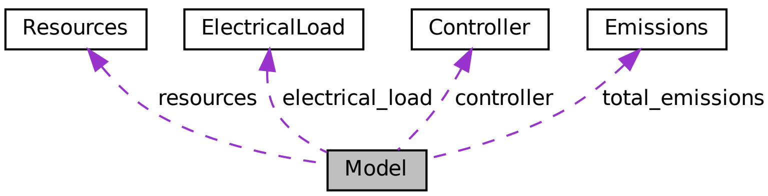
- header/Storage/LiIon.h

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 addLiIon (LiIonInputs)

  Method to add a LiIon 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 attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.
- void clear (void)

  Method to clear all attributes of the Model object.
- void writeResults (std::string, int=-1)

  Method which writes Model results to an output directory. Also calls out to writeResults() for each contained asset.
- ∼Model (void)

  Destructor for the Model class.

Public Attributes

- double total_fuel_consumed_L

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

  An Emissions structure for holding total emissions [kg].
- double net_present_cost

  The net present cost of the Model (undefined currency).
- double total_renewable_dispatch_kWh

  The total energy dispatched [kWh] by all renewable assets over the Model run.
- double total_dispatch_discharge_kWh

  The total energy dispatched/discharged [kWh] over the Model run.
- double levellized_cost_of_energy_kWh

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

- Controller controller

    Controller component of Model.
- ElectricalLoad electrical_load

    ElectricalLoad component of Model.
- Resources resources

    Resources component of Model.
- std::vector< Combustion * > combustion_ptr_vec

    A vector of pointers to the various Combustion assets in the Model.
- std::vector< Noncombustion * > noncombustion_ptr_vec

    A vector of pointers to the various Noncombustion assets in the Model.
- std::vector< Renewable * > renewable_ptr_vec

    A vector of pointers to the various Renewable assets in the Model.
- std::vector< Storage * > storage_ptr_vec

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

Private Member Functions

- void __checkInputs (ModelInputs)

    Helper method (private) to check inputs to the Model constructor.
- void __computeFuelAndEmissions (void)

    Helper method to compute the total fuel consumption and emissions over the Model run.
- void __computeNetPresentCost (void)

    Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs. Also tallies up total dispatch and discharge.
- void __computeLevellizedCostOfEnergy (void)

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

    Helper method to compute key economic metrics for the Model run.
- void __writeSummary (std::string)

    Helper method to write summary results for Model.
- void __writeTimeSeries (std::string, int=-1)

    Helper method to write time series results for Model.

4.15.1 Detailed Description

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

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Model() [1/2]

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

Constructor (dummy) for the Model class.

```
{
    return;
}   /* Model() */
```

4.15.2.2 Model() [2/2]

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

Constructor (intended) for the Model class.

Parameters

model_inputs	A structure of Model constructor inputs.

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

    //  2. read in electrical load data
    this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);

    //  3. set control mode
    this->controller.setControlMode(model_inputs.control_mode);

    //  4. set public attributes
    this->total_fuel_consumed_L = 0;
    this->net_present_cost = 0;
    this->total_dispatch_discharge_kWh = 0;
    this->total_renewable_dispatch_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    return;
}   /* Model() */
```

4.15.2.3 ∼Model()

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

Destructor for the Model class.

```
{
    this->clear();
    return;
}   /* ~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

model_inputs	A structure of Model constructor inputs.

```
{
    //  1. check path_2_electrical_load_time_series
    if (model_inputs.path_2_electrical_load_time_series.empty()) {
        std::string error_str = "ERROR:  Model()  path_2_electrical_load_time_series ";
        error_str += "cannot be empty";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.15.3.2 __computeEconomics()

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

Helper method to compute key economic metrics for the Model run.

```
{
    this->__computeNetPresentCost();
    this->__computeLevellizedCostOfEnergy();

    return;
}   /* __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.

```
{
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->computeFuelAndEmissions();

        this->total_fuel_consumed_L +=
            this->combustion_ptr_vec[i]->total_fuel_consumed_L;

        this->total_emissions.CO2_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CO2_kg;

        this->total_emissions.CO_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CO_kg;

        this->total_emissions.NOx_kg +=
            this->combustion_ptr_vec[i]->total_emissions.NOx_kg;

        this->total_emissions.SOx_kg +=
```

```
            this->combustion_ptr_vec[i]->total_emissions.SOx_kg;

        this->total_emissions.CH4_kg +=
            this->combustion_ptr_vec[i]->total_emissions.CH4_kg;

        this->total_emissions.PM_kg +=
            this->combustion_ptr_vec[i]->total_emissions.PM_kg;
    }

    return;
}   /* __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.

```
{
    //  1. account for Combustion economics in levellized cost of energy
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->combustion_ptr_vec[i]->total_dispatch_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

    //  2. account for Noncombustion economics in levellized cost of energy
    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->noncombustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->noncombustion_ptr_vec[i]->total_dispatch_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

    //  3. account for Renewable economics in levellized cost of energy
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->renewable_ptr_vec[i]->total_dispatch_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

    //  4. account for Storage economics in levellized cost of energy
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        this->levellized_cost_of_energy_kWh +=
            (
                this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
                this->storage_ptr_vec[i]->total_discharge_kWh
            ) / this->total_dispatch_discharge_kWh;
    }

    return;
}   /* __computeLevellizedCostOfEnergy() */
```

4.15.3.5 __computeNetPresentCost()

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

Helper method to compute the overall net present cost, for the Model run, from the asset-wise net present costs. Also tallies up total dispatch and discharge.

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

```
    //     increment total dispatch
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->combustion_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->combustion_ptr_vec[i]->total_dispatch_kWh;
    }

    //  2. account for Noncombustion economics in net present cost
    //     increment total dispatch
    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        this->noncombustion_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->noncombustion_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->noncombustion_ptr_vec[i]->total_dispatch_kWh;
    }

    //  3. account for Renewable economics in net present cost,
    //     increment total dispatch
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->renewable_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->renewable_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->renewable_ptr_vec[i]->total_dispatch_kWh;

        this->total_renewable_dispatch_kWh +=
            this->renewable_ptr_vec[i]->total_dispatch_kWh;
    }

    //  4. account for Storage economics in net present cost
    //     increment total dispatch
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        this->storage_ptr_vec[i]->computeEconomics(
            &(this->electrical_load.time_vec_hrs)
        );

        this->net_present_cost += this->storage_ptr_vec[i]->net_present_cost;

        this->total_dispatch_discharge_kWh +=
            this->storage_ptr_vec[i]->total_discharge_kWh;
    }

    return;
}   /* __computeNetPresentCost() */
```

4.15.3.6 __writeSummary()

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

Helper method to write summary results for Model.

Parameters

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

```
{
    //  1. create subdirectory
    write_path += "Model/";
    std::filesystem::create_directory(write_path);

```

```
    //  2. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  3. write summary results (markdown)
    ofs « "# Model Summary Results\n";
    ofs « "\n--------\n\n";

    //  3.1. ElectricalLoad
    ofs « "## Electrical Load\n";
    ofs « "\n";
    ofs « "Path: " «
        this->electrical_load.path_2_electrical_load_time_series « "  \n";
    ofs « "Data Points: " « this->electrical_load.n_points « "  \n";
    ofs « "Years: " « this->electrical_load.n_years « "  \n";
    ofs « "Min: " « this->electrical_load.min_load_kW « " kW  \n";
    ofs « "Mean: " « this->electrical_load.mean_load_kW « " kW  \n";
    ofs « "Max: " « this->electrical_load.max_load_kW « " kW  \n";
    ofs « "\n--------\n\n";

    //  3.2. Controller
    ofs « "## Controller\n";
    ofs « "\n";
    ofs « "Control Mode: " « this->controller.control_string « "  \n";
    ofs « "\n--------\n\n";

    //  3.3. Resources (1D)
    ofs « "## 1D Renewable Resources\n";
    ofs « "\n";

    std::map<int, std::string>::iterator string_map_1D_iter =
        this->resources.string_map_1D.begin();
    std::map<int, std::string>::iterator path_map_1D_iter =
        this->resources.path_map_1D.begin();

    while (
        string_map_1D_iter != this->resources.string_map_1D.end() and
        path_map_1D_iter != this->resources.path_map_1D.end()
    ) {
        ofs « "Resource Key: " « string_map_1D_iter->first « "  \n";
        ofs « "Type: " « string_map_1D_iter->second « "  \n";
        ofs « "Path: " « path_map_1D_iter->second « "  \n";
        ofs « "\n";

        string_map_1D_iter++;
        path_map_1D_iter++;
    }

    ofs « "\n--------\n\n";

    //  3.4. Resources (2D)
    ofs « "## 2D Renewable Resources\n";
    ofs « "\n";

    std::map<int, std::string>::iterator string_map_2D_iter =
        this->resources.string_map_2D.begin();
    std::map<int, std::string>::iterator path_map_2D_iter =
        this->resources.path_map_2D.begin();

    while (
        string_map_2D_iter != this->resources.string_map_2D.end() and
        path_map_2D_iter != this->resources.path_map_2D.end()
    ) {
        ofs « "Resource Key: " « string_map_2D_iter->first « "  \n";
        ofs « "Type: " « string_map_2D_iter->second « "  \n";
        ofs « "Path: " « path_map_2D_iter->second « "  \n";
        ofs « "\n";

        string_map_2D_iter++;
        path_map_2D_iter++;
    }

    ofs « "\n--------\n\n";

    //  3.5. Combustion
    ofs « "## Combustion Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->combustion_ptr_vec[i]->type_str « "  \n";
        ofs « "Capacity: " « this->combustion_ptr_vec[i]->capacity_kW « " kW  \n";
        ofs « "\n";
    }

    ofs « "\n--------\n\n";
```

```
    //  3.6. Noncombustion
    ofs « "## Noncombustion Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->noncombustion_ptr_vec[i]->type_str « "  \n";
        ofs « "Capacity: " « this->noncombustion_ptr_vec[i]->capacity_kW « " kW  \n";

        if (this->noncombustion_ptr_vec[i]->type == NoncombustionType :: HYDRO) {
            ofs « "Reservoir Capacity: " «
                ((Hydro*)(this->noncombustion_ptr_vec[i]))->reservoir_capacity_m3 «
                " m3  \n";
        }

        ofs « "\n";
    }

    ofs « "\n--------\n\n";

    //  3.7. Renewable
    ofs « "## Renewable Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->renewable_ptr_vec[i]->type_str « "  \n";
        ofs « "Capacity: " « this->renewable_ptr_vec[i]->capacity_kW « " kW  \n";
        ofs « "\n";
    }

    ofs « "\n--------\n\n";

    //  3.8. Storage
    ofs « "## Storage Assets\n";
    ofs « "\n";

    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        ofs « "Asset Index: " « i « "  \n";
        ofs « "Type: " « this->storage_ptr_vec[i]->type_str « "  \n";
        ofs « "Power Capacity: " « this->storage_ptr_vec[i]->power_capacity_kW
            « " kW  \n";
        ofs « "Energy Capacity: " « this->storage_ptr_vec[i]->energy_capacity_kWh
            « " kWh  \n";
        ofs « "\n";
    }

    ofs « "\n--------\n\n";

    //  3.9. Model Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch + Discharge: " « this->total_dispatch_discharge_kWh
        « " kWh  \n";

    ofs « "Renewable Penetration: "
        « this->total_renewable_dispatch_kWh / this->total_dispatch_discharge_kWh
        « "  \n";
    ofs « "\n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched/discharged  \n";
    ofs « "\n";

    ofs « "Total Fuel Consumed: " « this->total_fuel_consumed_L « " L "
        « "(Annual Average: " «
            this->total_fuel_consumed_L / this->electrical_load.n_years
        « " L/yr)  \n";
    ofs « "\n";

    ofs « "Total Carbon Dioxide (CO2) Emissions: " «
        this->total_emissions.CO2_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CO2_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Carbon Monoxide (CO) Emissions: " «
        this->total_emissions.CO_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CO_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

```

```
    ofs « "Total Nitrogen Oxides (NOx) Emissions: " «
        this->total_emissions.NOx_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.NOx_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Sulfur Oxides (SOx) Emissions: " «
        this->total_emissions.SOx_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.SOx_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Methane (CH4) Emissions: " « this->total_emissions.CH4_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.CH4_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "Total Particulate Matter (PM) Emissions: " «
        this->total_emissions.PM_kg « " kg "
        « "(Annual Average: " «
            this->total_emissions.PM_kg / this->electrical_load.n_years
        « " kg/yr)  \n";

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __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

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

```
{
    //  1. create filestream
    write_path += "Model/time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results header (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Electrical Load [kW],";
    ofs « "Net Load [kW],";
    ofs « "Missed Load [kW],";

    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        ofs « this->renewable_ptr_vec[i]->capacity_kW « " kW "
            « this->renewable_ptr_vec[i]->type_str « " Dispatch [kW],";
    }

    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        ofs « this->storage_ptr_vec[i]->power_capacity_kW « " kW "
            « this->storage_ptr_vec[i]->energy_capacity_kWh « " kWh "
            « this->storage_ptr_vec[i]->type_str « " Discharge [kW],";
    }

    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        ofs « this->noncombustion_ptr_vec[i]->capacity_kW « " kW "
            « this->noncombustion_ptr_vec[i]->type_str « " Dispatch [kW],";
    }

    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        ofs « this->combustion_ptr_vec[i]->capacity_kW « " kW "
```

```
            « this->combustion_ptr_vec[i]->type_str « " Dispatch [kW],";
    }

    ofs « "\n";

    //  3. write time series results values (comma separated value)
    for (int i = 0; i < max_lines; i++) {
        //  3.1. load values
        ofs « this->electrical_load.time_vec_hrs[i] « ",";
        ofs « this->electrical_load.load_vec_kW[i] « ",";
        ofs « this->controller.net_load_vec_kW[i] « ",";
        ofs « this->controller.missed_load_vec_kW[i] « ",";

        //  3.2. asset-wise dispatch/discharge
        for (size_t j = 0; j < this->renewable_ptr_vec.size(); j++) {
            ofs « this->renewable_ptr_vec[j]->dispatch_vec_kW[i] « ",";
        }

        for (size_t j = 0; j < this->storage_ptr_vec.size(); j++) {
            ofs « this->storage_ptr_vec[j]->discharging_power_vec_kW[i] « ",";
        }

        for (size_t j = 0; j < this->noncombustion_ptr_vec.size(); j++) {
            ofs « this->noncombustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
        }

        for (size_t j = 0; j < this->combustion_ptr_vec.size(); j++) {
            ofs « this->combustion_ptr_vec[j]->dispatch_vec_kW[i] « ",";
        }

        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __writeTimeSeries() */
```

4.15.3.8 addDiesel()

```
void Model::addDiesel (
        DieselInputs diesel_inputs )
```

Method to add a Diesel asset to the Model.

Parameters

diesel_inputs	A structure of Diesel constructor inputs.

```
{
    Combustion* diesel_ptr = new Diesel(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        diesel_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->combustion_ptr_vec.push_back(diesel_ptr);

    return;
}   /* addDiesel() */
```

4.15.3.9 addHydro()

```
void Model::addHydro (
        HydroInputs hydro_inputs )
```

Method to add a Hydro asset to the Model.

Parameters

hydro_inputs	A structure of Hydro constructor inputs.

```
{
    Noncombustion* hydro_ptr = new Hydro(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        hydro_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->noncombustion_ptr_vec.push_back(hydro_ptr);

    return;
}   /* addHydro() */
```

4.15.3.10 addLiIon()

```
void Model::addLiIon (
            LiIonInputs liion_inputs )
```

Method to add a LiIon asset to the Model.

Parameters

liion_inputs	A structure of LiIon constructor inputs.

```
{
    Storage* liion_ptr = new LiIon(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        liion_inputs
    );

    this->storage_ptr_vec.push_back(liion_ptr);

    return;
}   /* 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

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

```
{
    resources.addResource(
        noncombustion_type,
        path_2_resource_data,
        resource_key,
        &(this->electrical_load)
    );

    return;
}   /* 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

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

```
{
    resources.addResource(
        renewable_type,
        path_2_resource_data,
        resource_key,
        &(this->electrical_load)
    );

    return;
}   /* addResource() */
```

4.15.3.13 addSolar()

```
void Model::addSolar (
            SolarInputs solar_inputs )
```

Method to add a Solar asset to the Model.

Parameters

solar_inputs	A structure of Solar constructor inputs.

```
{
    Renewable* solar_ptr = new Solar(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        solar_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->renewable_ptr_vec.push_back(solar_ptr);
```

```
    return;
} /* addSolar() */
```

4.15.3.14 addTidal()

```
void Model::addTidal (
        TidalInputs tidal_inputs )
```

Method to add a Tidal asset to the Model.

Parameters

tidal_inputs	A structure of Tidal constructor inputs.

```
{
    Renewable* tidal_ptr = new Tidal(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        tidal_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->renewable_ptr_vec.push_back(tidal_ptr);

    return;
} /* addTidal() */
```

4.15.3.15 addWave()

```
void Model::addWave (
        WaveInputs wave_inputs )
```

Method to add a Wave asset to the Model.

Parameters

wave_inputs	A structure of Wave constructor inputs.

```
{
    Renewable* wave_ptr = new Wave(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        wave_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->renewable_ptr_vec.push_back(wave_ptr);

    return;
} /* addWave() */
```

4.15.3.16 addWind()

```
void Model::addWind (
        WindInputs wind_inputs )
```

Method to add a Wind asset to the Model.

Parameters

wind_inputs	A structure of Wind constructor inputs.

```
{
    Renewable* wind_ptr = new Wind(
        this->electrical_load.n_points,
        this->electrical_load.n_years,
        wind_inputs,
        &(this->electrical_load.time_vec_hrs)
    );

    this->renewable_ptr_vec.push_back(wind_ptr);

    return;
}   /* addWind() */
```

4.15.3.17 clear()

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

Method to clear all attributes of the Model object.

```
{
    //  1. reset
    this->reset();

    //  2. clear components
    controller.clear();
    electrical_load.clear();
    resources.clear();

    return;
}   /* 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 attribues. It leaves the Controller, ElectricalLoad, and Resources objects of the Model alone.

```
{
    //  1. clear combustion_ptr_vec
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        delete this->combustion_ptr_vec[i];
    }
    this->combustion_ptr_vec.clear();

    //  2. clear noncombustion_ptr_vec
    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        delete this->noncombustion_ptr_vec[i];
    }
    this->noncombustion_ptr_vec.clear();

    //  3. clear renewable_ptr_vec
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        delete this->renewable_ptr_vec[i];
    }
    this->renewable_ptr_vec.clear();

    //  4. clear storage_ptr_vec
```

```
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        delete this->storage_ptr_vec[i];
    }
    this->storage_ptr_vec.clear();

    //  5. reset components and attributes
    this->controller.clear();

    this->total_fuel_consumed_L = 0;

    this->total_emissions.CO2_kg = 0;
    this->total_emissions.CO_kg = 0;
    this->total_emissions.NOx_kg = 0;
    this->total_emissions.SOx_kg = 0;
    this->total_emissions.CH4_kg = 0;
    this->total_emissions.PM_kg = 0;

    this->net_present_cost = 0;
    this->total_dispatch_discharge_kWh = 0;
    this->total_renewable_dispatch_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    return;
}   /* reset() */
```

4.15.3.19 run()

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

A method to run the Model.

```
{
    // 1. init Controller
    this->controller.init(
        &(this->electrical_load),
        &(this->renewable_ptr_vec),
        &(this->resources),
        &(this->combustion_ptr_vec)
    );

    //  2. apply dispatch control
    this->controller.applyDispatchControl(
        &(this->electrical_load),
        &(this->resources),
        &(this->combustion_ptr_vec),
        &(this->noncombustion_ptr_vec),
        &(this->renewable_ptr_vec),
        &(this->storage_ptr_vec)
    );

    //  3. compute total fuel consumption and emissions
    this->__computeFuelAndEmissions();

    //  4. compute key economic metrics
    this->__computeEconomics();

    return;
}   /* 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

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

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->electrical_load.n_points;
    }

    //  2. check for pre-existing, warn (and remove), then create
    if (write_path.back() != '/') {
        write_path += '/';
    }

    if (std::filesystem::is_directory(write_path)) {
        std::string warning_str = "WARNING:  Model::writeResults():  ";
        warning_str += write_path;
        warning_str += " already exists, contents will be overwritten!";

        std::cout « warning_str « std::endl;

        std::filesystem::remove_all(write_path);
    }

    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->electrical_load.n_points) {
        max_lines = this->electrical_load.n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(write_path, max_lines);
    }

    //  5. call out to Combustion :: writeResults()
    for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
        this->combustion_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            i,
            max_lines
        );
    }

    //  6. call out to Noncombustion :: writeResults()
    for (size_t i = 0; i < this->noncombustion_ptr_vec.size(); i++) {
        this->noncombustion_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            i,
            max_lines
        );
    }

    //  7. call out to Renewable :: writeResults()
    for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
        this->renewable_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            &(this->resources.resource_map_1D),
            &(this->resources.resource_map_2D),
            i,
            max_lines
        );
    }

    //  8. call out to Storage :: writeResults()
    for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
        this->storage_ptr_vec[i]->writeResults(
            write_path,
            &(this->electrical_load.time_vec_hrs),
            i,
            max_lines
        );
    }

```

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

4.15.4 Member Data Documentation

4.15.4.1 combustion_ptr_vec

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

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

4.15.4.2 controller

`Controller Model::controller`

Controller component of Model.

4.15.4.3 electrical_load

`ElectricalLoad Model::electrical_load`

ElectricalLoad component of Model.

4.15.4.4 levellized_cost_of_energy_kWh

`double Model::levellized_cost_of_energy_kWh`

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

4.15.4.5 net_present_cost

`double Model::net_present_cost`

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

4.15.4.6 noncombustion_ptr_vec

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

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

4.15.4.7 renewable_ptr_vec

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

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

4.15.4.8 resources

`Resources Model::resources`

Resources component of Model.

4.15.4.9 storage_ptr_vec

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

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

4.15.4.10 total_dispatch_discharge_kWh

`double Model::total_dispatch_discharge_kWh`

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

4.15.4.11 total_emissions

`Emissions Model::total_emissions`

An Emissions structure for holding total emissions [kg].

4.15.4.12 total_fuel_consumed_L

`double Model::total_fuel_consumed_L`

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

4.15.4.13 total_renewable_dispatch_kWh

`double Model::total_renewable_dispatch_kWh`

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

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

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

4.16 ModelInputs Struct Reference

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

`#include <Model.h>`

Public Attributes

- std::string path_2_electrical_load_time_series = ""

  A string defining the path (either relative or absolute) to the given electrical load time series.
- ControlMode control_mode = ControlMode :: LOAD_FOLLOWING

  The control mode to be applied by the Controller object.

4.16.1 Detailed Description

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

4.16.2 Member Data Documentation

4.16.2.1 control_mode

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

The control mode to be applied by the Controller object.

4.16.2.2 path_2_electrical_load_time_series

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

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

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

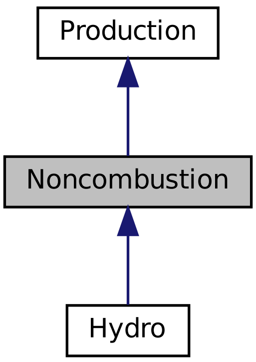
- header/Model.h

4.17 Noncombustion Class Reference

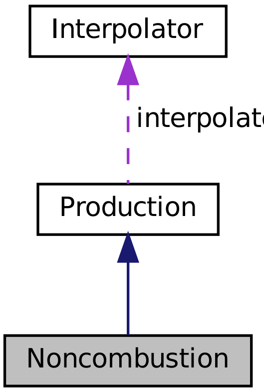
The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

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

Inheritance diagram for Noncombustion:



Collaboration diagram for Noncombustion:



Public Member Functions

- Noncombustion (void)

  Constructor (dummy) for the Noncombustion class.
- Noncombustion (int, double, NoncombustionInputs, std::vector< double > ∗)

  Constructor (intended) for the Noncombustion class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeEconomics (std::vector< double > ∗)

  Helper method to compute key economic metrics for the Model run.
- virtual double requestProductionkW (int, double, double)
- virtual double requestProductionkW (int, double, double, double)
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual double commit (int, double, double, double, double)
- void writeResults (std::string, std::vector< double > ∗, int, int=-1)

  Method which writes Noncombustion results to an output directory.
- virtual ∼Noncombustion (void)

  Destructor for the Noncombustion class.

Public Attributes

- NoncombustionType type

  The type (NoncombustionType) of the asset.
- int resource_key

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

Private Member Functions

- void __checkInputs (NoncombustionInputs)

    Helper method to check inputs to the Noncombustion constructor.
- void __handleStartStop (int, double, double)

    Helper method to handle the starting/stopping of the Noncombustion asset.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > *, int=-1)

4.17.1 Detailed Description

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

4.17.2 Constructor & Destructor Documentation

4.17.2.1 Noncombustion() [1/2]

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

Constructor (dummy) for the Noncombustion class.

```
{
    return;
}   /* Noncombustion() */
```

4.17.2.2 Noncombustion() [2/2]

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

Constructor (intended) for the Noncombustion class.

Parameters

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

```
  :
Production(
    n_points,
```

```
    n_years,
    noncombustion_inputs.production_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(noncombustion_inputs);

    //  2. set attributes
    //...

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Noncombustion object constructed at " « this « std::endl;
    }

    return;
}   /* Noncombustion() */
```

4.17.2.3 ∼Noncombustion()

```
Noncombustion::∼Noncombustion (
        void ) [virtual]
```

Destructor for the Noncombustion class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Noncombustion object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

noncombustion_inputs	A structure of Noncombustion constructor inputs.

```
{
    //...

    return;
}   /* __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.

```
{
    if (this->is_running) {
        // handle stopping
        if (production_kW <= 0) {
            this->is_running = false;
        }
    }

    else {
        // handle starting
        if (production_kW > 0) {
            this->is_running = true;
            this->n_starts++;
        }
    }

    return;
}   /* __handleStartStop() */
```

4.17.3.3 __writeSummary()

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

Reimplemented in Hydro.

```
{return;}
```

4.17.3.4 __writeTimeSeries()

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

Reimplemented in Hydro.

```
            {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

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

Returns

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

Reimplemented from Production.

```
{
    //  1. handle start/stop
    this->__handleStartStop(timestep, dt_hrs, production_kW);

    //  2. invoke base class method
    load_kW = Production :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

4.17.3.6 commit() [2/2]

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

Reimplemented in Hydro.

```
{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

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
{
    //  1. invoke base class method
    Production :: computeEconomics(time_vec_hrs_ptr);

    return;
}   /* computeEconomics() */
```

4.17.3.8 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Hydro.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Production :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.17.3.9 requestProductionkW() [1/2]

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

```
{return 0;}
```

4.17.3.10 requestProductionkW() [2/2]

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

Reimplemented in Hydro.

```
{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

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
noncombustion_index	An integer which corresponds to the index of the Noncombustion asset in the Model.
max_lines	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.

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Production/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += "Noncombustion/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->capacity_kW)));
    write_path += "kW_idx";
    write_path += std::to_string(combustion_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(write_path, time_vec_hrs_ptr, max_lines);
    }

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

4.17.4 Member Data Documentation

4.17.4.1 resource_key

```
int Noncombustion::resource_key
```

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

4.17.4.2 type

```
NoncombustionType Noncombustion::type
```

The type (NoncombustionType) of the asset.

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

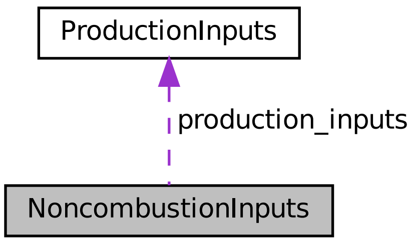
- header/Production/Noncombustion/Noncombustion.h
- source/Production/Noncombustion/Noncombustion.cpp

4.18 NoncombustionInputs Struct Reference

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

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

Collaboration diagram for NoncombustionInputs:



Public Attributes

- ProductionInputs production_inputs

    An encapsulated ProductionInputs instance.

4.18.1 Detailed Description

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

4.18.2 Member Data Documentation

4.18.2.1 production_inputs

`ProductionInputs NoncombustionInputs::production_inputs`

An encapsulated ProductionInputs instance.

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

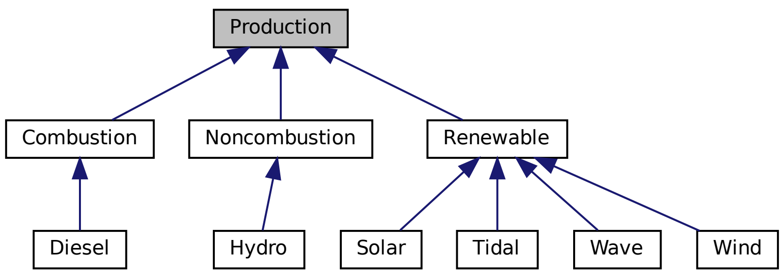
- header/Production/Noncombustion/Noncombustion.h

4.19 Production Class Reference

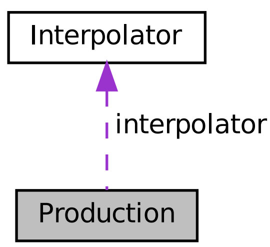
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

`#include <Production.h>`

Inheritance diagram for Production:



Collaboration diagram for Production:



Public Member Functions

- Production (void)

    Constructor (dummy) for the Production class.
- Production (int, double, ProductionInputs, std::vector< double > ∗)

    Constructor (intended) for the Production class.
- virtual void handleReplacement (int)

    Method to handle asset replacement and capital cost incursion, if applicable.
- double computeRealDiscountAnnual (double, double)

    Method to compute the real, annual discount rate to be used in computing model economics. This enables application of the discount factor approach.
- virtual void computeEconomics (std::vector< double > ∗)

    Helper method to compute key economic metrics for the Model run.
- double getProductionkW (int)

    A method to simply fetch the normalized production at a particular point in the given normalized production time series, multiply by the rated capacity of the asset, and return.
- virtual double commit (int, double, double, double)

    Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- virtual ∼Production (void)

    Destructor for the Production class.

Public Attributes

- Interpolator interpolator

    Interpolator component of Production.
- bool print_flag

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

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

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

    A boolen which indicates whether or not a normalized production time series is given.
- int n_points

    The number of points in the modelling time series.
- int n_starts

    The number of times the asset has been started.
- int n_replacements

    The number of times the asset has been replaced.
- double n_years

    The number of years being modelled.
- double running_hours

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

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

    The rated production capacity [kW] of the asset.
- double nominal_inflation_annual

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

- double nominal_discount_annual

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

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

  The capital cost of the asset (undefined currency).
- double operation_maintenance_cost_kWh

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

  The net present cost of this asset.
- double total_dispatch_kWh

  The total energy dispatched [kWh] over the Model run.
- double levellized_cost_of_energy_kWh

  The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatch.
- std::string type_str

  A string describing the type of the asset.
- std::string path_2_normalized_production_time_series

  A string defining the path (either relative or absolute) to the given normalized production time series.
- std::vector< bool > is_running_vec

  A boolean vector for tracking if the asset is running at a particular point in time.
- std::vector< double > normalized_production_vec

  A vector of normalizd production [ ] at each point in the modelling time series.
- std::vector< double > production_vec_kW

  A vector of production [kW] at each point in the modelling time series.
- std::vector< double > dispatch_vec_kW

  A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.
- std::vector< double > storage_vec_kW

  A vector of storage [kW] at each point in the modelling time series. Storage is the amount of production that is sent to storage.
- std::vector< double > curtailment_vec_kW

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

  A vector of capital costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are actual costs).
- std::vector< double > operation_maintenance_cost_vec

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

Private Member Functions

- void __checkInputs (int, double, ProductionInputs)

  Helper method to check inputs to the Production constructor.
- void __checkTimePoint (double, double)

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

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

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

- void __readNormalizedProductionData (std::vector< double > ∗)

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

4.19.1 Detailed Description

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

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Production() [1/2]

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

Constructor (dummy) for the Production class.

```
{
    return;
}   /* Production() */
```

4.19.2.2 Production() [2/2]

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

Constructor (intended) for the Production class.

Parameters

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

```
{
    //  1. check inputs
    this->__checkInputs(n_points, n_years, production_inputs);

    //  2. set attributes
    this->print_flag = production_inputs.print_flag;
    this->is_running = false;
    this->is_sunk = production_inputs.is_sunk;
    this->normalized_production_series_given = false;

```

```
    this->n_points = n_points;
    this->n_starts = 0;
    this->n_replacements = 0;

    this->n_years = n_years;

    this->running_hours = 0;
    this->replace_running_hrs = production_inputs.replace_running_hrs;

    this->capacity_kW = production_inputs.capacity_kW;

    this->nominal_inflation_annual = production_inputs.nominal_inflation_annual;
    this->nominal_discount_annual = production_inputs.nominal_discount_annual;

    this->real_discount_annual = this->computeRealDiscountAnnual(
        production_inputs.nominal_inflation_annual,
        production_inputs.nominal_discount_annual
    );

    this->capital_cost = 0;
    this->operation_maintenance_cost_kWh = 0;
    this->net_present_cost = 0;
    this->total_dispatch_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    this->path_2_normalized_production_time_series = "";

    this->is_running_vec.resize(this->n_points, 0);

    this->normalized_production_vec.resize(this->n_points, 0);
    this->production_vec_kW.resize(this->n_points, 0);
    this->dispatch_vec_kW.resize(this->n_points, 0);
    this->storage_vec_kW.resize(this->n_points, 0);
    this->curtailment_vec_kW.resize(this->n_points, 0);

    this->capital_cost_vec.resize(this->n_points, 0);
    this->operation_maintenance_cost_vec.resize(this->n_points, 0);

    //  3. read in normalized production time series (if given)
    if (not production_inputs.path_2_normalized_production_time_series.empty()) {
        this->normalized_production_series_given = true;

        this->path_2_normalized_production_time_series =
            production_inputs.path_2_normalized_production_time_series;

        this->__readNormalizedProductionData(time_vec_hrs_ptr);
    }

    //  4. construction print
    if (this->print_flag) {
        std::cout « "Production object constructed at " « this « std::endl;
    }

    return;
}   /* Production() */
```

4.19.2.3 ∼Production()

```
Production::∼Production (
              void  )  [virtual]
```

Destructor for the Production class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Production object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

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

```
{
    //  1. check n_points
    if (n_points <= 0) {
        std::string error_str = "ERROR:  Production():  n_points must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check n_years
    if (n_years <= 0) {
        std::string error_str = "ERROR:  Production():  n_years must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check capacity_kW
    if (production_inputs.capacity_kW <= 0) {
        std::string error_str = "ERROR:  Production():  ";
        error_str += "ProductionInputs::capacity_kW must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check replace_running_hrs
    if (production_inputs.replace_running_hrs <= 0) {
        std::string error_str = "ERROR:  Production():  ";
        error_str += "ProductionInputs::replace_running_hrs must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.19.3.2 __checkNormalizedProduction()

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

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

Parameters

normalized_production	The normalized production value to check

```
{
    if (normalized_production < 0 or normalized_production > 1) {
        std::string error_str = "ERROR:  Production():  ";
        error_str += "the given normalized production time series at ";
        error_str += this->path_2_normalized_production_time_series;
        error_str += " contains normalized production values outside the closed ";
        error_str += "interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::runtime_error(error_str);
    }

    return;
}   /* __throwValueError() */
```

4.19.3.3 __checkTimePoint()

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

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

Parameters

time_received_hrs	The point in time received from the given data.
time_expected_hrs	The point in time expected (this comes from the electrical load time series).

```
{
    if (time_received_hrs != time_expected_hrs) {
        std::string error_str = "ERROR:  Production():  ";
        error_str += "the given normalized production time series at ";
        error_str += this->path_2_normalized_production_time_series;
        error_str += " does not align with the ";
        error_str += "previously given electrical load time series";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::runtime_error(error_str);
    }

    return;
}   /* __checkTimePoint() */
```

4.19.3.4 __readNormalizedProductionData()

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

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

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    //  1. init CSV reader
    io::CSVReader<2> CSV(this->path_2_normalized_production_time_series);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Normalized Production [ ]"
    );

    //  2. read in normalized performance data,
    //     check values and check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double normalized_production = 0;

    while (CSV.read_row(time_hrs, normalized_production)) {
        //  2.1. check length of data
        if (n_points > this->n_points) {
            this->__throwLengthError();
        }

        //  2.2. check normalized production value
        this->__checkNormalizedProduction(normalized_production);

        //  2.3. check time point
        time_expected_hrs = time_vec_hrs_ptr->at(n_points);
        this->__checkTimePoint(time_hrs, time_expected_hrs);

        //  2.4. write to normalized production vector, increment n_points
        this->normalized_production_vec[n_points] = normalized_production;
        n_points++;
    }

    //  3. check length of data
    if (n_points != this->n_points) {
        this->__throwLengthError();
    }

    return;
}   /* __readNormalizedProductionData() */
```

4.19.3.5 __throwLengthError()

```
void Production::__throwLengthError (
            void ) [private]
```

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

```
{
    std::string error_str = "ERROR:  Production():  ";
    error_str += "the given normalized production time series at ";
    error_str += this->path_2_normalized_production_time_series;
    error_str += " is not the same length as the previously given electrical";
    error_str += " load time series";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);

    return;
}   /* __throwLengthError() */
```

4.19.3.6 commit()

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

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

Parameters

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

Returns

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

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

```
{
    //  1. record production
    this->production_vec_kW[timestep] = production_kW;

    //  2. compute and record dispatch and curtailment
    double dispatch_kW = 0;
    double curtailment_kW = 0;

    if (production_kW > load_kW) {
        dispatch_kW = load_kW;
        curtailment_kW = production_kW - dispatch_kW;
    }

    else {
        dispatch_kW = production_kW;
    }

    this->dispatch_vec_kW[timestep] = dispatch_kW;
    this->total_dispatch_kWh += dispatch_kW * dt_hrs;
    this->curtailment_vec_kW[timestep] = curtailment_kW;

    //  3. update load
    load_kW -= dispatch_kW;

    //  4. update and log running attributes
    if (this->is_running) {
        //  4.1. log running state, running hours
        this->is_running_vec[timestep] = this->is_running;
        this->running_hours += dt_hrs;

        //  4.2. incur operation and maintenance costs
        double produced_kWh = production_kW * dt_hrs;

        double operation_maintenance_cost =
            this->operation_maintenance_cost_kWh * produced_kWh;
        this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
    }

    //  5. trigger replacement, if applicable
    if (this->running_hours >= (this->n_replacements + 1) * this->replace_running_hrs) {
        this->handleReplacement(timestep);
    }

    return load_kW;
}   /* commit() */
```

4.19.3.7 computeEconomics()

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

Helper method to compute key economic metrics for the Model run.

Ref: HOMER [2023b]
Ref: HOMER [2023g]
Ref: HOMER [2023i]
Ref: HOMER [2023a]

Parameters

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

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

Reimplemented in Renewable, Noncombustion, and Combustion.

```
{
    //  1. compute net present cost
    double t_hrs = 0;
    double real_discount_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_discount_scalar = 1.0 / pow(
            1 + this->real_discount_annual,
            t_hrs / 8760
        );

        this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];

        this->net_present_cost +=
            real_discount_scalar * this->operation_maintenance_cost_vec[i];
    }

    //     assuming 8,760 hours per year
    if (this->total_dispatch_kWh <= 0) {
        this->levellized_cost_of_energy_kWh = this->net_present_cost;
    }

    else {
        double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;

        double capital_recovery_factor =
            (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
            (pow(1 + this->real_discount_annual, n_years) - 1);

        double total_annualized_cost = capital_recovery_factor *
            this->net_present_cost;

        this->levellized_cost_of_energy_kWh =
            (n_years * total_annualized_cost) /
            this->total_dispatch_kWh;
    }

    return;
}   /* computeEconomics() */
```

4.19.3.8 computeRealDiscountAnnual()

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

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

Ref: HOMER [2023h]
Ref: HOMER [2023b]

Parameters

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

Returns

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

```
{
    double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
    real_discount_annual /= 1 + nominal_inflation_annual;

    return real_discount_annual;
}   /* __computeRealDiscountAnnual() */
```

4.19.3.9 getProductionkW()

```
double Production::getProductionkW (
            int timestep )
```

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

Returns

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

```
{
    double production_kW =
        this->normalized_production_vec[timestep] * this->capacity_kW;

    return production_kW;
}   /* getProductionkW() */
```

4.19.3.10 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

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

```
{
    //  1. reset attributes
    this->is_running = false;

    //  2. log replacement
    this->n_replacements++;

    //  3. incur capital cost in timestep
    this->capital_cost_vec[timestep] = this->capital_cost;

    return;
}   /* __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 levellized_cost_of_energy_kWh

`double Production::levellized_cost_of_energy_kWh`

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

4.19.4.11 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.19.4.12 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.19.4.13 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.19.4.14 n_years

```
double Production::n_years
```

The number of years being modelled.

4.19.4.15 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.19.4.16 nominal_discount_annual

```
double Production::nominal_discount_annual
```

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

4.19.4.17 nominal_inflation_annual

```
double Production::nominal_inflation_annual
```

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

4.19.4.18 normalized_production_series_given

```
bool Production::normalized_production_series_given
```

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

4.19.4.19 normalized_production_vec

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

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

4.19.4.20 operation_maintenance_cost_kWh

```
double Production::operation_maintenance_cost_kWh
```

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

4.19.4.21 operation_maintenance_cost_vec

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

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

4.19.4.22 path_2_normalized_production_time_series

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

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

4.19.4.23 print_flag

```
bool Production::print_flag
```

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

4.19.4.24 production_vec_kW

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

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

4.19.4.25 real_discount_annual

```
double Production::real_discount_annual
```

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

4.19.4.26 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.19.4.27 running_hours

```
double Production::running_hours
```

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

4.19.4.28 storage_vec_kW

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

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

4.19.4.29 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

4.19.4.30 type_str

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

A string describing the type of the asset.

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

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

4.20 ProductionInputs Struct Reference

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

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

Public Attributes

- bool print_flag = false
  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_sunk = false
  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double capacity_kW = 100
  The rated production capacity [kW] of the asset.
- double nominal_inflation_annual = 0.02
  The nominal, annual inflation rate to use in computing model economics.
- double nominal_discount_annual = 0.04
  The nominal, annual discount rate to use in computing model economics.
- double replace_running_hrs = 90000
  The number of running hours after which the asset must be replaced.
- std::string path_2_normalized_production_time_series = ""
  A string defining the path (either relative or absolute) to the given normalized production time series.

4.20.1 Detailed Description

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

4.20.2 Member Data Documentation

4.20.2.1 capacity_kW

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

The rated production capacity [kW] of the asset.

4.20.2.2 is_sunk

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

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

4.20.2.3 nominal_discount_annual

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

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

4.20.2.4 nominal_inflation_annual

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

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

4.20.2.5 path_2_normalized_production_time_series

```
std::string ProductionInputs::path_2_normalized_production_time_series = ""
```

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

4.20.2.6 print_flag

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

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

4.20.2.7 replace_running_hrs

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

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

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

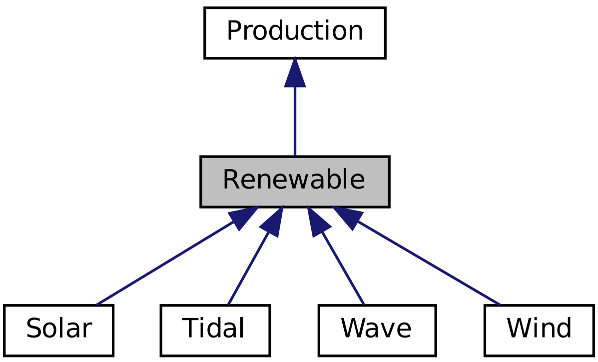
- header/Production/Production.h

4.21 Renewable Class Reference

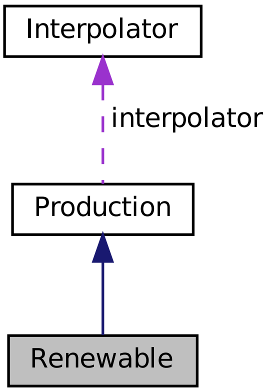
The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy.

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

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



Public Member Functions

- Renewable (void)

  Constructor (dummy) for the Renewable class.
- Renewable (int, double, RenewableInputs, std::vector< double > ∗)

  Constructor (intended) for the Renewable class.
- virtual void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- void computeEconomics (std::vector< double > ∗)

  Helper method to compute key economic metrics for the Model run.
- virtual double computeProductionkW (int, double, double)
- virtual double computeProductionkW (int, double, double, double)
- virtual double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- void writeResults (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std::map< int, std::vector< std::vector< double >>> ∗, int, int=-1)

  Method which writes Renewable results to an output directory.
- virtual ∼Renewable (void)

  Destructor for the Renewable class.

Public Attributes

- RenewableType type

  The type (RenewableType) of the asset.
- int resource_key

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

Private Member Functions

- void __checkInputs (RenewableInputs)

    Helper method to check inputs to the Renewable constructor.
- void __handleStartStop (int, double, double)

    Helper method to handle the starting/stopping of the renewable asset.
- virtual void __writeSummary (std::string)
- virtual void __writeTimeSeries (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std::map< int, std::vector< std::vector< double >>> ∗, int=-1)

4.21.1 Detailed Description

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

4.21.2 Constructor & Destructor Documentation

4.21.2.1 Renewable() [1/2]

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

Constructor (dummy) for the Renewable class.

```
{
    //...

    return;
}   /* Renewable() */
```

4.21.2.2 Renewable() [2/2]

```
Renewable::Renewable (
            int n_points,
            double n_years,
            RenewableInputs renewable_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Renewable class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
renewable_inputs	A structure of Renewable constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
    :
Production(
    n_points,
    n_years,
    renewable_inputs.production_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(renewable_inputs);

    //  2. set attributes
    //...

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Renewable object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

4.21.2.3 ∼Renewable()

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

Destructor for the Renewable class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Renewable object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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.

```
{
    //...

    return;
}   /* __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.

```
{
    if (this->is_running) {
        // handle stopping
        if (production_kW <= 0) {
            this->is_running = false;
        }
    }

    else {
        // handle starting
        if (production_kW > 0) {
            this->is_running = true;
            this->n_starts++;
        }
    }

    return;
}   /* __handleStartStop() */
```

4.21.3.3 __writeSummary()

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

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{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.

```
        {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

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

Returns

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

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{
    // 1. handle start/stop
    this->__handleStartStop(timestep, dt_hrs, production_kW);

    // 2. invoke base class method
    load_kW = Production :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* 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

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

Reimplemented from Production.

```
{
    // 1. invoke base class method
    Production :: computeEconomics(time_vec_hrs_ptr);

    return;
}   /* computeEconomics() */
```

4.21.3.7 computeProductionkW() [1/2]

```
virtual double Renewable::computeProductionkW (
            int ,
```

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

Reimplemented in Wind, Tidal, and Solar.

```
{return 0;}
```

4.21.3.8 computeProductionkW() [2/2]

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

Reimplemented in Wave.

```
{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

timestep	The current time step of the Model run.

Reimplemented from Production.

Reimplemented in Wind, Wave, Tidal, and Solar.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Production :: handleReplacement(timestep);

    return;
}   /* __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

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
resource_map_1D_ptr	A pointer to the 1D map of Resources.
resource_map_2D_ptr	A pointer to the 2D map of Resources.
renewable_index	An integer which corresponds to the index of the Renewable asset in the Model.
max_lines	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.

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Production/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += "Renewable/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->capacity_kW)));
    write_path += "kW_idx";
    write_path += std::to_string(renewable_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(
            write_path,
            time_vec_hrs_ptr,
            resource_map_1D_ptr,
            resource_map_2D_ptr,
            max_lines
        );
    }

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

4.21.4 Member Data Documentation

4.21.4.1 resource_key

```
int Renewable::resource_key
```

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

4.21.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

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

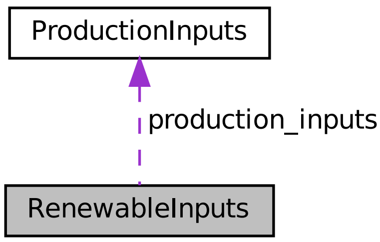
- header/Production/Renewable/Renewable.h
- source/Production/Renewable/Renewable.cpp

4.22 RenewableInputs Struct Reference

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

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

Collaboration diagram for RenewableInputs:



Public Attributes

- ProductionInputs production_inputs

  An encapsulated ProductionInputs instance.

4.22.1 Detailed Description

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

4.22.2 Member Data Documentation

4.22.2.1 production_inputs

```
ProductionInputs RenewableInputs::production_inputs
```

An encapsulated ProductionInputs instance.

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

- header/Production/Renewable/Renewable.h

4.23 Resources Class Reference

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

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

Public Member Functions

- Resources (void)

  Constructor for the Resources class.
- void addResource (NoncombustionType, std::string, int, ElectricalLoad ∗)

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

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

  Method to clear all attributes of the Resources object.
- ∼Resources (void)

  Destructor for the Resources class.

Public Attributes

- std::map< int, std::vector< double > > resource_map_1D

  A map <int, vector<double>> of given 1D renewable resource time series.
- std::map< int, std::string > string_map_1D

  A map <int, string> of descriptors for the type of the given 1D renewable resource time series.
- std::map< int, std::string > path_map_1D

  A map <int, string> of the paths (either relative or absolute) to given 1D renewable resource time series.
- std::map< int, std::vector< std::vector< double > > > resource_map_2D

  A map <int, vector<vector<double>>> of given 2D renewable resource time series.
- std::map< int, std::string > string_map_2D

  A map <int, string> of descriptors for the type of the given 2D renewable resource time series.
- std::map< int, std::string > path_map_2D

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

Private Member Functions

- void __checkResourceKey1D (int, RenewableType)

  Helper method to check if given resource key (1D) is already in use.
- void __checkResourceKey2D (int, RenewableType)

  Helper method to check if given resource key (2D) is already in use.
- void __checkResourceKey1D (int, NoncombustionType)

  Helper method to check if given resource key (1D) is already in use.
- void __checkTimePoint (double, double, std::string, ElectricalLoad ∗)

  Helper method to check received time point against expected time point. The given time series should align point-wise with the previously given electrical load time series.
- void __throwLengthError (std::string, ElectricalLoad ∗)

  Helper method to throw data length error (if not the same as the given electrical load time series).
- void __readHydroResource (std::string, int, ElectricalLoad ∗)

  Helper method to handle reading a hydro resource time series into Resources.
- void __readSolarResource (std::string, int, ElectricalLoad ∗)

  Helper method to handle reading a solar resource time series into Resources.
- void __readTidalResource (std::string, int, ElectricalLoad ∗)

  Helper method to handle reading a tidal resource time series into Resources.
- void __readWaveResource (std::string, int, ElectricalLoad ∗)

  Helper method to handle reading a wave resource time series into Resources.
- void __readWindResource (std::string, int, ElectricalLoad ∗)

  Helper method to handle reading a wind resource time series into Resources.

4.23.1 Detailed Description

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

4.23.2 Constructor & Destructor Documentation

4.23.2.1 Resources()

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

Constructor for the Resources class.

```
{
    return;
}   /* Resources() */
```

4.23.2.2 ∼Resources()

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

Destructor for the Resources class.

```
{
    this->clear();
    return;
}   /* ~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

resource_key	The key associated with the given renewable resource.
noncombustion_type	The type of renewable resource being added to Resources.

```
{
    if (this->resource_map_1D.count(resource_key) > 0) {
        std::string error_str = "ERROR:  Resources::addResource(";

        switch (noncombustion_type) {
            case (NoncombustionType :: HYDRO): {
                error_str += "HYDRO):  ";

                break;
            }

            default: {
                error_str += "UNDEFINED_TYPE):  ";

                break;
            }
        }

        error_str += "resource key (1D) ";
        error_str += std::to_string(resource_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /*  __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

resource_key	The key associated with the given renewable resource.
renewable_type	The type of renewable resource being added to Resources.

```
{
    if (this->resource_map_1D.count(resource_key) > 0) {
        std::string error_str = "ERROR:  Resources::addResource(";

        switch (renewable_type) {
            case (RenewableType :: SOLAR): {
                error_str += "SOLAR):  ";

                break;
            }

            case (RenewableType :: TIDAL): {
                error_str += "TIDAL):  ";

                break;
            }

            case (RenewableType :: WIND): {
                error_str += "WIND):  ";

                break;
            }

            default: {
                error_str += "UNDEFINED_TYPE):  ";

                break;
            }
        }

        error_str += "resource key (1D) ";
        error_str += std::to_string(resource_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

resource_key	The key associated with the given renewable resource.

```
{
    if (this->resource_map_2D.count(resource_key) > 0) {
        std::string error_str = "ERROR:  Resources::addResource(";

        switch (renewable_type) {
            case (RenewableType :: WAVE): {
                error_str += "WAVE):  ";

                break;
            }

            default: {
                error_str += "UNDEFINED_TYPE):  ";

                break;
            }
        }

```

```
        error_str += "resource key (2D) ";
        error_str += std::to_string(resource_key);
        error_str += " is already in use";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /*  __checkResourceKey2D() */
```

4.23.3.4 __checkTimePoint()

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

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

Parameters

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

```
{
    if (time_received_hrs != time_expected_hrs) {
        std::string error_str = "ERROR:  Resources::addResource():  ";
        error_str += "the given resource time series at ";
        error_str += path_2_resource_data;
        error_str += " does not align with the ";
        error_str += "previously given electrical load time series at ";
        error_str += electrical_load_ptr->path_2_electrical_load_time_series;

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::runtime_error(error_str);
    }

    return;
}   /* __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

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

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Hydro Inflow [m3/hr]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "HYDRO"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double hydro_resource_m3hr = 0;

    while (CSV.read_row(time_hrs, hydro_resource_m3hr)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = hydro_resource_m3hr;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __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

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

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Solar GHI [kW/m2]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "SOLAR"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double>>(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double solar_resource_kWm2 = 0;

    while (CSV.read_row(time_hrs, solar_resource_kWm2)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __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

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

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Tidal Speed (hub depth) [m/s]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "TIDAL"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double tidal_resource_ms = 0;

    while (CSV.read_row(time_hrs, tidal_resource_ms)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readTidalResource() */
```

4.23.3.8 __readWaveResource()

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

Helper method to handle reading a wave resource time series into Resources.

Parameters

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

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<3> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Significant Wave Height [m]",
        "Energy Period [s]"
    );

    this->path_map_2D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_2D.insert(std::pair<int, std::string>(resource_key, "WAVE"));

    //  2. init map element
    this->resource_map_2D.insert(
        std::pair<int, std::vector<std::vector<double>>>(resource_key, {})
    );
    this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double significant_wave_height_m = 0;
    double energy_period_s = 0;

    while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
        this->resource_map_2D[resource_key][n_points][1] = energy_period_s;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __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

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

```
{
    //  1. init CSV reader, record path and type
    io::CSVReader<2> CSV(path_2_resource_data);

    CSV.read_header(
        io::ignore_extra_column,
        "Time (since start of data) [hrs]",
        "Wind Speed (hub height) [m/s]"
    );

    this->path_map_1D.insert(
        std::pair<int, std::string>(resource_key, path_2_resource_data)
    );

    this->string_map_1D.insert(std::pair<int, std::string>(resource_key, "WIND"));

    //  2. init map element
    this->resource_map_1D.insert(
        std::pair<int, std::vector<double»(resource_key, {})
    );
    this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);


    //  3. read in resource data, check against time series (point-wise and length)
    int n_points = 0;
    double time_hrs = 0;
    double time_expected_hrs = 0;
    double wind_resource_ms = 0;

    while (CSV.read_row(time_hrs, wind_resource_ms)) {
        if (n_points > electrical_load_ptr->n_points) {
            this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
        }

        time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
        this->__checkTimePoint(
            time_hrs,
            time_expected_hrs,
            path_2_resource_data,
            electrical_load_ptr
        );

        this->resource_map_1D[resource_key][n_points] = wind_resource_ms;

        n_points++;
    }

    //  4. check data length
    if (n_points != electrical_load_ptr->n_points) {
        this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
    }

    return;
}   /* __readWindResource() */
```

4.23.3.10 __throwLengthError()

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

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

Parameters

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

```
{
    std::string error_str = "ERROR:  Resources::addResource():  ";
    error_str += "the given resource time series at ";
    error_str += path_2_resource_data;
    error_str += " is not the same length as the previously given electrical";
    error_str += " load time series at ";
    error_str += electrical_load_ptr->path_2_electrical_load_time_series;

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);

    return;
}   /* __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

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

```
{
    switch (noncombustion_type) {
        case (NoncombustionType :: HYDRO): {
            this->__checkResourceKey1D(resource_key, noncombustion_type);

            this->__readHydroResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Resources :: addResource(:  ";
            error_str += "noncombustion type ";
            error_str += std::to_string(noncombustion_type);
            error_str += " has no associated resource";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return;
```

```
}   /* 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

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

```
{
    switch (renewable_type) {
        case (RenewableType :: SOLAR): {
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readSolarResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: TIDAL): {
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readTidalResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: WAVE): {
            this->__checkResourceKey2D(resource_key, renewable_type);

            this->__readWaveResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );

            break;
        }

        case (RenewableType :: WIND): {
            this->__checkResourceKey1D(resource_key, renewable_type);

            this->__readWindResource(
                path_2_resource_data,
                resource_key,
                electrical_load_ptr
            );
```

```
            break;
        }

        default: {
            std::string error_str = "ERROR:  Resources :: addResource(:  ";
            error_str += "renewable type ";
            error_str += std::to_string(renewable_type);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return;
}   /* addResource() */
```

4.23.3.13 clear()

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

Method to clear all attributes of the Resources object.

```
{
    this->resource_map_1D.clear();
    this->string_map_1D.clear();
    this->path_map_1D.clear();

    this->resource_map_2D.clear();
    this->string_map_2D.clear();
    this->path_map_2D.clear();

    return;
}   /* clear() */
```

4.23.4 Member Data Documentation

4.23.4.1 path_map_1D

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

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

4.23.4.2 path_map_2D

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

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

4.23.4.3 resource_map_1D

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

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

4.23.4.4 resource_map_2D

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

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

4.23.4.5 string_map_1D

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

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

4.23.4.6 string_map_2D

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

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

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

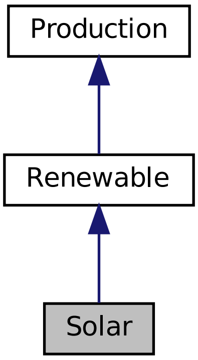
- header/Resources.h
- source/Resources.cpp

4.24 Solar Class Reference

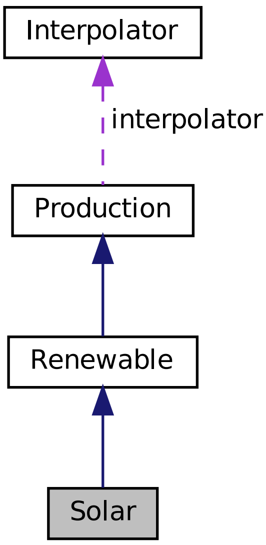
A derived class of the Renewable branch of Production which models solar production.

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

Inheritance diagram for Solar:



Collaboration diagram for Solar:



Public Member Functions

- Solar (void)

    Constructor (dummy) for the Solar class.
- Solar (int, double, SolarInputs, std::vector< double > ∗)

    Constructor (intended) for the Solar class.
- void handleReplacement (int)

    Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

    Method which takes in the solar resource at a particular point in time, and then returns the solar PV production at that point in time.
- double commit (int, double, double, double)

    Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ∼Solar (void)

    Destructor for the Solar class.

Public Attributes

- double derating

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

Private Member Functions

- void __checkInputs (SolarInputs)

    Helper method to check inputs to the Solar constructor.
- double __getGenericCapitalCost (void)

    Helper method to generate a generic solar PV array capital cost.
- double __getGenericOpMaintCost (void)

    Helper method to generate a generic solar PV array operation and maintenance cost. This is a cost incurred per unit energy produced.
- void __writeSummary (std::string)

    Helper method to write summary results for Solar.
- void __writeTimeSeries (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std↩ ::map< int, std::vector< std::vector< double >>> ∗, int=-1)

    Helper method to write time series results for Solar.

4.24.1 Detailed Description

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

4.24.2 Constructor & Destructor Documentation

4.24.2.1 Solar() [1/2]

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

Constructor (dummy) for the Solar class.

```
{
    //...

    return;
}   /* Solar() */
```

4.24.2.2 Solar() [2/2]

```
Solar::Solar (
            int n_points,
            double n_years,
            SolarInputs solar_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Solar class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
solar_inputs	A structure of Solar constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Renewable(
    n_points,
    n_years,
    solar_inputs.renewable_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(solar_inputs);

    //  2. set attributes
    this->type = RenewableType :: SOLAR;
    this->type_str = "SOLAR";

    this->resource_key = solar_inputs.resource_key;

    this->derating = solar_inputs.derating;

    if (solar_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = solar_inputs.capital_cost;
    }

    if (solar_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            solar_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

```

```
    //  3. construction print
    if (this->print_flag) {
        std::cout « "Solar object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

4.24.2.3 ∼Solar()

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

Destructor for the Solar class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Solar object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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.

```
{
    //  1. check derating
    if (
        solar_inputs.derating < 0 or
        solar_inputs.derating > 1
    ) {
        std::string error_str = "ERROR:  Solar():  ";
        error_str += "SolarInputs::derating must be in the closed interval [0, 1]";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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].

```
{
    double capital_cost_per_kW = 1000 * pow(this->capacity_kW, -0.15) + 3000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __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].

```
{
    return 0.01;
}   /* __getGenericOpMaintCost() */
```

4.24.3.4 __writeSummary()

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

Helper method to write summary results for Solar.

Parameters

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW SOLAR Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « " kW  \n";
    ofs « "\n";

    ofs « "Production Override: (N = 0 / Y = 1): "
        « this->normalized_production_series_given « "  \n";
    if (this->normalized_production_series_given) {
        ofs « "Path to Normalized Production Time Series: "
            « this->path_2_normalized_production_time_series « "  \n";
    }
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Solar attributes
    ofs « "## Solar Attributes\n";
    ofs « "\n";

    ofs « "Derating Factor: " « this->derating « "  \n";

    ofs « "\n--------\n\n";

    //  2.4. Solar Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();
    return;
}   /* __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

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Solar Resource [kW/m2],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";

        if (not this->normalized_production_series_given) {
            ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        }

        else {
            ofs « "OVERRIDE" « ",";
        }

        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    ofs.close();
    return;
}   /* __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

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

Returns

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* 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

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

Returns

The production [kW] of the solar PV array.

Reimplemented from Renewable.

```
{
    //  given production time series override
    if (this->normalized_production_series_given) {
        double production_kW = Production :: getProductionkW(timestep);

        return production_kW;
    }

    // check if no resource
    if (solar_resource_kWm2 <= 0) {
        return 0;
    }

    // compute production
    double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;

    // cap production at capacity
    if (production_kW > this->capacity_kW) {
        production_kW = this->capacity_kW;
    }

    return production_kW;
}   /* computeProductionkW() */
```

4.24.3.8 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __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:

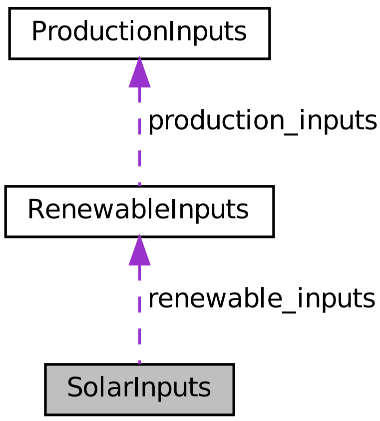
- header/Production/Renewable/Solar.h
- source/Production/Renewable/Solar.cpp

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:

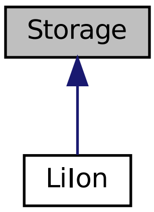
- header/Production/Renewable/Solar.h

4.26 Storage Class Reference

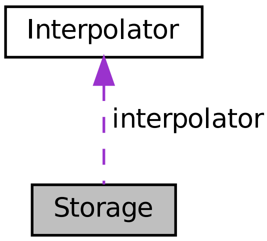
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

`#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.

```
{
    return;
}   /* 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

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
storage_inputs	A structure of Storage constructor inputs.

```
{
    //  1. check inputs
    this->__checkInputs(n_points, n_years, storage_inputs);

    //  2. set attributes
    this->print_flag = storage_inputs.print_flag;
    this->is_depleted = false;
    this->is_sunk = storage_inputs.is_sunk;

    this->n_points = n_points;
    this->n_replacements = 0;

    this->n_years = n_years;

    this->power_capacity_kW = storage_inputs.power_capacity_kW;
    this->energy_capacity_kWh = storage_inputs.energy_capacity_kWh;

    this->charge_kWh = 0;
    this->power_kW = 0;

    this->nominal_inflation_annual = storage_inputs.nominal_inflation_annual;
    this->nominal_discount_annual = storage_inputs.nominal_discount_annual;

    this->real_discount_annual = this->__computeRealDiscountAnnual(
        storage_inputs.nominal_inflation_annual,
        storage_inputs.nominal_discount_annual
    );

    this->capital_cost = 0;
    this->operation_maintenance_cost_kWh = 0;
    this->net_present_cost = 0;
    this->total_discharge_kWh = 0;
    this->levellized_cost_of_energy_kWh = 0;

    this->charge_vec_kWh.resize(this->n_points, 0);
    this->charging_power_vec_kW.resize(this->n_points, 0);
    this->discharging_power_vec_kW.resize(this->n_points, 0);

    this->capital_cost_vec.resize(this->n_points, 0);
    this->operation_maintenance_cost_vec.resize(this->n_points, 0);

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Storage object constructed at " « this « std::endl;
    }

    return;
}   /* Storage() */
```

4.26.2.3 ∼Storage()

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

Destructor for the Storage class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Storage object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

n_points	The number of points in the modelling time series.
storage_inputs	A structure of Storage constructor inputs.

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

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check n_years
    if (n_years <= 0) {
        std::string error_str = "ERROR:  Storage():  n_years must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. check power_capacity_kW
    if (storage_inputs.power_capacity_kW <= 0) {
        std::string error_str = "ERROR:  Storage():  ";
        error_str += "StorageInputs::power_capacity_kW must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  4. check energy_capacity_kWh
    if (storage_inputs.energy_capacity_kWh <= 0) {
        std::string error_str = "ERROR:  Storage():  ";
        error_str += "StorageInputs::energy_capacity_kWh must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

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

Returns

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

```
{
    double real_discount_annual = nominal_discount_annual - nominal_inflation_annual;
    real_discount_annual /= 1 + nominal_inflation_annual;

    return real_discount_annual;
}   /* __computeRealDiscountAnnual() */
```

4.26.3.3 __writeSummary()

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

Reimplemented in LiIon.

```
{return;}
```

4.26.3.4 __writeTimeSeries()

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

Reimplemented in LiIon.

```
{return;}
```

4.26.3.5 commitCharge()

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

Reimplemented in LiIon.

```
{return;}
```

4.26.3.6 commitDischarge()

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

Reimplemented in LiIon.

```
{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

time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.

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

```
{
    //  1. compute net present cost
    double t_hrs = 0;
    double real_discount_scalar = 0;

    for (int i = 0; i < this->n_points; i++) {
        t_hrs = time_vec_hrs_ptr->at(i);

        real_discount_scalar = 1.0 / pow(
            1 + this->real_discount_annual,
            t_hrs / 8760
        );

        this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];

        this->net_present_cost +=
            real_discount_scalar * this->operation_maintenance_cost_vec[i];
    }

    //      assuming 8,760 hours per year
    if (this->total_discharge_kWh <= 0) {
        this->levellized_cost_of_energy_kWh = this->net_present_cost;
    }

    else {
        double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;

        double capital_recovery_factor =
            (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
            (pow(1 + this->real_discount_annual, n_years) - 1);

        double total_annualized_cost = capital_recovery_factor *
            this->net_present_cost;
```

```

        this->levellized_cost_of_energy_kWh =
            (n_years * total_annualized_cost) /
            this->total_discharge_kWh;
    }

    return;
}   /* computeEconomics() */
```

4.26.3.8 getAcceptablekW()

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

Reimplemented in LiIon.

```
{return 0;}
```

4.26.3.9 getAvailablekW()

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

Reimplemented in LiIon.

```
{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

timestep	The current time step of the Model run.

Reimplemented in LiIon.

```
{
    //  1. reset attributes
    this->charge_kWh = 0;
    this->power_kW = 0;

    //  2. log replacement
    this->n_replacements++;

    //  3. incur capital cost in timestep
    this->capital_cost_vec[timestep] = this->capital_cost;

    return;
}   /* __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

write_path	A path (either relative or absolute) to the directory location where results are to be written. If already exists, will overwrite.
time_vec_hrs_ptr	A pointer to the time_vec_hrs attribute of the ElectricalLoad.
storage_index	An integer which corresponds to the index of the Storage asset in the Model.
max_lines	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.

```
{
    //  1. handle sentinel
    if (max_lines < 0) {
        max_lines = this->n_points;
    }

    //  2. create subdirectories
    write_path += "Storage/";
    if (not std::filesystem::is_directory(write_path)) {
        std::filesystem::create_directory(write_path);
    }

    write_path += this->type_str;
    write_path += "_";
    write_path += std::to_string(int(ceil(this->power_capacity_kW)));
    write_path += "kW_";
    write_path += std::to_string(int(ceil(this->energy_capacity_kWh)));
    write_path += "kWh_idx";
    write_path += std::to_string(storage_index);
    write_path += "/";
    std::filesystem::create_directory(write_path);

    //  3. write summary
    this->__writeSummary(write_path);

    //  4. write time series
    if (max_lines > this->n_points) {
        max_lines = this->n_points;
    }

    if (max_lines > 0) {
        this->__writeTimeSeries(
            write_path,
            time_vec_hrs_ptr,
            max_lines
        );
    }

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

4.26.4 Member Data Documentation

4.26.4.1 capital_cost

```
double Storage::capital_cost
```

The capital cost of the asset (undefined currency).

4.26.4.2 capital_cost_vec

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

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

4.26.4.3 charge_kWh

```
double Storage::charge_kWh
```

The energy [kWh] stored in the asset.

4.26.4.4 charge_vec_kWh

```
std::vector<double> Storage::charge_vec_kWh
```

A vector of the charge state [kWh] at each point in the modelling time series.

4.26.4.5 charging_power_vec_kW

```
std::vector<double> Storage::charging_power_vec_kW
```

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

4.26.4.6 discharging_power_vec_kW

```
std::vector<double> Storage::discharging_power_vec_kW
```

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

4.26.4.7 energy_capacity_kWh

```
double Storage::energy_capacity_kWh
```

The rated energy capacity [kWh] of the asset.

4.26.4.8 interpolator

```
Interpolator Storage::interpolator
```

Interpolator component of Storage.

4.26.4.9 is_depleted

```
bool Storage::is_depleted
```

A boolean which indicates whether or not the asset is currently considered depleted.

4.26.4.10 is_sunk

```
bool Storage::is_sunk
```

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

4.26.4.11 levellized_cost_of_energy_kWh

```
double Storage::levellized_cost_of_energy_kWh
```

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

4.26.4.12 n_points

```
int Storage::n_points
```

The number of points in the modelling time series.

4.26.4.13 n_replacements

```
int Storage::n_replacements
```

The number of times the asset has been replaced.

4.26.4.14 n_years

```
double Storage::n_years
```

The number of years being modelled.

4.26.4.15 net_present_cost

```
double Storage::net_present_cost
```

The net present cost of this asset.

4.26.4.16 nominal_discount_annual

```
double Storage::nominal_discount_annual
```

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

4.26.4.17 nominal_inflation_annual

```
double Storage::nominal_inflation_annual
```

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

4.26.4.18 operation_maintenance_cost_kWh

```
double Storage::operation_maintenance_cost_kWh
```

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

4.26.4.19 operation_maintenance_cost_vec

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

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

4.26.4.20 power_capacity_kW

```
double Storage::power_capacity_kW
```

The rated power capacity [kW] of the asset.

4.26.4.21 power_kW

```
double Storage::power_kW
```

The power [kW] currently being charged/discharged by the asset.

4.26.4.22 print_flag

```
bool Storage::print_flag
```

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

4.26.4.23 real_discount_annual

```
double Storage::real_discount_annual
```

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

4.26.4.24 total_discharge_kWh

```
double Storage::total_discharge_kWh
```

The total energy discharged [kWh] over the Model run.

4.26.4.25 type

`StorageType Storage::type`

The type (StorageType) of the asset.

4.26.4.26 type_str

`std::string Storage::type_str`

A string describing the type of the asset.

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

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

4.27 StorageInputs Struct Reference

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

`#include <Storage.h>`

Public Attributes

- bool print_flag = false

  A flag which indicates whether or not object construct/destruction should be verbose.
- bool is_sunk = false

  A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double power_capacity_kW = 100

  The rated power capacity [kW] of the asset.
- double energy_capacity_kWh = 1000

  The rated energy capacity [kWh] of the asset.
- double nominal_inflation_annual = 0.02

  The nominal, annual inflation rate to use in computing model economics.
- double nominal_discount_annual = 0.04

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

4.27.1 Detailed Description

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

4.27.2 Member Data Documentation

4.27.2.1 energy_capacity_kWh

```
double StorageInputs::energy_capacity_kWh = 1000
```

The rated energy capacity [kWh] of the asset.

4.27.2.2 is_sunk

```
bool StorageInputs::is_sunk = false
```

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

4.27.2.3 nominal_discount_annual

```
double StorageInputs::nominal_discount_annual = 0.04
```

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

4.27.2.4 nominal_inflation_annual

```
double StorageInputs::nominal_inflation_annual = 0.02
```

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

4.27.2.5 power_capacity_kW

```
double StorageInputs::power_capacity_kW = 100
```

The rated power capacity [kW] of the asset.

4.27.2.6 print_flag

```
bool StorageInputs::print_flag = false
```

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

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

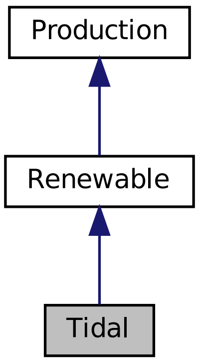
- header/Storage/Storage.h

4.28 Tidal Class Reference

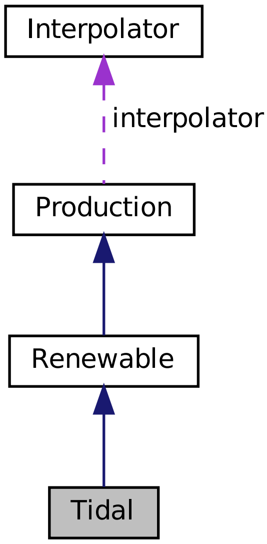
A derived class of the Renewable branch of Production which models tidal production.

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

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- Tidal (void)

  Constructor (dummy) for the Tidal class.
- Tidal (int, double, TidalInputs, std::vector< double > ∗)

  Constructor (intended) for the Tidal class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

  Method which takes in the tidal resource at a particular point in time, and then returns the tidal turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ∼Tidal (void)

  Destructor for the Tidal class.

Public Attributes

- double design_speed_ms

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

  The tidal power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (TidalInputs)

    Helper method to check inputs to the Tidal constructor.
- double __getGenericCapitalCost (void)

    Helper method to generate a generic tidal turbine capital cost.
- double __getGenericOpMaintCost (void)

    Helper method to generate a generic tidal turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeCubicProductionkW (int, double, double)

    Helper method to compute tidal turbine production under a cubic production model.
- double __computeExponentialProductionkW (int, double, double)

    Helper method to compute tidal turbine production under an exponential production model.
- double __computeLookupProductionkW (int, double, double)

    Helper method to compute tidal turbine production by way of looking up using given power curve data.
- void __writeSummary (std::string)

    Helper method to write summary results for Tidal.
- void __writeTimeSeries (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std::map< int, std::vector< std::vector< double >>> ∗, int=-1)

    Helper method to write time series results for Tidal.

4.28.1 Detailed Description

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

4.28.2 Constructor & Destructor Documentation

4.28.2.1 Tidal() [1/2]

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

Constructor (dummy) for the Tidal class.

```
{
    return;
}   /* Tidal() */
```

4.28.2.2 Tidal() [2/2]

```
Tidal::Tidal (
            int n_points,
            double n_years,
            TidalInputs tidal_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Tidal class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
tidal_inputs	A structure of Tidal constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Renewable(
    n_points,
    n_years,
    tidal_inputs.renewable_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(tidal_inputs);

    //  2. set attributes
    this->type = RenewableType :: TIDAL;
    this->type_str = "TIDAL";

    this->resource_key = tidal_inputs.resource_key;

    this->design_speed_ms = tidal_inputs.design_speed_ms;

    this->power_model = tidal_inputs.power_model;

    switch (this->power_model) {
        case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
            this->power_model_string = "CUBIC";

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
            this->power_model_string = "EXPONENTIAL";

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Tidal():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    if (tidal_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = tidal_inputs.capital_cost;
    }

    if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            tidal_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
```

```
    if (this->print_flag) {
        std::cout « "Tidal object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

4.28.2.3 ∼Tidal()

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

Destructor for the Tidal class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Tidal object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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.

```
{
    //  1. check design_speed_ms
    if (tidal_inputs.design_speed_ms <= 0) {
        std::string error_str = "ERROR:  Tidal():  ";
        error_str += "TidalInputs::design_speed_ms must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

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

Returns

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

```
{
    double production = 0;

    if (
        tidal_resource_ms < 0.15 * this->design_speed_ms or
        tidal_resource_ms > 1.25 * this->design_speed_ms
    ){
        production = 0;
    }

    else if (
        0.15 * this->design_speed_ms <= tidal_resource_ms and
        tidal_resource_ms <= this->design_speed_ms
    ) {
        production = (1 / pow(this->design_speed_ms, 3)) * pow(tidal_resource_ms, 3);
    }

    else {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __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

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

Returns

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

```
{
    double production = 0;

    double turbine_speed =
        (tidal_resource_ms - this->design_speed_ms) / this->design_speed_ms;
```

```
    if (turbine_speed < -0.71 or turbine_speed > 0.65) {
        production = 0;
    }

    else if (turbine_speed >= -0.71 and turbine_speed <= 0) {
        production = 1.69215 * exp(1.25909 * turbine_speed) - 0.69215;
    }

    else {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __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

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

Returns

The interpolated production [kW] of the tidal tubrine.

```
{
    // *** WORK IN PROGRESS *** //

    return 0;
}   /* __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].

```
{
    double capital_cost_per_kW = 2000 * pow(this->capacity_kW, -0.15) + 4000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __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].

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

4.28.3.7 __writeSummary()

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

Helper method to write summary results for Tidal.

Parameters

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs << "# ";
    ofs << std::to_string(int(ceil(this->capacity_kW)));
    ofs << " kW TIDAL Summary Results\n";
    ofs << "\n--------\n\n";

    //  2.1. Production attributes
    ofs << "## Production Attributes\n";
    ofs << "\n";

    ofs << "Capacity: " << this->capacity_kW << " kW  \n";
    ofs << "\n";

    ofs << "Production Override: (N = 0 / Y = 1): "
        << this->normalized_production_series_given << "  \n";
    if (this->normalized_production_series_given) {
        ofs << "Path to Normalized Production Time Series: "
            << this->path_2_normalized_production_time_series << "  \n";
    }
    ofs << "\n";

```

```
    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Tidal attributes
    ofs « "## Tidal Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";

    ofs « "\n--------\n\n";

    //  2.4. Tidal Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __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

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Tidal Resource [m/s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";

        if (not this->normalized_production_series_given) {
            ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        }

        else {
            ofs « "OVERRIDE" « ",";
        }

        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    return;
}   /* __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

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

Returns

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

Reimplemented from Renewable.

```
{
```

```
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* 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

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

Returns

The production [kW] of the tidal turbine.

Reimplemented from Renewable.

```
{
    //  given production time series override
    if (this->normalized_production_series_given) {
        double production_kW = Production :: getProductionkW(timestep);

        return production_kW;
    }

    // check if no resource
    if (tidal_resource_ms <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
            production_kW = this->__computeCubicProductionkW(
                timestep,
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }


        case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
            production_kW = this->__computeExponentialProductionkW(
                timestep,
```

```
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }

        case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                tidal_resource_ms
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Tidal::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

4.28.3.11 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.28.4 Member Data Documentation

4.28.4.1 design_speed_ms

`double Tidal::design_speed_ms`

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

4.28.4.2 power_model

`TidalPowerProductionModel Tidal::power_model`

The tidal power production model to be applied.

4.28.4.3 power_model_string

`std::string Tidal::power_model_string`

A string describing the active power production model.

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

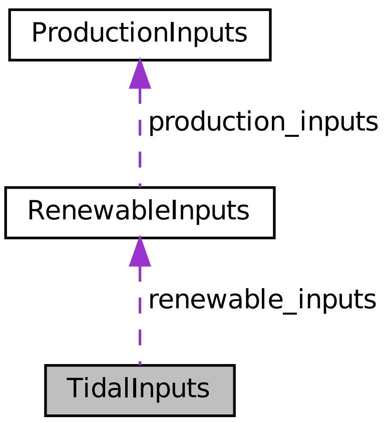
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.29 TidalInputs Struct Reference

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

`#include <Tidal.h>`

Collaboration diagram for TidalInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

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

  The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double design_speed_ms = 3

  The tidal stream speed [m/s] at which the tidal turbine achieves its rated capacity.
- TidalPowerProductionModel power_model = TidalPowerProductionModel :: TIDAL_POWER_CUBIC

  The tidal power production model to be applied.

4.29.1 Detailed Description

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

4.29.2 Member Data Documentation

4.29.2.1 capital_cost

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

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

4.29.2.2 design_speed_ms

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

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

4.29.2.3 operation_maintenance_cost_kWh

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

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

4.29.2.4 power_model

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

The tidal power production model to be applied.

4.29.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.29.2.6 resource_key

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

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

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

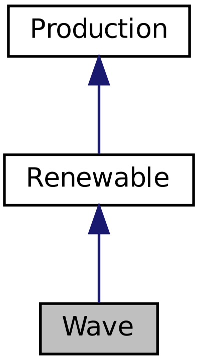
- header/Production/Renewable/Tidal.h

4.30 Wave Class Reference

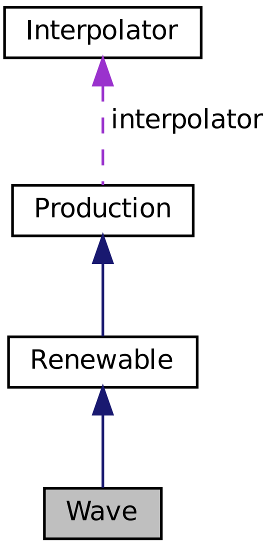
A derived class of the Renewable branch of Production which models wave production.

`#include <Wave.h>`

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

- Wave (void)

  Constructor (dummy) for the Wave class.
- Wave (int, double, WaveInputs, std::vector< double > *)

  Constructor (intended) for the Wave class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double, double)

  Method which takes in the wave resource at a particular point in time, and then returns the wave turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ∼Wave (void)

  Destructor for the Wave class.

Public Attributes

- double design_significant_wave_height_m

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

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

  The wave power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (WaveInputs)

  Helper method to check inputs to the Wave constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic wave energy converter capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic wave energy converter operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeGaussianProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production under a Gaussian production model.
- double __computeParaboloidProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production under a paraboloid production model.
- double __computeLookupProductionkW (int, double, double, double)

  Helper method to compute wave energy converter production by way of looking up using given performance matrix.
- void __writeSummary (std::string)

  Helper method to write summary results for Wave.
- void __writeTimeSeries (std::string, std::vector< double > *, std::map< int, std::vector< double >> *, std::map< int, std::vector< std::vector< double >>> *, int=-1)

  Helper method to write time series results for Wave.

4.30.1 Detailed Description

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

4.30.2 Constructor & Destructor Documentation

4.30.2.1 Wave() [1/2]

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

Constructor (dummy) for the Wave class.

```
{
    return;
}   /* Wave() */
```

4.30.2.2 Wave() [2/2]

```
Wave::Wave (
            int n_points,
            double n_years,
            WaveInputs wave_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Wave class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
wave_inputs	A structure of Wave constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Renewable(
    n_points,
    n_years,
    wave_inputs.renewable_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(wave_inputs);

    //  2. set attributes
    this->type = RenewableType :: WAVE;
    this->type_str = "WAVE";

    this->resource_key = wave_inputs.resource_key;

    this->design_significant_wave_height_m =
        wave_inputs.design_significant_wave_height_m;
    this->design_energy_period_s = wave_inputs.design_energy_period_s;

    this->power_model = wave_inputs.power_model;
```

```

    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            this->power_model_string = "GAUSSIAN";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
            this->power_model_string = "PARABOLOID";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            this->interpolator.addData2D(
                0,
                wave_inputs.path_2_normalized_performance_matrix
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wave():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    if (wave_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = wave_inputs.capital_cost;
    }

    if (wave_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            wave_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Wave object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

4.30.2.3 ∼Wave()

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

Destructor for the Wave class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Wave object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

wave_inputs	A structure of Wave constructor inputs.

```
{
    //  1. check design_significant_wave_height_m
    if (wave_inputs.design_significant_wave_height_m <= 0) {
        std::string error_str = "ERROR:  Wave():  ";
        error_str += "WaveInputs::design_significant_wave_height_m must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  2. check design_energy_period_s
    if (wave_inputs.design_energy_period_s <= 0) {
        std::string error_str = "ERROR:  Wave():  ";
        error_str += "WaveInputs::design_energy_period_s must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    //  3. if WAVE_POWER_LOOKUP, check that path is given
    if (
        wave_inputs.power_model == WavePowerProductionModel :: WAVE_POWER_LOOKUP and
        wave_inputs.path_2_normalized_performance_matrix.empty()
    ) {
        std::string error_str = "ERROR:  Wave()  power model was set to ";
        error_str += "WavePowerProductionModel::WAVE_POWER_LOOKUP, but no path to a ";
        error_str += "normalized performance matrix was given";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __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

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height←_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	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.

```
{
    double H_s_nondim =
        (significant_wave_height_m - this->design_significant_wave_height_m) /
        this->design_significant_wave_height_m;

    double T_e_nondim =
        (energy_period_s - this->design_energy_period_s) /
        this->design_energy_period_s;

    double production = exp(
        -2.25119 * pow(T_e_nondim, 2) +
        3.44570 * T_e_nondim * H_s_nondim -
        4.01508 * pow(H_s_nondim, 2)
    );

    return production * this->capacity_kW;
}   /* __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

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height←_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	The energy period [s] in the vicinity of the wave energy converter

Returns

The interpolated production [kW] of the wave energy converter.

```
{
    double prod = this->interpolator.interp2D(
        0,
        significant_wave_height_m,
        energy_period_s
    );

    return prod * this->capacity_kW;
}   /* __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

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
significant_wave_height↩_m	The significant wave height [m] in the vicinity of the wave energy converter.
energy_period_s	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.

```
{
    // first, check for idealized wave breaking (deep water)
    if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
        return 0;
    }

    // otherwise, apply generic quadratic performance model
    // (with outputs bounded to [0, 1])
    double production =
        0.289 * significant_wave_height_m -
        0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
        0.0169 * energy_period_s;

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

    else if (production > 1) {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __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].

```
{
    double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __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/k↩ Wh].

```
{
    double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

4.30.3.7 __writeSummary()

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

Helper method to write summary results for Wave.

Parameters

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW WAVE Summary Results\n";
    ofs « "\n--------\n\n";

    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « " kW  \n";
    ofs « "\n";

    ofs « "Production Override: (N = 0 / Y = 1): "
        « this->normalized_production_series_given « "  \n";
    if (this->normalized_production_series_given) {
        ofs « "Path to Normalized Production Time Series: "
            « this->path_2_normalized_production_time_series « "  \n";
    }
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (2D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Wave attributes
    ofs « "## Wave Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            ofs « "Design Significant Wave Height: "
                « this->design_significant_wave_height_m « " m  \n";

            ofs « "Design Energy Period: " « this->design_energy_period_s « " s  \n";

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            ofs « "Normalized Performance Matrix: "
                « this->interpolator.path_map_2D[0] « "  \n";

            break;
        }

        default: {
            // write nothing!

            break;
        }
    }
```

```
    ofs « "\n--------\n\n";

    //  2.4. Wave Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __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

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Significant Wave Height [m],";
    ofs « "Energy Period [s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";

    for (int i = 0; i < max_lines; i++) {
```

```
        ofs « time_vec_hrs_ptr->at(i) « ",";

        if (not this->normalized_production_series_given) {
            ofs « resource_map_2D_ptr->at(this->resource_key)[i][0] « ",";
            ofs « resource_map_2D_ptr->at(this->resource_key)[i][1] « ",";
        }

        else {
            ofs « "OVERRIDE" « ",";
            ofs « "OVERRIDE" « ",";
        }

        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    return;
}   /* __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

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

Returns

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* 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

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

Returns

The production [kW] of the wave turbine.

Reimplemented from Renewable.

```
{
    //  given production time series override
    if (this->normalized_production_series_given) {
        double production_kW = Production :: getProductionkW(timestep);

        return production_kW;
    }

    // check if no resource
    if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
            production_kW = this->__computeParaboloidProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
            production_kW = this->__computeGaussianProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

            break;
        }

        case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                significant_wave_height_m,
                energy_period_s
            );

```

```
            break;
        }

        default: {
            std::string error_str = "ERROR:  Wave::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

4.30.3.11 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.30.4 Member Data Documentation

4.30.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

4.30.4.2 design_significant_wave_height_m

`double Wave::design_significant_wave_height_m`

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

4.30.4.3 power_model

`WavePowerProductionModel Wave::power_model`

The wave power production model to be applied.

4.30.4.4 power_model_string

`std::string Wave::power_model_string`

A string describing the active power production model.

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

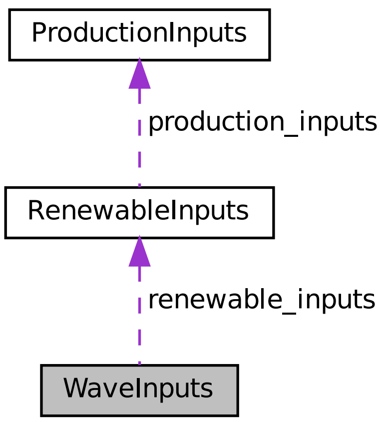
- header/Production/Renewable/Wave.h
- source/Production/Renewable/Wave.cpp

4.31 WaveInputs Struct Reference

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

`#include <Wave.h>`

Collaboration diagram for WaveInputs:



Public Attributes

- RenewableInputs renewable_inputs

    An encapsulated RenewableInputs instance.
- int resource_key = 0

    A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

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

    The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double design_significant_wave_height_m = 3

    The significant wave height [m] at which the wave energy converter achieves its rated capacity.
- double design_energy_period_s = 10

    The energy period [s] at which the wave energy converter achieves its rated capacity.
- WavePowerProductionModel power_model = WavePowerProductionModel :: WAVE_POWER_PARABOLOID

    The wave power production model to be applied.
- std::string path_2_normalized_performance_matrix = ""

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

4.31.1 Detailed Description

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

4.31.2 Member Data Documentation

4.31.2.1 capital_cost

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

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

4.31.2.2 design_energy_period_s

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

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

4.31.2.3 design_significant_wave_height_m

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

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

4.31.2.4 operation_maintenance_cost_kWh

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

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

4.31.2.5 path_2_normalized_performance_matrix

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

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

4.31.2.6 power_model

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

The wave power production model to be applied.

4.31.2.7 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.31.2.8 resource_key

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

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

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

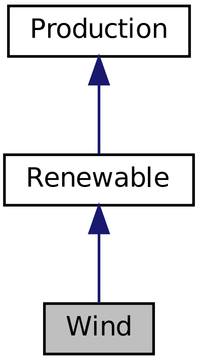
- header/Production/Renewable/Wave.h

4.32 Wind Class Reference

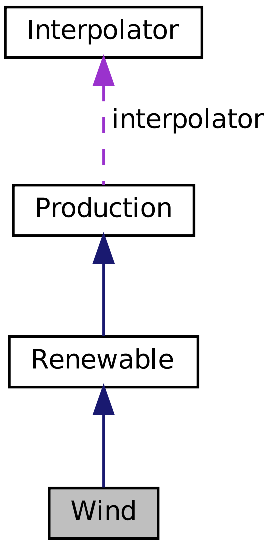
A derived class of the Renewable branch of Production which models wind production.

`#include <Wind.h>`

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- Wind (void)

  Constructor (dummy) for the Wind class.
- Wind (int, double, WindInputs, std::vector< double > ∗)

  Constructor (intended) for the Wind class.
- void handleReplacement (int)

  Method to handle asset replacement and capital cost incursion, if applicable.
- double computeProductionkW (int, double, double)

  Method which takes in the wind resource at a particular point in time, and then returns the wind turbine production at that point in time.
- double commit (int, double, double, double)

  Method which takes in production and load for the current timestep, computes and records dispatch and curtailment, and then returns remaining load.
- ∼Wind (void)

  Destructor for the Wind class.

Public Attributes

- double design_speed_ms

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

  The wind power production model to be applied.
- std::string power_model_string

  A string describing the active power production model.

Private Member Functions

- void __checkInputs (WindInputs)

  Helper method to check inputs to the Wind constructor.
- double __getGenericCapitalCost (void)

  Helper method to generate a generic wind turbine capital cost.
- double __getGenericOpMaintCost (void)

  Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double __computeCubicProductionkW (int, double, double)

  Helper method to compute wind turbine production under a cubic production model.
- double __computeExponentialProductionkW (int, double, double)

  Helper method to compute wind turbine production under an exponential production model.
- double __computeLookupProductionkW (int, double, double)

  Helper method to compute wind turbine production by way of looking up using given power curve data.
- void __writeSummary (std::string)

  Helper method to write summary results for Wind.
- void __writeTimeSeries (std::string, std::vector< double > ∗, std::map< int, std::vector< double >> ∗, std↩ ::map< int, std::vector< std::vector< double >>> ∗, int=-1)

  Helper method to write time series results for Wind.

4.32.1 Detailed Description

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

4.32.2 Constructor & Destructor Documentation

4.32.2.1 Wind() [1/2]

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

Constructor (dummy) for the Wind class.

```
{
    return;
}   /* Wind() */
```

4.32.2.2 Wind() [2/2]

```
Wind::Wind (
            int n_points,
            double n_years,
            WindInputs wind_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

Constructor (intended) for the Wind class.

Parameters

n_points	The number of points in the modelling time series.
n_years	The number of years being modelled.
wind_inputs	A structure of Wind constructor inputs.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
  :
Renewable(
    n_points,
    n_years,
    wind_inputs.renewable_inputs,
    time_vec_hrs_ptr
)
{
    //  1. check inputs
    this->__checkInputs(wind_inputs);

    //  2. set attributes
    this->type = RenewableType :: WIND;
    this->type_str = "WIND";

    this->resource_key = wind_inputs.resource_key;

    this->design_speed_ms = wind_inputs.design_speed_ms;

    this->power_model = wind_inputs.power_model;

    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
            this->power_model_string = "CUBIC";

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            this->power_model_string = "EXPONENTIAL";

```

```
            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            this->power_model_string = "LOOKUP";

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wind():  ";
            error_str += "power production model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";

            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    if (wind_inputs.capital_cost < 0) {
        this->capital_cost = this->__getGenericCapitalCost();
    }
    else {
        this->capital_cost = wind_inputs.capital_cost;
    }

    if (wind_inputs.operation_maintenance_cost_kWh < 0) {
        this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
    }
    else {
        this->operation_maintenance_cost_kWh =
            wind_inputs.operation_maintenance_cost_kWh;
    }

    if (not this->is_sunk) {
        this->capital_cost_vec[0] = this->capital_cost;
    }

    //  3. construction print
    if (this->print_flag) {
        std::cout « "Wind object constructed at " « this « std::endl;
    }

    return;
}   /* Renewable() */
```

4.32.2.3 ∼Wind()

```
Wind::∼Wind (
        void )
```

Destructor for the Wind class.

```
{
    //  1. destruction print
    if (this->print_flag) {
        std::cout « "Wind object at " « this « " destroyed" « std::endl;
    }

    return;
}   /* ~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

wind_inputs	A structure of Wind constructor inputs.

```
{
    //  1. check design_speed_ms
    if (wind_inputs.design_speed_ms <= 0) {
        std::string error_str = "ERROR:  Wind():  ";
        error_str += "WindInputs::design_speed_ms must be > 0";

        #ifdef _WIN32
            std::cout « error_str « std::endl;
        #endif

        throw std::invalid_argument(error_str);
    }

    return;
}   /* __checkInputs() */
```

4.32.3.2 __computeCubicProductionkW()

```
double Wind::__computeCubicProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms )  [private]
```

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

Ref: Milan et al. [2010]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

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

```
{
    double production = 0;

    double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
        this->design_speed_ms;

    if (turbine_speed < -0.7857 or turbine_speed > 0.7857) {
        production = 0;
    }

    else if (turbine_speed >= -0.7857 and turbine_speed <= 0) {
        production = (1 / pow(this->design_speed_ms, 3)) * pow(wind_resource_ms, 3);
```

```
    }

    else {
        production = 1;
    }

    return production * this->capacity_kW;
}   /* __computeCubicProductionkW() */
```

4.32.3.3 __computeExponentialProductionkW()

```
double Wind::__computeExponentialProductionkW (
              int timestep,
              double dt_hrs,
              double wind_resource_ms )  [private]
```

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

Ref: Truelove et al. [2019]

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

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

```
{
    double production = 0;

    double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
        this->design_speed_ms;

    if (turbine_speed < -0.76 or turbine_speed > 0.68) {
        production = 0;
    }

    else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
        production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
    }

    else {
        production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
    }

    return production * this->capacity_kW;
}   /* __computeExponentialProductionkW() */
```

4.32.3.4 __computeLookupProductionkW()

```
double Wind::__computeLookupProductionkW (
              int timestep,
              double dt_hrs,
              double wind_resource_ms )  [private]
```

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

Parameters

timestep	The current time step of the Model run.
dt_hrs	The interval of time [hrs] associated with the action.
wind_resource_ms	The available wind resource [m/s].

Returns

The interpolated production [kW] of the wind turbine.

```
{
    // *** WORK IN PROGRESS *** //

    return 0;
}   /* __computeLookupProductionkW() */
```

4.32.3.5 __getGenericCapitalCost()

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

Helper method to generate a generic wind turbine capital cost.

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

Returns

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

```
{
    double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;

    return capital_cost_per_kW * this->capacity_kW;
}   /* __getGenericCapitalCost() */
```

4.32.3.6 __getGenericOpMaintCost()

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

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

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

Returns

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

```
{
    double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;

    return operation_maintenance_cost_kWh;
}   /* __getGenericOpMaintCost() */
```

4.32.3.7 __writeSummary()

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

Helper method to write summary results for Wind.

Parameters

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "summary_results.md";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write summary results (markdown)
    ofs « "# ";
    ofs « std::to_string(int(ceil(this->capacity_kW)));
    ofs « " kW WIND Summary Results\n";
    ofs « "\n--------\n\n";


    //  2.1. Production attributes
    ofs « "## Production Attributes\n";
    ofs « "\n";

    ofs « "Capacity: " « this->capacity_kW « " kW  \n";
    ofs « "\n";

    ofs « "Production Override: (N = 0 / Y = 1): "
        « this->normalized_production_series_given « "  \n";
    if (this->normalized_production_series_given) {
        ofs « "Path to Normalized Production Time Series: "
            « this->path_2_normalized_production_time_series « "  \n";
    }
    ofs « "\n";

    ofs « "Sunk Cost (N = 0 / Y = 1): " « this->is_sunk « "  \n";
    ofs « "Capital Cost: " « this->capital_cost « "  \n";
    ofs « "Operation and Maintenance Cost: " « this->operation_maintenance_cost_kWh
        « " per kWh produced  \n";
    ofs « "Nominal Inflation Rate (annual): " « this->nominal_inflation_annual
        « "  \n";
    ofs « "Nominal Discount Rate (annual): " « this->nominal_discount_annual
        « "  \n";
    ofs « "Real Discount Rate (annual): " « this->real_discount_annual « "  \n";
    ofs « "\n";

    ofs « "Replacement Running Hours: " « this->replace_running_hrs « "  \n";
    ofs « "\n--------\n\n";

    //  2.2. Renewable attributes
    ofs « "## Renewable Attributes\n";
    ofs « "\n";

    ofs « "Resource Key (1D): " « this->resource_key « "  \n";

    ofs « "\n--------\n\n";

    //  2.3. Wind attributes
    ofs « "## Wind Attributes\n";
    ofs « "\n";

    ofs « "Power Production Model: " « this->power_model_string « "  \n";
    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
            ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            ofs « "Design Speed: " « this->design_speed_ms « " m/s  \n";

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            //...

            break;
        }

        default: {
            // write nothing!

```

```
            break;
        }
    }

    ofs « "\n--------\n\n";

    //  2.4. Wind Results
    ofs « "## Results\n";
    ofs « "\n";

    ofs « "Net Present Cost: " « this->net_present_cost « "  \n";
    ofs « "\n";

    ofs « "Total Dispatch: " « this->total_dispatch_kWh
        « " kWh  \n";

    ofs « "Levellized Cost of Energy: " « this->levellized_cost_of_energy_kWh
        « " per kWh dispatched  \n";
    ofs « "\n";

    ofs « "Running Hours: " « this->running_hours « "  \n";
    ofs « "Replacements: " « this->n_replacements « "  \n";

    ofs « "\n--------\n\n";

    ofs.close();

    return;
}   /* __writeSummary() */
```

4.32.3.8 __writeTimeSeries()

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

Helper method to write time series results for Wind.

Parameters

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

Reimplemented from Renewable.

```
{
    //  1. create filestream
    write_path += "time_series_results.csv";
    std::ofstream ofs;
    ofs.open(write_path, std::ofstream::out);

    //  2. write time series results (comma separated value)
    ofs « "Time (since start of data) [hrs],";
    ofs « "Wind Resource [m/s],";
    ofs « "Production [kW],";
    ofs « "Dispatch [kW],";
    ofs « "Storage [kW],";
    ofs « "Curtailment [kW],";
    ofs « "Capital Cost (actual),";
    ofs « "Operation and Maintenance Cost (actual),";
    ofs « "\n";
```

```
    for (int i = 0; i < max_lines; i++) {
        ofs « time_vec_hrs_ptr->at(i) « ",";

        if (not this->normalized_production_series_given) {
            ofs « resource_map_1D_ptr->at(this->resource_key)[i] « ",";
        }

        else {
            ofs « "OVERRIDE" « ",";
        }

        ofs « this->production_vec_kW[i] « ",";
        ofs « this->dispatch_vec_kW[i] « ",";
        ofs « this->storage_vec_kW[i] « ",";
        ofs « this->curtailment_vec_kW[i] « ",";
        ofs « this->capital_cost_vec[i] « ",";
        ofs « this->operation_maintenance_cost_vec[i] « ",";
        ofs « "\n";
    }

    return;
}   /* __writeTimeSeries() */
```

4.32.3.9 commit()

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

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

Parameters

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

Returns

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

Reimplemented from Renewable.

```
{
    //  1. invoke base class method
    load_kW = Renewable :: commit(
        timestep,
        dt_hrs,
        production_kW,
        load_kW
    );


    //...

    return load_kW;
}   /* commit() */
```

4.32.3.10 computeProductionkW()

```
double Wind::computeProductionkW (
            int timestep,
            double dt_hrs,
            double wind_resource_ms )  [virtual]
```

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

Parameters

timestep	The timestep (i.e., time series index) for the request.
dt_hrs	The interval of time [hrs] associated with the timestep.
wind_resource_ms	Wind resource (i.e. wind speed) [m/s].

Returns

The production [kW] of the wind turbine.

Reimplemented from Renewable.

```
{
    //  given production time series override
    if (this->normalized_production_series_given) {
        double production_kW = Production :: getProductionkW(timestep);

        return production_kW;
    }

    // check if no resource
    if (wind_resource_ms <= 0) {
        return 0;
    }

    // compute production
    double production_kW = 0;

    switch (this->power_model) {
        case (WindPowerProductionModel :: WIND_POWER_CUBIC): {
            production_kW = this->__computeCubicProductionkW(
                timestep,
                dt_hrs,
                wind_resource_ms
            );

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
            production_kW = this->__computeExponentialProductionkW(
                timestep,
                dt_hrs,
                wind_resource_ms
            );

            break;
        }

        case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
            production_kW = this->__computeLookupProductionkW(
                timestep,
                dt_hrs,
                wind_resource_ms
            );

            break;
        }

        default: {
            std::string error_str = "ERROR:  Wind::computeProductionkW():  ";
            error_str += "power model ";
            error_str += std::to_string(this->power_model);
            error_str += " not recognized";
```

```
            #ifdef _WIN32
                std::cout « error_str « std::endl;
            #endif

            throw std::runtime_error(error_str);

            break;
        }
    }

    return production_kW;
}   /* computeProductionkW() */
```

4.32.3.11 handleReplacement()

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

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

Parameters

timestep	The current time step of the Model run.

Reimplemented from Renewable.

```
{
    //  1. reset attributes
    //...

    //  2. invoke base class method
    Renewable :: handleReplacement(timestep);

    return;
}   /* __handleReplacement() */
```

4.32.4 Member Data Documentation

4.32.4.1 design_speed_ms

```
double Wind::design_speed_ms
```

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

4.32.4.2 power_model

```
WindPowerProductionModel Wind::power_model
```

The wind power production model to be applied.

4.32.4.3 power_model_string

```
std::string Wind::power_model_string
```

A string describing the active power production model.

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

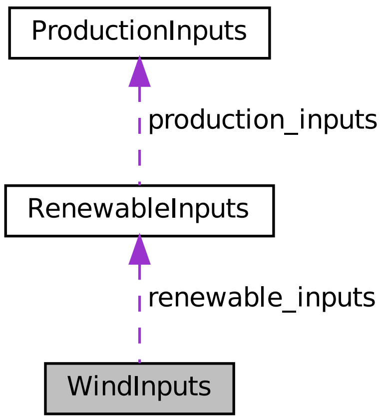
- header/Production/Renewable/Wind.h
- source/Production/Renewable/Wind.cpp

4.33 WindInputs Struct Reference

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

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

Collaboration diagram for WindInputs:



Public Attributes

- RenewableInputs renewable_inputs

  An encapsulated RenewableInputs instance.
- int resource_key = 0

  A key used to index into the Resources object, to associate this asset with the appropriate resource time series.
- double capital_cost = -1

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

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

- double design_speed_ms = 14

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

- WindPowerProductionModel power_model = WindPowerProductionModel :: WIND_POWER_CUBIC

  The wind power production model to be applied.

4.33.1 Detailed Description

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

4.33.2 Member Data Documentation

4.33.2.1 capital_cost

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

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

4.33.2.2 design_speed_ms

```
double WindInputs::design_speed_ms = 14
```

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

4.33.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

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

4.33.2.4 power_model

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

The wind power production model to be applied.

4.33.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated RenewableInputs instance.

4.33.2.6 resource_key

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

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

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

- header/Production/Renewable/Wind.h

Chapter 5

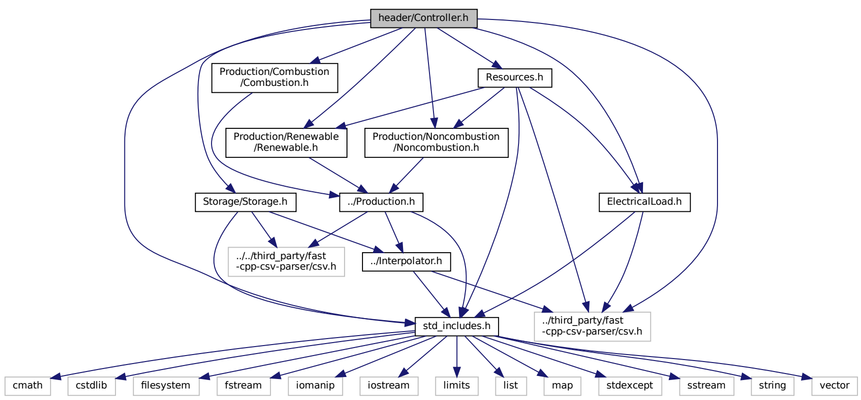
File Documentation

5.1 header/Controller.h File Reference

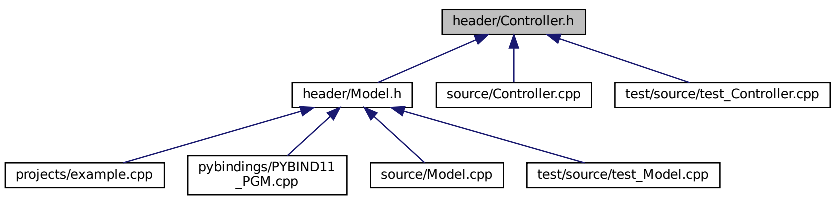
Header file for the Controller class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Combustion.h"
#include "Production/Noncombustion/Noncombustion.h"
#include "Production/Renewable/Renewable.h"
#include "Storage/Storage.h"
```

Include dependency graph for Controller.h:



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



Classes

- class Controller

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

Enumerations

- enum ControlMode { LOAD_FOLLOWING , CYCLE_CHARGING , N_CONTROL_MODES }

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

5.1.1 Detailed Description

Header file for the Controller class.

5.1.2 Enumeration Type Documentation

5.1.2.1 ControlMode

`enum ControlMode`

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

Enumerator

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

```
                 {
    LOAD_FOLLOWING,
    CYCLE_CHARGING,
    N_CONTROL_MODES
};
```

5.2 header/doxygen_cite.h File Reference

Header file which simply cites the doxygen tool.

5.2.1 Detailed Description

Header file which simply cites the doxygen tool.

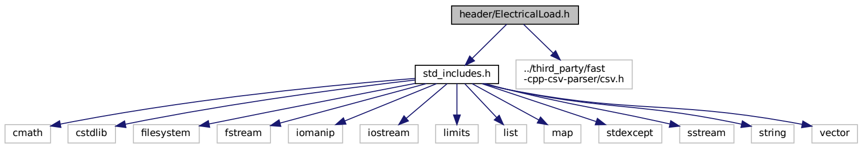
Ref: van Heesch. [2023]

5.3 header/ElectricalLoad.h File Reference

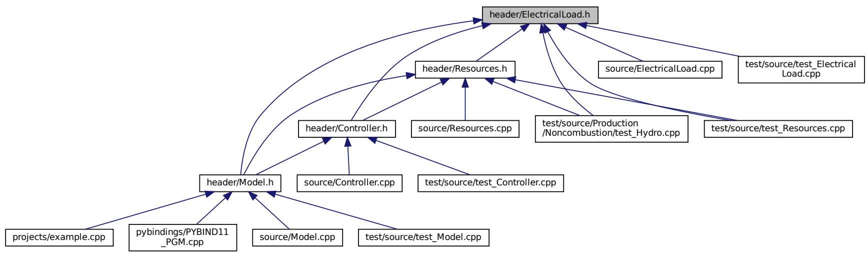
Header file for the ElectricalLoad class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
```

Include dependency graph for ElectricalLoad.h:



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



Classes

- class ElectricalLoad

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

5.3.1 Detailed Description

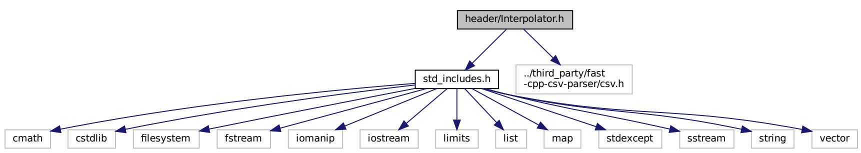
Header file for the ElectricalLoad class.

5.4 header/Interpolator.h File Reference

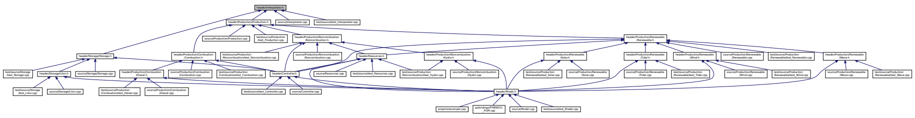
Header file for the Interpolator class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
```

Include dependency graph for Interpolator.h:



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



Classes

- struct InterpolatorStruct1D

    A struct which holds two parallel vectors for use in 1D interpolation.
- struct InterpolatorStruct2D

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

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

5.4.1 Detailed Description

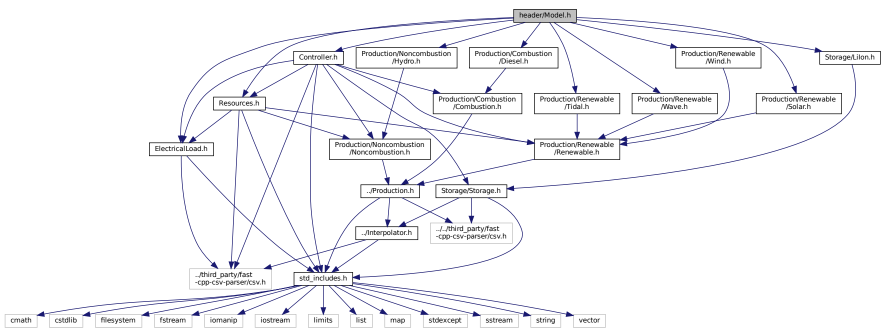
Header file for the Interpolator class.

5.5 header/Model.h File Reference

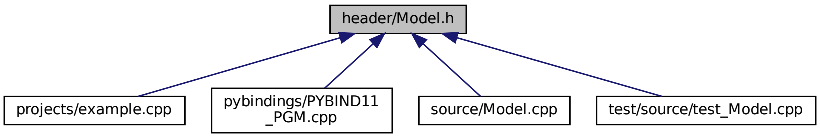
Header file for the Model class.

```
#include "Controller.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Diesel.h"
#include "Production/Noncombustion/Hydro.h"
#include "Production/Renewable/Solar.h"
#include "Production/Renewable/Tidal.h"
#include "Production/Renewable/Wave.h"
#include "Production/Renewable/Wind.h"
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:



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



Classes

- struct ModelInputs

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

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

5.5.1 Detailed Description

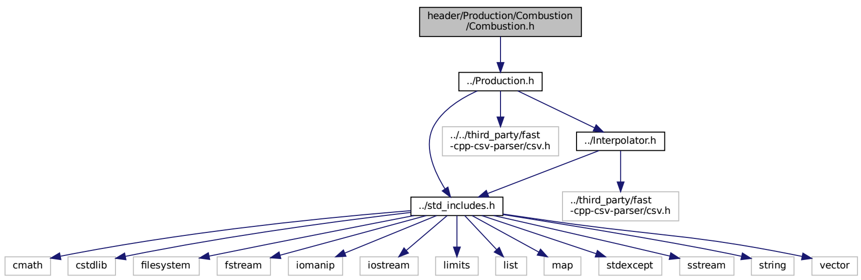
Header file for the Model class.

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

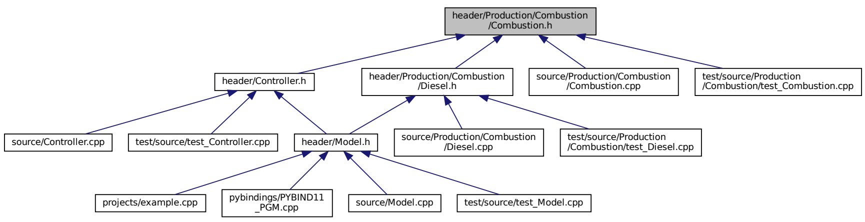
Header file for the Combustion class.

```
#include "../Production.h"
```

Include dependency graph for Combustion.h:



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



Classes

- struct CombustionInputs

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

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

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

Enumerations

- enum CombustionType { DIESEL , N_COMBUSTION_TYPES }

  An enumeration of the types of Combustion asset supported by PGMcpp.
- enum FuelMode { FUEL_MODE_LINEAR , FUEL_MODE_LOOKUP , N_FUEL_MODES }

  An enumeration of the fuel modes for the Combustion asset which are supported by PGMcpp.

5.6.1 Detailed Description

Header file for the Combustion class.

Header file for the Noncombustion class.

5.6.2 Enumeration Type Documentation

5.6.2.1 CombustionType

`enum CombustionType`

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

Enumerator

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

```
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.

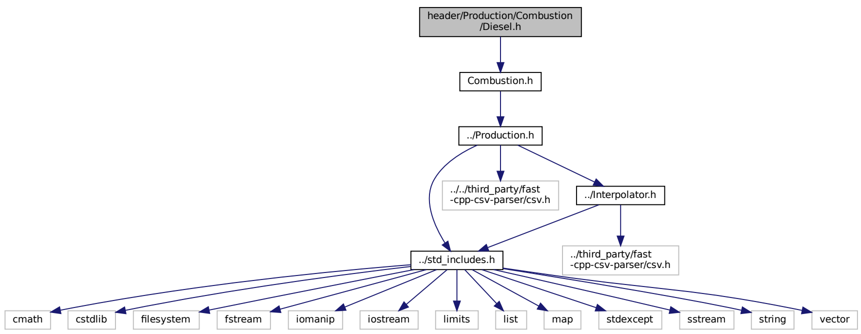
```
            {
    FUEL_MODE_LINEAR,
    FUEL_MODE_LOOKUP,
    N_FUEL_MODES
};
```

5.7 header/Production/Combustion/Diesel.h File Reference

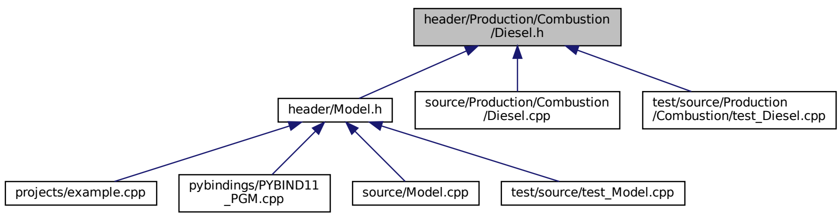
Header file for the Diesel class.

```
#include "Combustion.h"
```

Include dependency graph for Diesel.h:



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



Classes

- struct DieselInputs

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

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

5.7.1 Detailed Description

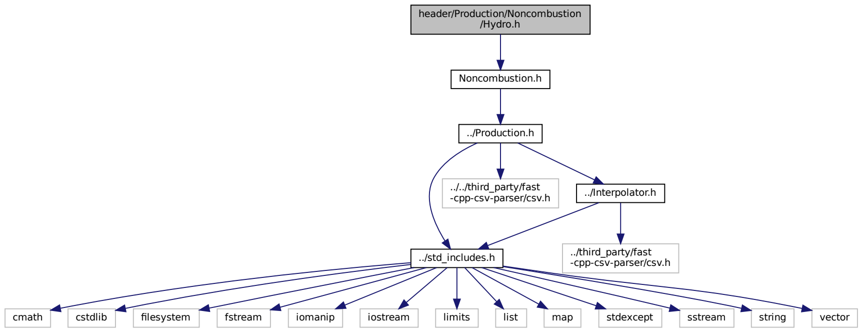
Header file for the Diesel class.

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

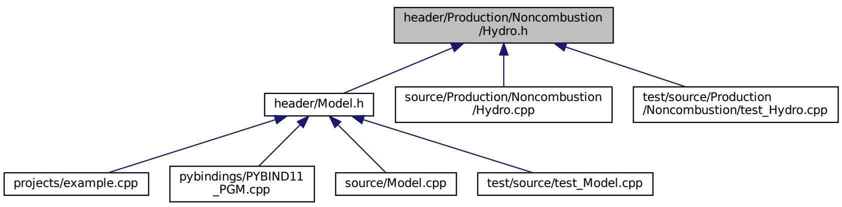
Header file for the Hydro class.

```
#include "Noncombustion.h"
```

Include dependency graph for Hydro.h:



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



Classes

- struct HydroInputs

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

- class Hydro

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

Enumerations

- enum HydroTurbineType { HYDRO_TURBINE_PELTON , HYDRO_TURBINE_FRANCIS , HYDRO_TURBINE_KAPLAN , N_HYDRO_TURBINES }

  An enumeration of the types of hydroelectric turbine supported by PGMcpp.
- enum HydroInterpKeys { GENERATOR_EFFICIENCY_INTERP_KEY , TURBINE_EFFICIENCY_INTERP_KEY , FLOW_TO_POWER_INTERP_KEY , N_HYDRO_INTERP_KEYS }

  An enumeration of the Interpolator keys used by the Hydro asset.

5.8.1 Detailed Description

Header file for the Hydro class.

5.8.2 Enumeration Type Documentation

5.8.2.1 HydroInterpKeys

```
enum HydroInterpKeys
```

An enumeration of the Interpolator keys used by the Hydro asset.

Enumerator

GENERATOR_EFFICIENCY_INTERP_KEY	The key for generator efficiency interpolation.
TURBINE_EFFICIENCY_INTERP_KEY	The key for turbine efficiency interpolation.
FLOW_TO_POWER_INTERP_KEY	The key for flow to power interpolation.
N_HYDRO_INTERP_KEYS	A simple hack to get the number of elements in HydroInterpKeys.

```
                    {
    GENERATOR_EFFICIENCY_INTERP_KEY,
    TURBINE_EFFICIENCY_INTERP_KEY,
    FLOW_TO_POWER_INTERP_KEY,
    N_HYDRO_INTERP_KEYS
};
```

5.8.2.2 HydroTurbineType

```
enum HydroTurbineType
```

An enumeration of the types of hydroelectric turbine supported by PGMcpp.

Enumerator

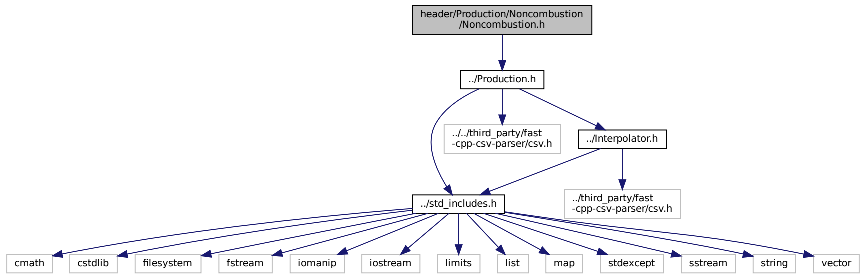
HYDRO_TURBINE_PELTON	A Pelton turbine (impluse)
HYDRO_TURBINE_FRANCIS	A Francis turbine (reaction)
HYDRO_TURBINE_KAPLAN	A Kaplan turbine (reaction)
N_HYDRO_TURBINES	A simple hack to get the number of elements in HydroTurbineType.

```
                        {
       HYDRO_TURBINE_PELTON,
       HYDRO_TURBINE_FRANCIS,
       HYDRO_TURBINE_KAPLAN,
       N_HYDRO_TURBINES
   };
```

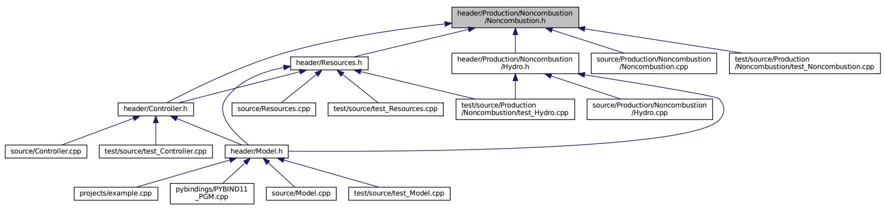
5.9 header/Production/Noncombustion/Noncombustion.h File Reference

`#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.

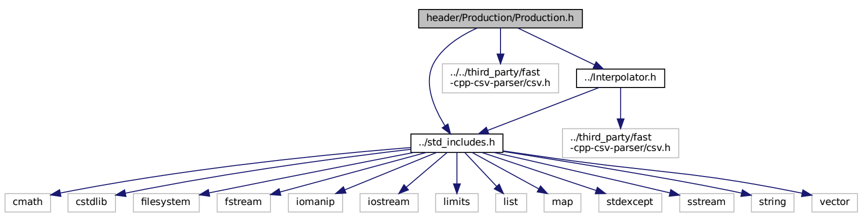
```
{
    HYDRO,
    N_NONCOMBUSTION_TYPES
};
```

5.10 header/Production/Production.h File Reference

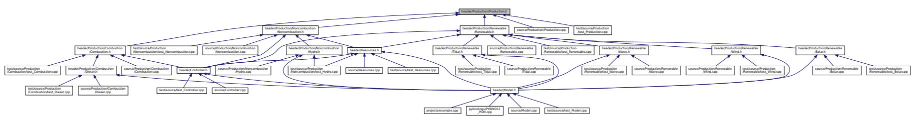
Header file for the Production class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
#include "../Interpolator.h"
```

Include dependency graph for Production.h:



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



Classes

- struct ProductionInputs

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

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

5.10.1 Detailed Description

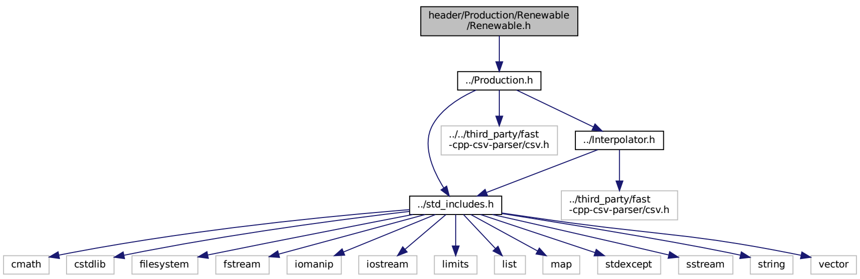
Header file for the Production class.

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

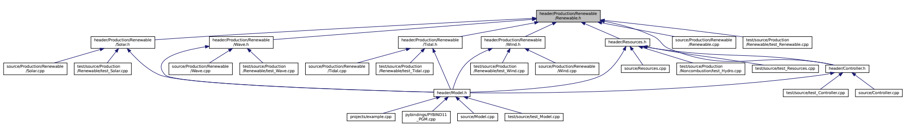
Header file for the Renewable class.

`#include "../Production.h"`

Include dependency graph for Renewable.h:



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



Classes

- struct RenewableInputs

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

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

Enumerations

- enum RenewableType {
  SOLAR , TIDAL , WAVE , WIND ,
  N_RENEWABLE_TYPES }

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

5.11.1 Detailed Description

Header file for the Renewable class.

5.11.2 Enumeration Type Documentation

5.11.2.1 RenewableType

```
enum RenewableType
```

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

Enumerator

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.

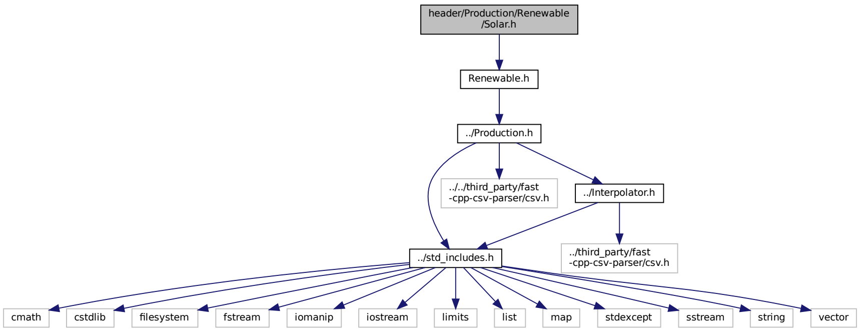
```
                      {
      SOLAR,
      TIDAL,
      WAVE,
      WIND,
      N_RENEWABLE_TYPES
 };
```

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

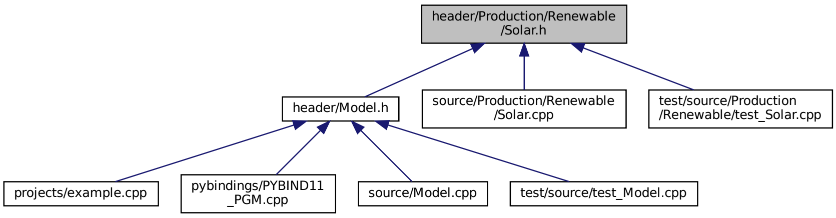
Header file for the Solar class.

`#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

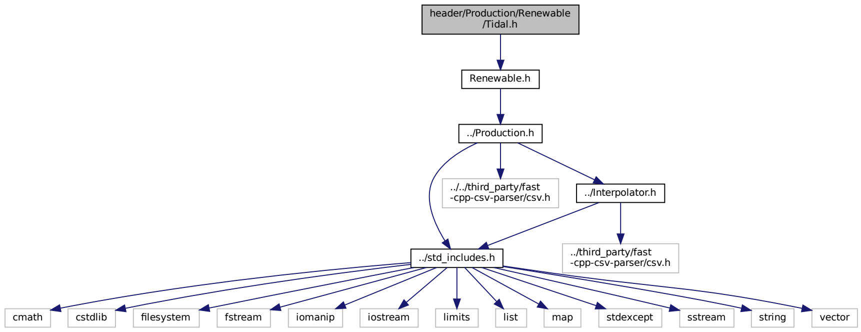
Header file for the Solar class.

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

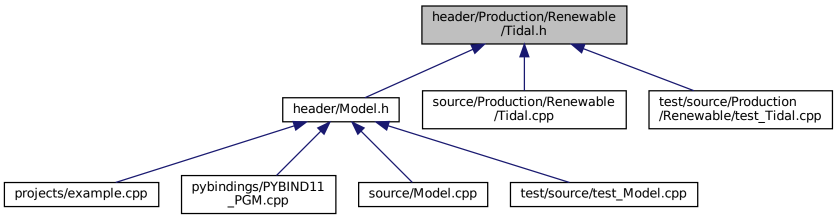
Header file for the Tidal class.

`#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.

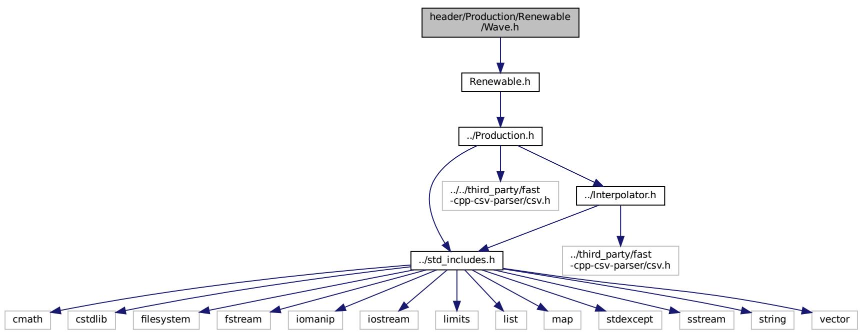
```
                    {
    TIDAL_POWER_CUBIC,
    TIDAL_POWER_EXPONENTIAL,
    TIDAL_POWER_LOOKUP,
    N_TIDAL_POWER_PRODUCTION_MODELS
};
```

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

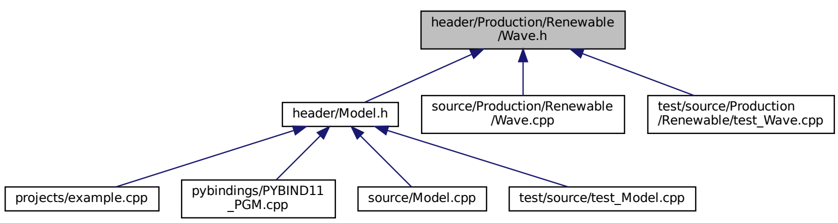
Header file for the Wave class.

`#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.

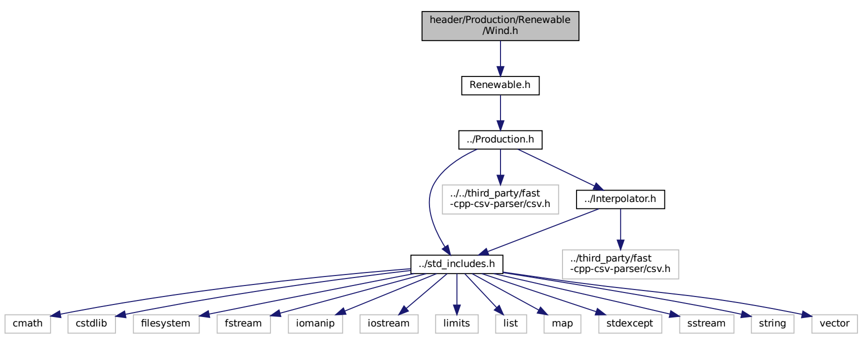
```
                        {
    WAVE_POWER_GAUSSIAN,
    WAVE_POWER_PARABOLOID,
    WAVE_POWER_LOOKUP,
    N_WAVE_POWER_PRODUCTION_MODELS
};
```

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

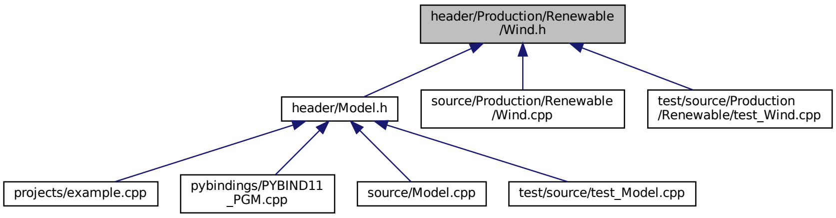
Header file for the Wind class.

`#include "Renewable.h"`

Include dependency graph for Wind.h:



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



Classes

- struct WindInputs

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

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

Enumerations

- enum WindPowerProductionModel { WIND_POWER_CUBIC , WIND_POWER_EXPONENTIAL , WIND_POWER_LOOKUP , N_WIND_POWER_PRODUCTION_MODELS }

5.15.1 Detailed Description

Header file for the Wind class.

5.15.2 Enumeration Type Documentation

5.15.2.1 WindPowerProductionModel

```
enum WindPowerProductionModel
```

Enumerator

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

```
                            {
    WIND_POWER_CUBIC,
```

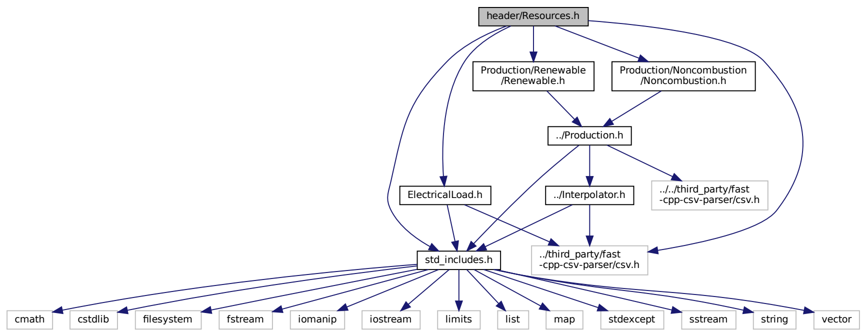
```
    WIND_POWER_EXPONENTIAL,
    WIND_POWER_LOOKUP,
    N_WIND_POWER_PRODUCTION_MODELS
};
```

5.16 header/Resources.h File Reference

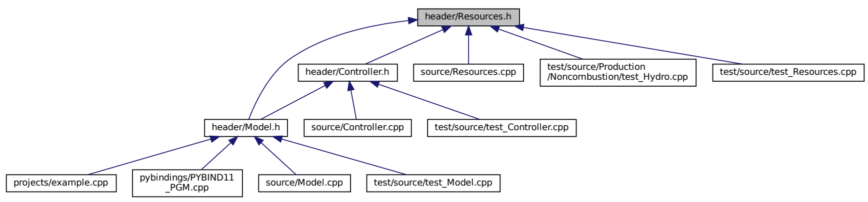
Header file for the Resources class.

```
#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

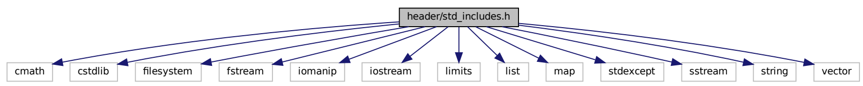
Header file for the Resources class.

5.17 header/std_includes.h File Reference

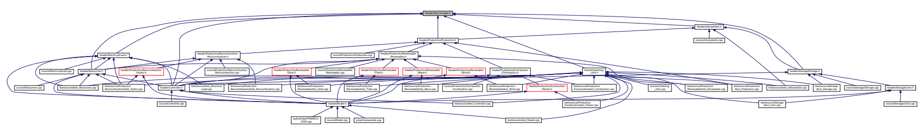
Header file which simply batches together some standard includes.

```
#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

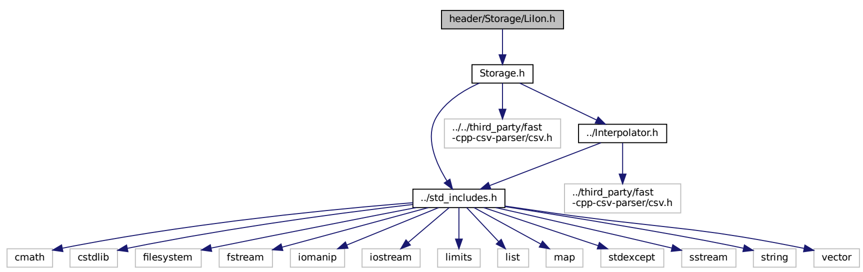
Header file which simply batches together some standard includes.

5.18 header/Storage/LiIon.h File Reference

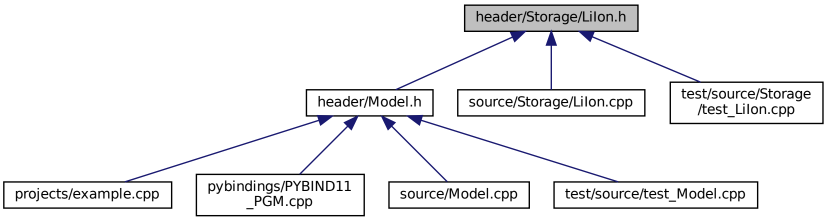
Header file for the LiIon class.

```
#include "Storage.h"
```

Include dependency graph for LiIon.h:



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



Classes

- struct LiIonInputs

  A structure which bundles the necessary inputs for the LiIon constructor. Provides default values for every necessary input. Note that this structure encapsulates StorageInputs.
- class LiIon

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

5.18.1 Detailed Description

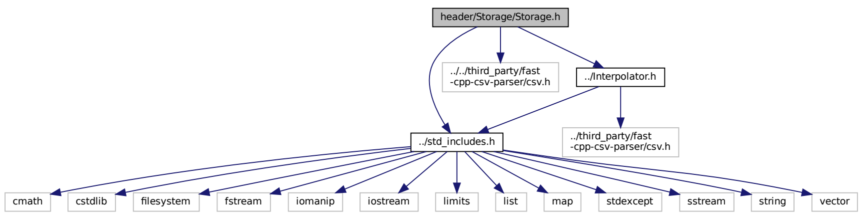
Header file for the LiIon class.

5.19 header/Storage/Storage.h File Reference

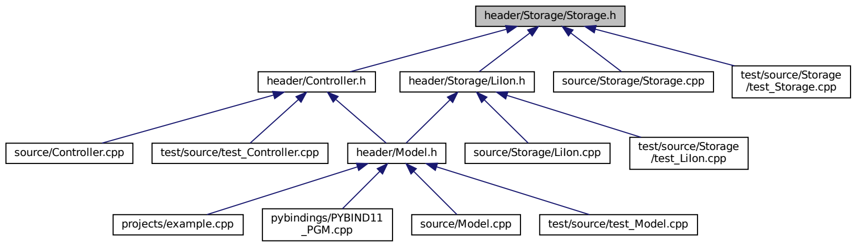
Header file for the Storage class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
#include "../Interpolator.h"
```

Include dependency graph for Storage.h:



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



Classes

- struct StorageInputs

    A structure which bundles the necessary inputs for the Storage constructor. Provides default values for every necessary input.
- class Storage

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

Enumerations

- enum StorageType { LIION , N_STORAGE_TYPES }

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

5.19.1 Detailed Description

Header file for the Storage class.

5.19.2 Enumeration Type Documentation

5.19.2.1 StorageType

```
enum StorageType
```

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

Enumerator

LIION	A system of lithium ion batteries.
N_STORAGE_TYPES	A simple hack to get the number of elements in StorageType.

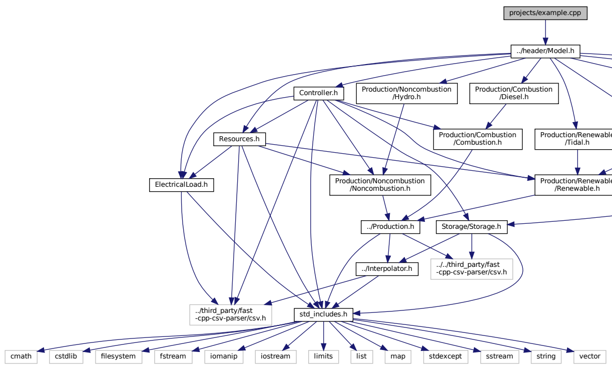
```
36               {
37     LIION,
```

```
    N_STORAGE_TYPES
};
```

5.20 projects/example.cpp File Reference

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

Include dependency graph for example.cpp:



Functions

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

5.20.1 Function Documentation

5.20.1.1 main()

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

```
{
    /*
     * 1. construct Model object
     *
     * This block constructs a Model object, which is the central container for the
     * entire microgrid model.
     *
     * The fist argument that must be provided to the Model constructor is a valid
     * path (either relative or absolute) to a time series of electrical load data.
     * For an example of the expected format, see
     *
     * data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv
     *
     * Note that the length of the given electrical load time series defines the
     * modelled project life (so if you want to model n years of microgrid operation,
     * then you must pass a path to n years worth of electrical load data). In addition,
     * the given electrical load time series defines which points in time are modelled.
     * As such, all subsequent time series data which is passed in must (1) be of the
     * same length as the electrical load time series, and (2) provide data for the
     * same set of points in time. Of course, the electrical load time series can be
     * of arbitrary length, and it need not be a uniform time series.
     *
     * The second argument that one can provide is the desired disptach control mode.
```

```
     *  If nothing is given here, then the model will default to simple load following
     *  control. However, one can stipulate which control mode to use by altering the
     *  control_mode attribute of the ModelInputs structure. In this case, the
     *  cycle charging control mode is being set.
     */

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

    ModelInputs model_inputs;

    model_inputs.path_2_electrical_load_time_series =
        path_2_electrical_load_time_series;

    model_inputs.control_mode = ControlMode :: CYCLE_CHARGING;

    Model model(model_inputs);



    /*
     *  2. add Diesel objects to Model
     *
     *  This block defines and adds a set of diesel generators to the Model object.
     *
     *  In this example, a single DieselInputs structure is used to define and add
     *  three diesel generators to the model.
     *
     *  The first diesel generator is defined as a 300 kW generator (which shows an
     *  example of how to access and alter an encapsulated attribute of DieselInputs).
     *  In addition, the diesel generator is taken to be a sunk cost (and so no capital
     *  cost is incurred in the first time step; the opposite is true for non-sunk
     *  assets).
     *
     *  The last two diesel generators are defined as 150 kW each. Likewise, they are
     *  also sunk assets (since the same DieselInputs structure is being re-used without
     *  overwriting the is_sunk attribute).
     *
     *  For more details on the various attributes of DieselInputs, refer to the
     *  PGMcpp manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     */

    DieselInputs diesel_inputs;

    //  2.1. add 1 x 300 kW diesel generator (since mean load is ~250 kW)
    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 300;
    diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;

    model.addDiesel(diesel_inputs);

    //  2.2. add 2 x 150 kW diesel generators (since max load is 500 kW)
    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;

    model.addDiesel(diesel_inputs);
    model.addDiesel(diesel_inputs);



    /*
     *  3. add renewable resources to Model
     *
     *  This block adds a set of renewable resource time series to the Model object.
     *
     *  The first resource added is a solar resource time series, which gives
     *  horizontal irradiance [kW/m2] at each point in time. Again, remember that all
     *  given time series must align with the electrical load time series (i.e., same
     *  length, same points). For an example of the expected format, see
     *
     *  data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv
     *
     *  Finally, note the declaration of a solar resource key. This variable will be
     *  re-used later to associate a solar PV array object with this particular solar
     *  resource. This method of key association between resource and asset allows for
     *  greater flexibility in modelling production assets that are exposed to different
     *  renewable resources (due to being geographically separated, etc.).
     *
     *  The second resource added is a tidal resource time series, which gives tidal
     *  stream speed [m/s] at each point in time. For an example of the expected format,
     *  see
     *
     *  data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv
     *
     *  Again, note the tidal resource key.
     *
     *  The third resource added is a wave resource time series, which gives significant
     *  wave height [m] and energy period [s] at each point in time. For an example of
```

```
     * the expected format, see
     *
     * data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv
     *
     * Again, note the wave resource key.
     *
     * The fourth resource added is a wind resource time series, which gives wind speed
     * [m/s] at each point in time. For an example of the expected format, see
     *
     * data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv
     *
     * Again, note the wind resource key.
     *
     * The fifth resource added is a hydro resource time series, which gives inflow
     * rate [m3/hr] at each point in time. For an example of the expected format, see
     *
     * data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv
     *
     * Again, note the hydro resource key.
     */

    // 3.1. add solar resource time series
    int solar_resource_key = 0;
    std::string path_2_solar_resource_data =
        "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

    model.addResource(
        RenewableType :: SOLAR,
        path_2_solar_resource_data,
        solar_resource_key
    );

    // 3.2. add tidal resource time series
    int tidal_resource_key = 1;
    std::string path_2_tidal_resource_data =
        "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

    model.addResource(
        RenewableType :: TIDAL,
        path_2_tidal_resource_data,
        tidal_resource_key
    );

    // 3.3. add wave resource time series
    int wave_resource_key = 2;
    std::string path_2_wave_resource_data =
        "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

    model.addResource(
        RenewableType :: WAVE,
        path_2_wave_resource_data,
        wave_resource_key
    );

    // 3.4. add wind resource time series
    int wind_resource_key = 3;
    std::string path_2_wind_resource_data =
        "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";

    model.addResource(
        RenewableType :: WIND,
        path_2_wind_resource_data,
        wind_resource_key
    );

    // 3.5. add hydro resource time series
    int hydro_resource_key = 4;
    std::string path_2_hydro_resource_data =
        "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";

    model.addResource(
        NoncombustionType :: HYDRO,
        path_2_hydro_resource_data,
        hydro_resource_key
    );



    /*
     * 4. add Hydro object to Model
     *
     * This block defines and adds a hydroelectric asset to the Model object.
     *
     * In this example, a 300 kW hydroelectric station with a 10,000 m3 reservoir
     * is defined. The initial reservoir state is set to 50% (so half full), and the
     * hydroelectric asset is taken to be a sunk asset (so no capital cost incurred
     * in the first time step). Note the association with the previously given hydro
```

```
     *  resource series by way of the hydro resource key.
     *
     *  For more details on the various attributes of HydroInputs, refer to the
     *  PGMcpp manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     */

    HydroInputs hydro_inputs;
    hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
    hydro_inputs.reservoir_capacity_m3 = 10000;
    hydro_inputs.init_reservoir_state = 0.5;
    hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
    hydro_inputs.resource_key = hydro_resource_key;

    model.addHydro(hydro_inputs);



    /*
     *  5. add Renewable objects to Model
     *
     *  This block defines and adds a set of renewable production assets to the Model
     *  object.
     *
     *  The first block defines and adds a solar PV array to the Model object. In this
     *  example, the installed solar capacity is set to 250 kW. Note the association
     *  with the previously given solar resource series by way of the solar resource
     *  key. Also, note that this asset is not taken as sunk (as the is_sunk attribute
     *  of the SolarInputs structure is unchanged and thus defaults to true). As such,
     *  this asset will incur a capital cost in the first time step.
     *
     *  For more details on the various attributes of SolarInputs, refer to the PGMcpp
     *  manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     *
     *  The second block defines and adds a tidal turbine to the Model object. In this
     *  example, the installed tidal capacity is set to 120 kW. In addition, the design
     *  speed of the asset (i.e., the speed at which the rated capacity is achieved) is
     *  set to 2.5 m/s. Note the association with the previously given tidal resource
     *  series by way of the tidal resource key.
     *
     *  For more details on the various attributes of TidalInputs, refer to the PGMcpp
     *  manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     *
     *  The third block defines and adds a wind turbine to the Model object. In this
     *  example, the installed wind capacity is set to 150 kW. In addition, the design
     *  speed of the asset is not given, and so will default to 8 m/s. Note the
     *  association with the previously given tidal resource series by way of the wind
     *  resource key.
     *
     *  For more details on the various attributes of WindInputs, refer to the PGMcpp
     *  manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     *
     *  The fourth block defines and adds a wave energy converter to the Model object.
     *  In this example, the installed wave capacity is set to 100 kW. Note the
     *  association with the previously given wave resource series by way of the wave
     *  resource key.
     *
     *  For more details on the various attributes of WaveInputs, refer to the PGMcpp
     *  manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     */

    //  5.1. add 1 x 250 kW solar PV array
    SolarInputs solar_inputs;

    solar_inputs.renewable_inputs.production_inputs.capacity_kW = 250;
    solar_inputs.resource_key = solar_resource_key;

    model.addSolar(solar_inputs);

    //  5.2. add 1 x 120 kW tidal turbine
    TidalInputs tidal_inputs;

    tidal_inputs.renewable_inputs.production_inputs.capacity_kW = 120;
    tidal_inputs.design_speed_ms = 2.5;
    tidal_inputs.resource_key = tidal_resource_key;

    model.addTidal(tidal_inputs);

    //  5.3. add 1 x 150 kW wind turbine
    WindInputs wind_inputs;

    wind_inputs.renewable_inputs.production_inputs.capacity_kW = 150;
    wind_inputs.resource_key = wind_resource_key;
```

```

    model.addWind(wind_inputs);

    //  5.4. add 1 x 100 kW wave energy converter
    WaveInputs wave_inputs;

    wave_inputs.renewable_inputs.production_inputs.capacity_kW = 100;
    wave_inputs.resource_key = wave_resource_key;

    model.addWave(wave_inputs);



    /*
     *  6. add LiIon object to Model
     *
     *  This block defines and adds a lithium ion battery energy storage system to the
     *  Model object.
     *
     *  In this example, a battery energy storage system with a 500 kW power capacity
     *  and a 1050 kWh energy capacity (which represents about four hours of mean load
     *  autonomy) is defined.
     *
     *  For more details on the various attributes of LiIonInputs, refer to the PGMcpp
     *  manual. For instance, note that no economic inputs are given; in this
     *  example, the default values apply.
     */

    //  6.1. add 1 x (500 kW, ) lithium ion battery energy storage system
    LiIonInputs liion_inputs;

    liion_inputs.storage_inputs.power_capacity_kW = 500;
    liion_inputs.storage_inputs.energy_capacity_kWh = 1050;

    model.addLiIon(liion_inputs);



    /*
     *  7. run and write results
     *
     *  This block runs the model and then writes results to the given output path
     *  (either relative or absolute). Note that the writeResults() will create the
     *  last directory on the given path, but not any in-between directories, so be
     *  sure those exist before calling out to this method.
     */

    model.run();

    model.writeResults("projects/example_cpp");

    return 0;
}   /* main() */
```

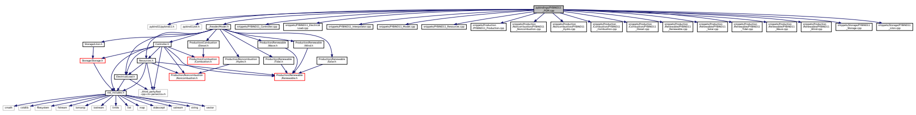
5.21 pybindings/PYBIND11_PGM.cpp File Reference

Bindings file for PGMcpp.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include "../header/Model.h"
#include "snippets/PYBIND11_Controller.cpp"
#include "snippets/PYBIND11_ElectricalLoad.cpp"
#include "snippets/PYBIND11_Interpolator.cpp"
#include "snippets/PYBIND11_Model.cpp"
#include "snippets/PYBIND11_Resources.cpp"
#include "snippets/Production/PYBIND11_Production.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
#include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"
#include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
#include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"
#include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
#include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
```

```
#include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
#include "snippets/Production/Renewable/PYBIND11_Wind.cpp"
#include "snippets/Storage/PYBIND11_Storage.cpp"
#include "snippets/Storage/PYBIND11_LiIon.cpp"
```

Include dependency graph for PYBIND11_PGM.cpp:



Functions

- PYBIND11_MODULE (PGMcpp, m)

5.21.1 Detailed Description

Bindings file for PGMcpp.

Ref: Jakob [2023]

A file which instructs pybind11 how to build Python bindings for PGMcpp. Only public attributes/methods are bound!

5.21.2 Function Documentation

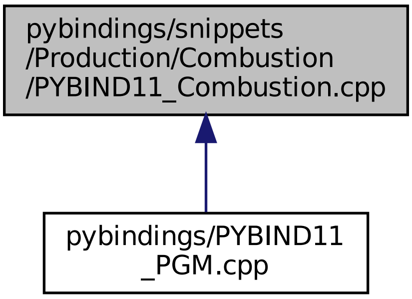
5.21.2.1 PYBIND11_MODULE()

```
PYBIND11_MODULE (
            PGMcpp ,
            m )
                         {

    #include "snippets/PYBIND11_Controller.cpp"
    #include "snippets/PYBIND11_ElectricalLoad.cpp"
    #include "snippets/PYBIND11_Interpolator.cpp"
    #include "snippets/PYBIND11_Model.cpp"
    #include "snippets/PYBIND11_Resources.cpp"

    #include "snippets/Production/PYBIND11_Production.cpp"

    #include "snippets/Production/Noncombustion/PYBIND11_Noncombustion.cpp"
    #include "snippets/Production/Noncombustion/PYBIND11_Hydro.cpp"

    #include "snippets/Production/Combustion/PYBIND11_Combustion.cpp"
    #include "snippets/Production/Combustion/PYBIND11_Diesel.cpp"

    #include "snippets/Production/Renewable/PYBIND11_Renewable.cpp"
    #include "snippets/Production/Renewable/PYBIND11_Solar.cpp"
    #include "snippets/Production/Renewable/PYBIND11_Tidal.cpp"
    #include "snippets/Production/Renewable/PYBIND11_Wave.cpp"
    #include "snippets/Production/Renewable/PYBIND11_Wind.cpp"

    #include "snippets/Storage/PYBIND11_Storage.cpp"
    #include "snippets/Storage/PYBIND11_LiIon.cpp"

}   /* 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

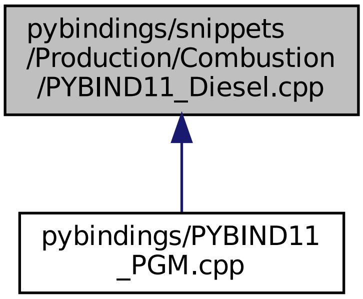
5.22.3.1 def_readwrite

```
&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_intens 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_intens &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  )
```

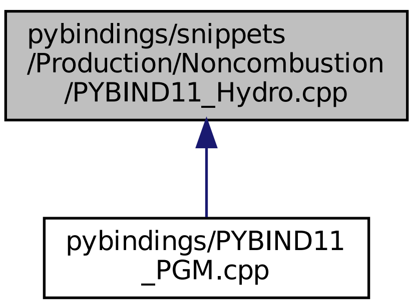
5.23.2.9 def_readwrite() [8/8]

```
& DieselInputs::combustion_inputs def_readwrite (
            "replace_running_hrs" ,
            &DieselInputs::replace_running_hrs  )
```

5.24 pybindings/snippets/Production/Noncombustion/PYBIND11_↩ Hydro.cpp File Reference

Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- HydroTurbineType::HYDRO_TURBINE_PELTON value ("HYDRO_TURBINE_FRANCIS", HydroTurbine↩ Type::HYDRO_TURBINE_FRANCIS) .value("HYDRO_TURBINE_KAPLAN"
- HydroTurbineType::HYDRO_TURBINE_PELTON HydroTurbineType::HYDRO_TURBINE_KAPLAN value ("N_HYDRO_TURBINES", HydroTurbineType::N_HYDRO_TURBINES)
- &HydroInputs::noncombustion_inputs def_readwrite ("resource_key", &HydroInputs::resource_key) .def_↩ readwrite("capital_cost"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost def_readwrite ("operation_maintenance↩ _cost_kWh", &HydroInputs::operation_maintenance_cost_kWh) .def_readwrite("fluid_density_kgm3"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 def_readwrite ("net_head_m", &HydroInputs::net_head_m) .def_readwrite("reservoir_capacity_m3"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 &HydroInputs::reservoir_capacity_m3 def_readwrite ("init_reservoir_state", &HydroInputs::init_reservoir↩ _state) .def_readwrite("turbine_type"
- &HydroInputs::noncombustion_inputs &HydroInputs::capital_cost &HydroInputs::fluid_density_kgm3 &HydroInputs::reservoir_capacity_m3 &HydroInputs::turbine_type def (pybind11::init())
- &Hydro::turbine_type def_readwrite ("fluid_density_kgm3", &Hydro::fluid_density_kgm3) .def_readwrite("net↩ _head_m"
- &Hydro::turbine_type &Hydro::net_head_m def_readwrite ("reservoir_capacity_m3", &Hydro::reservoir_↩ capacity_m3) .def_readwrite("init_reservoir_state"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state def_readwrite ("stored_volume_↩ m3", &Hydro::stored_volume_m3) .def_readwrite("minimum_power_kW"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state &Hydro::minimum_power_kW def_readwrite ("minimum_flow_m3hr", &Hydro::minimum_flow_m3hr) .def_readwrite("maximum_flow_m3hr"
- &Hydro::turbine_type &Hydro::net_head_m &Hydro::init_reservoir_state &Hydro::minimum_power_kW &Hydro::maximum_flow_m3hr def_readwrite ("turbine_flow_vec_m3hr", &Hydro::turbine_flow_vec_m3hr) .def_readwrite("spill_rate_vec_m3hr"

5.24.1 Detailed Description

Bindings file for the Hydro class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Hydro class. Only public attributes/methods are bound!

5.24.2 Function Documentation

5.24.2.1 def()

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 & HydroInputs::turbine_type def (
            pybind11::init()  )
```

5.24.2.2 def_readwrite() [1/9]

```
& Hydro::turbine_type def_readwrite (
            "fluid_density_kgm3" ,
            &Hydro::fluid_density_kgm3 )
```

5.24.2.3 def_readwrite() [2/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
& HydroInputs::reservoir_capacity_m3 def_readwrite (
            "init_reservoir_state" ,
            &HydroInputs::init_reservoir_state )
```

5.24.2.4 def_readwrite() [3/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
def_readwrite (
            "minimum_flow_m3hr" ,
            &Hydro::minimum_flow_m3hr )
```

5.24.2.5 def_readwrite() [4/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost & HydroInputs::fluid_density_kgm3
def_readwrite (
            "net_head_m" ,
            &HydroInputs::net_head_m )
```

5.24.2.6 def_readwrite() [5/9]

```
& HydroInputs::noncombustion_inputs & HydroInputs::capital_cost def_readwrite (
            "operation_maintenance_cost_kWh" ,
            &HydroInputs::operation_maintenance_cost_kWh )
```

5.24.2.7 def_readwrite() [6/9]

```
& Hydro::turbine_type & Hydro::net_head_m def_readwrite (
            "reservoir_capacity_m3" ,
            &Hydro::reservoir_capacity_m3 )
```

5.24.2.8 def_readwrite() [7/9]

```
& HydroInputs::noncombustion_inputs def_readwrite (
            "resource_key" ,
            &HydroInputs::resource_key )
```

5.24.2.9 def_readwrite() [8/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state def_readwrite (
            "stored_volume_m3" ,
            &Hydro::stored_volume_m3 )
```

5.24.2.10 def_readwrite() [9/9]

```
& Hydro::turbine_type & Hydro::net_head_m & Hydro::init_reservoir_state & Hydro::minimum_power_kW
& Hydro::maximum_flow_m3hr def_readwrite (
            "turbine_flow_vec_m3hr" ,
            &Hydro::turbine_flow_vec_m3hr )
```

5.24.2.11 value() [1/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON value (
            "HYDRO_TURBINE_FRANCIS" ,
            HydroTurbineType::HYDRO_TURBINE_FRANCIS )
```

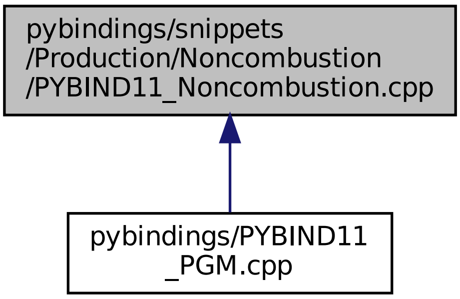
5.24.2.12 value() [2/2]

```
HydroTurbineType::HYDRO_TURBINE_PELTON HydroTurbineType::HYDRO_TURBINE_KAPLAN value (
            "N_HYDRO_TURBINES" ,
            HydroTurbineType::N_HYDRO_TURBINES )
```

5.25 pybindings/snippets/Production/Noncombustion/PYBIND11_↩ Noncombustion.cpp File Reference

Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- NoncombustionType::HYDRO value ("N_NONCOMBUSTION_TYPES", NoncombustionType::N_↩ NONCOMBUSTION_TYPES)
- &NoncombustionInputs::production_inputs def (pybind11::init())

5.25.1 Detailed Description

Bindings file for the Noncombustion class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Noncombustion class. Only public attributes/methods are bound!

5.25.2 Function Documentation

5.25.2.1 def()

```
& NoncombustionInputs::production_inputs def (
            pybind11::init()  )
```

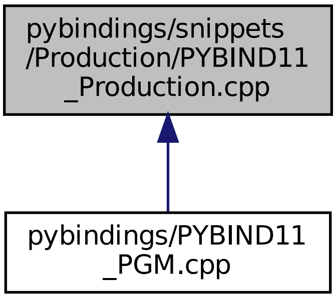
5.25.2.2 value()

```
NoncombustionType::HYDRO value (
            "N_NONCOMBUSTION_TYPES" ,
            NoncombustionType::N_NONCOMBUSTION_TYPES  )
```

5.26 pybindings/snippets/Production/PYBIND11_Production.cpp File Reference

Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &ProductionInputs::print_flag def_readwrite ("is_sunk", &ProductionInputs::is_sunk) .def_readwrite("capacity↩_kW"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW def_readwrite ("nominal_inflation_annual", &ProductionInputs::nominal_inflation_annual) .def_readwrite("nominal_discount_annual"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual def_readwrite ("replace_running_hrs", &ProductionInputs::replace_running_hrs) .def_readwrite("path_2_↩normalized_production_time_series"
- &ProductionInputs::print_flag &ProductionInputs::capacity_kW &ProductionInputs::nominal_discount_annual &ProductionInputs::path_2_normalized_production_time_series def (pybind11::init())
- &Production::interpolator def_readwrite ("print_flag", &Production::print_flag) .def_readwrite("is_running"
- &Production::interpolator &Production::is_running def_readwrite ("is_sunk", &Production::is_sunk) .def_↩readwrite("normalized_production_series_given"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given def_readwrite ("n_points", &Production::n_points) .def_readwrite("n_starts"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts def_readwrite ("n_replacements", &Production::n_replacements) .def_readwrite("n_↩years"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years def_readwrite ("running_hours", &Production::running_hours) .def_readwrite("replace_running_hrs"

- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs def_readwrite ("capacity_↩ kW", &Production::capacity_kW) .def_readwrite("nominal_inflation_annual"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual def_readwrite ("nominal_discount_annual", &Production::nominal_discount_annual) .def_readwrite("real_↩ discount_annual"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual def_readwrite ("capital_cost", &Production::capital_cost) .def_↩ readwrite("operation_maintenance_cost_kWh"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh def_readwrite ("net_↩ present_cost", &Production::net_present_cost) .def_readwrite("total_dispatch_kWh"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh def_readwrite ("levellized_cost_of_energy_kWh", &Production::levellized_cost_of_energy_kWh) .def_↩ readwrite("type_str"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str def_readwrite ("path_2_normalized_production_time_series", &Production::path_2_↩ normalized_production_time_series) .def_readwrite("is_running_vec"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec def_readwrite ("normalized_production_vec", &Production↩ ::normalized_production_vec) .def_readwrite("production_vec_kW"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW def_readwrite ("dispatch_vec_kW", &Production::dispatch_vec_kW) .def_readwrite("storage_vec_kW"
- &Production::interpolator &Production::is_running &Production::normalized_production_series_given &Production::n_starts &Production::n_years &Production::replace_running_hrs &Production::nominal_inflation_annual &Production::real_discount_annual &Production::operation_maintenance_cost_kWh &Production::total_dispatch_kWh &Production::type_str &Production::is_running_vec &Production::production_vec_kW &Production::storage_vec_kW def_readwrite ("curtailment_vec_kW", &Production::curtailment_vec_kW) .def_readwrite("capital_cost_vec"

5.26.1 Detailed Description

Bindings file for the Production class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Production class. Only public attributes/methods are bound!

5.26.2 Function Documentation

5.26.2.1 def()

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
& ProductionInputs::path_2_normalized_production_time_series def (
            pybind11::init()  )
```

5.26.2.2 def_readwrite() [1/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs def_readwrite (
            "capacity_kW" ,
            &Production::capacity_kW  )
```

5.26.2.3 def_readwrite() [2/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual def_readwrite (
            "capital_cost" ,
            &Production::capital_cost  )
```

5.26.2.4 def_readwrite() [3/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW & Production::storage_vec_
def_readwrite (
            "curtailment_vec_kW" ,
            &Production::curtailment_vec_kW  )
```

5.26.2.5 def_readwrite() [4/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec & Production::production_vec_kW def_↩
readwrite (
            "dispatch_vec_kW" ,
            &Production::dispatch_vec_kW  )
```

5.26.2.6 def_readwrite() [5/17]

```
& Production::interpolator & Production::is_running def_readwrite (
            "is_sunk" ,
            &Production::is_sunk )
```

5.26.2.7 def_readwrite() [6/17]

```
& ProductionInputs::print_flag def_readwrite (
            "is_sunk" ,
            &ProductionInputs::is_sunk )
```

5.26.2.8 def_readwrite() [7/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
def_readwrite (
            "levellized_cost_of_energy_kWh" ,
            &Production::levellized_cost_of_energy_kWh )
```

5.26.2.9 def_readwrite() [8/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
def_readwrite (
            "n_points" ,
            &Production::n_points )
```

5.26.2.10 def_readwrite() [9/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts def_readwrite (
            "n_replacements" ,
            &Production::n_replacements )
```

5.26.2.11 def_readwrite() [10/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh def_readwrite
(
            "net_present_cost" ,
            &Production::net_present_cost  )
```

5.26.2.12 def_readwrite() [11/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
def_readwrite (
            "nominal_discount_annual" ,
            &Production::nominal_discount_annual  )
```

5.26.2.13 def_readwrite() [12/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW def_readwrite (
            "nominal_inflation_annual" ,
            &ProductionInputs::nominal_inflation_annual  )
```

5.26.2.14 def_readwrite() [13/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str & Production::is_running_vec def_readwrite (
            "normalized_production_vec" ,
            &Production::normalized_production_vec  )
```

5.26.2.15 def_readwrite() [14/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years & Production::replace_running_hrs & Production::nominal_inflatior
& Production::real_discount_annual & Production::operation_maintenance_cost_kWh & Production::total_dispatch_k
& Production::type_str def_readwrite (
            "path_2_normalized_production_time_series" ,
            &Production::path_2_normalized_production_time_series  )
```

5.26.2.16 def_readwrite() [15/17]

```
& Production::interpolator def_readwrite (
            "print_flag" ,
            &Production::print_flag )
```

5.26.2.17 def_readwrite() [16/17]

```
& ProductionInputs::print_flag & ProductionInputs::capacity_kW & ProductionInputs::nominal_discount_annual
def_readwrite (
            "replace_running_hrs" ,
            &ProductionInputs::replace_running_hrs )
```

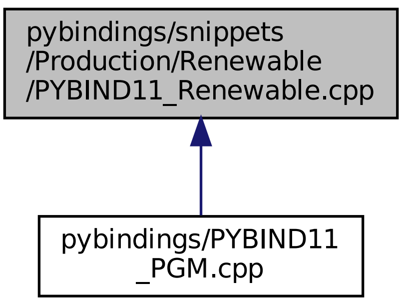
5.26.2.18 def_readwrite() [17/17]

```
& Production::interpolator & Production::is_running & Production::normalized_production_series_given
& Production::n_starts & Production::n_years def_readwrite (
            "running_hours" ,
            &Production::running_hours )
```

5.27 pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp File Reference

Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- RenewableType::SOLAR value ("TIDAL", RenewableType::TIDAL) .value("WAVE"
- RenewableType::SOLAR RenewableType::WAVE value ("WIND", RenewableType::WIND) .value("N_↩ RENEWABLE_TYPES"
- &RenewableInputs::production_inputs def (pybind11::init())

5.27.1 Detailed Description

Bindings file for the Renewable class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Renewable class. Only public attributes/methods are bound!

5.27.2 Function Documentation

5.27.2.1 def()

```
& RenewableInputs::production_inputs def (
            pybind11::init()  )
```

5.27.2.2 value() [1/2]

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

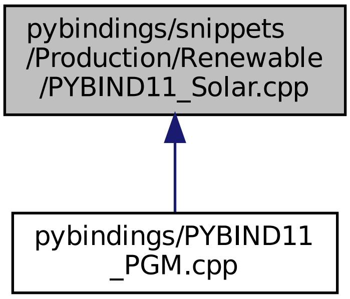
5.27.2.3 value() [2/2]

```
RenewableType::SOLAR RenewableType::WAVE value (
            "WIND" ,
            RenewableType::WIND  )
```

5.28 pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp File Reference

Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &SolarInputs::renewable_inputs def_readwrite ("resource_key", &SolarInputs::resource_key) .def_↩ readwrite("capital_cost"
- &SolarInputs::renewable_inputs &SolarInputs::capital_cost def_readwrite ("operation_maintenance_cost_↩ kWh", &SolarInputs::operation_maintenance_cost_kWh) .def_readwrite("derating"
- &SolarInputs::renewable_inputs &SolarInputs::capital_cost &SolarInputs::derating def (pybind11::init())

5.28.1 Detailed Description

Bindings file for the Solar class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Solar class. Only public attributes/methods are bound!

5.28.2 Function Documentation

5.28.2.1 def()

```
& SolarInputs::renewable_inputs & SolarInputs::capital_cost & SolarInputs::derating def (
            pybind11::init()  )
```

5.28.2.2 def_readwrite() [1/2]

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

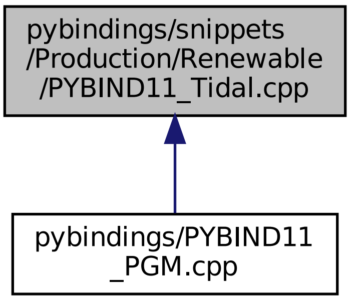
5.28.2.3 def_readwrite() [2/2]

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

5.29 pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp File Reference

Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- TidalPowerProductionModel::TIDAL_POWER_CUBIC value ("TIDAL_POWER_EXPONENTIAL", Tidal↩ PowerProductionModel::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_k↩ Wh", &TidalInputs::operation_maintenance_cost_kWh) .def_readwrite("design_speed_ms"

Variables

- &TidalInputs::renewable_inputs &TidalInputs::capital_cost &TidalInputs::design_speed_ms def_readwrite("power←_model", &TidalInputs::power_model) .def(pybind11 &Tidal::design_speed_ms def_readwrite ("power_←model", &Tidal::power_model) .def_readwrite("power_model_string"

5.29.1 Detailed Description

Bindings file for the Tidal class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Tidal class. Only public attributes/methods are bound!

5.29.2 Function Documentation

5.29.2.1 def_readwrite() [1/2]

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

5.29.2.2 def_readwrite() [2/2]

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

5.29.2.3 value() [1/2]

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

5.29.2.4 value() [2/2]

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

5.29.3 Variable Documentation

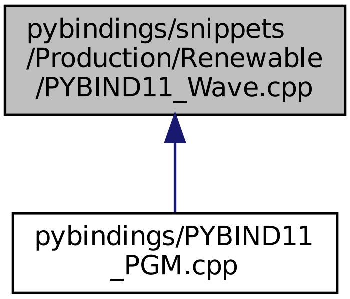
5.29.3.1 def_readwrite

```
& TidalInputs::renewable_inputs & TidalInputs::capital_cost & TidalInputs::design_speed_ms
def_readwrite ("power_model", &TidalInputs::power_model) .def(pybind11 & Tidal::design_speed_ms
def_readwrite("power_model", &Tidal::power_model) .def_readwrite("power_model_string" (
            "power_model" ,
            &Tidal::power_model  )
```

5.30 pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp File Reference

Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- WavePowerProductionModel::WAVE_POWER_GAUSSIAN value ("WAVE_POWER_PARABOLOID", WavePowerProductionModel::WAVE_POWER_PARABOLOID) .value("WAVE_POWER_LOOKUP"
- WavePowerProductionModel::WAVE_POWER_GAUSSIAN WavePowerProductionModel::WAVE_POWER_LOOKUP value ("N_WAVE_POWER_PRODUCTION_MODELS", WavePowerProductionModel::N_WAVE_POWER←_PRODUCTION_MODELS)
- &WaveInputs::renewable_inputs def_readwrite ("resource_key", &WaveInputs::resource_key) .def_←readwrite("capital_cost"
- &WaveInputs::renewable_inputs &WaveInputs::capital_cost def_readwrite ("operation_maintenance_cost_←kWh", &WaveInputs::operation_maintenance_cost_kWh) .def_readwrite("design_significant_wave_height←_m"
- &WaveInputs::renewable_inputs &WaveInputs::capital_cost &WaveInputs::design_significant_wave_height_m def_readwrite ("design_energy_period_s", &WaveInputs::design_energy_period_s) .def_readwrite("power←_model"

Variables

- &WaveInputs::renewable_inputs &WaveInputs::capital_cost &WaveInputs::design_significant_wave_height_m &WaveInputs::power_model def_readwrite("path_2_normalized_performance_matrix", &WaveInputs↩::path_2_normalized_performance_matrix) .def(pybind11 &Wave::design_significant_wave_height_m def_readwrite ("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power_model"

5.30.1 Detailed Description

Bindings file for the Wave class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Wave class. Only public attributes/methods are bound!

5.30.2 Function Documentation

5.30.2.1 def_readwrite() [1/3]

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
def_readwrite (
            "design_energy_period_s" ,
            &WaveInputs::design_energy_period_s )
```

5.30.2.2 def_readwrite() [2/3]

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

5.30.2.3 def_readwrite() [3/3]

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

5.30.2.4 value() [1/2]

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

5.30.2.5 value() [2/2]

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

5.30.3 Variable Documentation

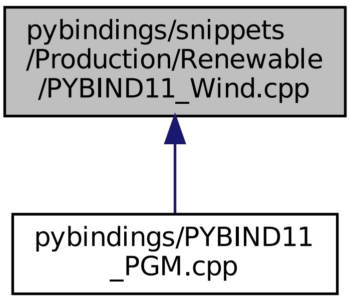
5.30.3.1 def_readwrite

```
& WaveInputs::renewable_inputs & WaveInputs::capital_cost & WaveInputs::design_significant_wave_height_m
& WaveInputs::power_model def_readwrite ( "path_2_normalized_performance_matrix", &Wave↩
Inputs::path_2_normalized_performance_matrix ) .def(pybind11 & Wave::design_significant_wave_height_m
def_readwrite("design_energy_period_s", &Wave::design_energy_period_s) .def_readwrite("power↩
_model" (
            "design_energy_period_s" ,
            &Wave::design_energy_period_s   )
```

5.31 pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp File Reference

Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- WindPowerProductionModel::WIND_POWER_EXPONENTIAL value ("WIND_POWER_LOOKUP", 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"

5.31.1 Detailed Description

Bindings file for the Wind class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Wind class. Only public attributes/methods are bound!

5.31.2 Function Documentation

5.31.2.1 def_readwrite() [1/2]

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

5.31.2.2 def_readwrite() [2/2]

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

5.31.2.3 value()

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

5.31.3 Variable Documentation

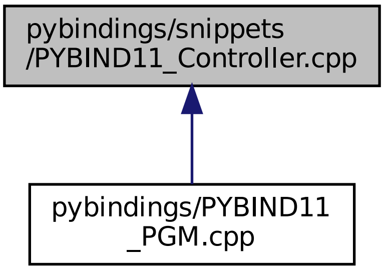
5.31.3.1 def_readwrite

```
& WindInputs::renewable_inputs & WindInputs::capital_cost & WindInputs::design_speed_ms def↩
_readwrite ("power_model", &WindInputs::power_model) .def(pybind11 & Wind::design_speed_ms
def_readwrite("power_model", &Wind::power_model) .def_readwrite("power_model_string" (
            "power_model" ,
            &Wind::power_model  )
```

5.32 pybindings/snippets/PYBIND11_Controller.cpp File Reference

Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- ControlMode::LOAD_FOLLOWING value ("CYCLE_CHARGING", ControlMode::CYCLE_CHARGING) .value("N_CONTROL_MODES"
- &Controller::control_mode def_readwrite ("control_string", &Controller::control_string) .def_readwrite("net↩ _load_vec_kW"
- &Controller::control_mode &Controller::net_load_vec_kW def_readwrite ("missed_load_vec_kW", &Controller↩ ::missed_load_vec_kW) .def_readwrite("combustion_map"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map def (pybind11↩ ::init<>()) .def("setControlMode"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode def ("init", &Controller::init) .def("applyDispatchControl"
- &Controller::control_mode &Controller::net_load_vec_kW &Controller::combustion_map &Controller::setControlMode &Controller::applyDispatchControl def ("clear", &Controller::clear)

5.32.1 Detailed Description

Bindings file for the Controller class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Controller class. Only public attributes/methods are bound!

5.32.2 Function Documentation

5.32.2.1 def() [1/3]

```
& Controller::control_mode & Controller::net_load_vec_kW & Controller::combustion_map & Controller::setControl
& Controller::applyDispatchControl def (
            "clear" ,
            &Controller::clear )
```

5.32.2.2 def() [2/3]

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

5.32.2.3 def() [3/3]

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

5.32.2.4 def_readwrite() [1/2]

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

5.32.2.5 def_readwrite() [2/2]

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

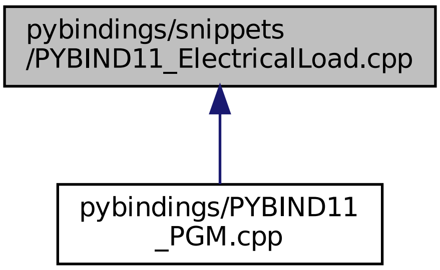
5.32.2.6 value()

```
ControlMode::LOAD_FOLLOWING value (
            "CYCLE_CHARGING" ,
            ControlMode::CYCLE_CHARGING )
```

5.33 pybindings/snippets/PYBIND11_ElectricalLoad.cpp File Reference

Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &ElectricalLoad::n_points def_readwrite ("n_years", &ElectricalLoad::n_years) .def_readwrite("min_load_kW"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW def_readwrite ("mean_load_kW", &ElectricalLoad::mean_load_kW) .def_readwrite("max_load_kW"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW &ElectricalLoad::max_load_kW def_readwrite ("path_2_electrical_load_time_series", &ElectricalLoad::path_2_electrical_load_time_series) .def_readwrite("time_vec_hrs"
- &ElectricalLoad::n_points &ElectricalLoad::min_load_kW &ElectricalLoad::max_load_kW &ElectricalLoad::time_vec_hrs def_readwrite ("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs) .def_readwrite("load_vec_kW"

5.33.1 Detailed Description

Bindings file for the ElectricalLoad class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the ElectricalLoad class. Only public attributes/methods are bound!

5.33.2 Function Documentation

5.33.2.1 def_readwrite() [1/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW & ElectricalLoad::max_load_kW & ElectricalLoad::time_
def_readwrite (
            "dt_vec_hrs" ,
            &ElectricalLoad::dt_vec_hrs )
```

5.33.2.2 def_readwrite() [2/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW def_readwrite (
            "mean_load_kW" ,
            &ElectricalLoad::mean_load_kW )
```

5.33.2.3 def_readwrite() [3/4]

```
& ElectricalLoad::n_points def_readwrite (
            "n_years" ,
            &ElectricalLoad::n_years )
```

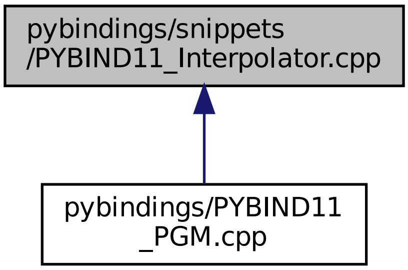
5.33.2.4 def_readwrite() [4/4]

```
& ElectricalLoad::n_points & ElectricalLoad::min_load_kW & ElectricalLoad::max_load_kW def_←
readwrite (
            "path_2_electrical_load_time_series" ,
            &ElectricalLoad::path_2_electrical_load_time_series )
```

5.34 pybindings/snippets/PYBIND11_Interpolator.cpp File Reference

Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &InterpolatorStruct1D::n_points def_readwrite ("x_vec", &InterpolatorStruct1D::x_vec) .def_readwrite("min←
_x"
- &InterpolatorStruct1D::n_points &InterpolatorStruct1D::min_x def_readwrite ("max_x", &Interpolator←
Struct1D::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", &InterpolatorStruct2←
D::min_x) .def_readwrite("max_x"
- &InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x def_readwrite
("y_vec", &InterpolatorStruct2D::y_vec) .def_readwrite("min_y"
- &InterpolatorStruct2D::n_rows &InterpolatorStruct2D::x_vec &InterpolatorStruct2D::max_x &InterpolatorStruct2D::min_y
def_readwrite ("max_y", &InterpolatorStruct2D::max_y) .def_readwrite("z_matrix"
- &Interpolator::interp_map_1D def_readwrite ("path_map_1D", &Interpolator::path_map_1D) .def_←
readwrite("interp_map_2D"

5.34.1 Detailed Description

Bindings file for the Interpolator class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Interpolator class. Only public attributes/methods are bound!

5.34.2 Function Documentation

5.34.2.1 def()

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x & InterpolatorStruct1D::y_vec
def (
            pybind11::init() )
```

5.34.2.2 def_readwrite() [1/7]

```
& InterpolatorStruct1D::n_points & InterpolatorStruct1D::min_x def_readwrite (
            "max_x" ,
            &InterpolatorStruct1D::max_x )
```

5.34.2.3 def_readwrite() [2/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x &
InterpolatorStruct2D::min_y def_readwrite (
            "max_y" ,
            &InterpolatorStruct2D::max_y )
```

5.34.2.4 def_readwrite() [3/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec def_readwrite (
            "min_x" ,
            &InterpolatorStruct2D::min_x )
```

5.34.2.5 def_readwrite() [4/7]

```
& InterpolatorStruct2D::n_rows def_readwrite (
            "n_cols" ,
            &InterpolatorStruct2D::n_cols )
```

5.34.2.6 def_readwrite() [5/7]

```
& Interpolator::interp_map_1D def_readwrite (
            "path_map_1D" ,
            &Interpolator::path_map_1D )
```

5.34.2.7 def_readwrite() [6/7]

```
& InterpolatorStruct1D::n_points def_readwrite (
            "x_vec" ,
            &InterpolatorStruct1D::x_vec )
```

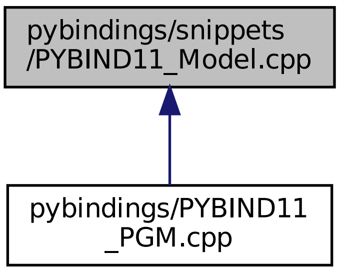
5.34.2.8 def_readwrite() [7/7]

```
& InterpolatorStruct2D::n_rows & InterpolatorStruct2D::x_vec & InterpolatorStruct2D::max_x
def_readwrite (
            "y_vec" ,
            &InterpolatorStruct2D::y_vec )
```

5.35 pybindings/snippets/PYBIND11_Model.cpp File Reference

Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Variables

- &ModelInputs::path_2_electrical_load_time_series def_readwrite("control_mode", &ModelInputs::control_↩ mode) .def(pybind11 &Model::total_fuel_consumed_L def_readwrite ("total_emissions", &Model::total_↩ emissions) .def_readwrite("net_present_cost"

5.35.1 Detailed Description

Bindings file for the Model class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Model class. Only public attributes/methods are bound!

5.35.2 Variable Documentation

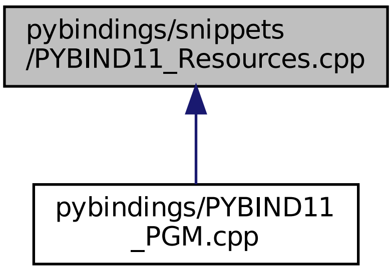
5.35.2.1 def_readwrite

```
& ModelInputs::path_2_electrical_load_time_series def_readwrite ("control_mode", &Model←
Inputs::control_mode) .def(pybind11 & Model::total_fuel_consumed_L & Model::net_present_cost &
Model::total_dispatch_discharge_kWh & Model::controller & Model::resources & Model::noncombustion_ptr_vec
def_readwrite("renewable_ptr_vec", &Model::renewable_ptr_vec) .def_readwrite("storage_ptr_vec"
(
            "total_emissions" ,
            &Model::total_emissions  )
```

5.36 pybindings/snippets/PYBIND11_Resources.cpp File Reference

Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &Resources::resource_map_1D def_readwrite ("string_map_1D", &Resources::string_map_1D) .def_←
readwrite("path_map_1D"
- &Resources::resource_map_1D &Resources::path_map_1D def_readwrite ("resource_map_2D", &Resources←
::resource_map_2D) .def_readwrite("string_map_2D"

5.36.1 Detailed Description

Bindings file for the Resources class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Resources class. Only public attributes/methods are bound!

5.36.2 Function Documentation

5.36.2.1 def_readwrite() [1/2]

```
& Resources::resource_map_1D & Resources::path_map_1D def_readwrite (
            "resource_map_2D" ,
            &Resources::resource_map_2D  )
```

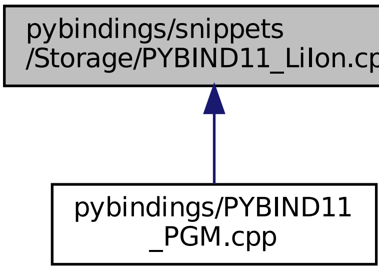
5.36.2.2 def_readwrite() [2/2]

```
& Resources::resource_map_1D def_readwrite (
            "string_map_1D" ,
            &Resources::string_map_1D  )
```

5.37 pybindings/snippets/Storage/PYBIND11_LiIon.cpp File Reference

Bindings file for the LiIon class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- &LiIonInputs::storage_inputs def_readwrite ("capital_cost", &LiIonInputs::capital_cost) .def_readwrite("operation←_maintenance_cost_kWh"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh def_readwrite ("init_SOC", &LiIonInputs::init_SOC) .def_readwrite("min_SOC"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC def_readwrite ("hysteresis_SOC", &LiIonInputs::hysteresis_SOC) .def_readwrite("max_SOC"

- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC def_readwrite ("charging_efficiency", &LiIonInputs::charging_efficiency) .def_readwrite("discharging_efficiency"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC &LiIonInputs::discharging_efficiency def_readwrite ("replace_SOH", &LiIonInputs::replace_SOH) .def_readwrite("degradation_alpha"
- &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) .def_readwrite("degradation_B_hat_cal_0"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC &LiIonInputs::discharging_efficiency &LiIonInputs::degradation_alpha &LiIonInputs::degradation_B_ha def_readwrite ("degradation_r_cal", &LiIonInputs::degradation_r_cal) .def_readwrite("degradation_Ea_cal_0"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC &LiIonInputs::discharging_efficiency &LiIonInputs::degradation_alpha &LiIonInputs::degradation_B_ha &LiIonInputs::degradation_Ea_cal_0 def_readwrite ("degradation_a_cal", &LiIonInputs::degradation_a_cal) .def_readwrite("degradation_s_cal"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC &LiIonInputs::discharging_efficiency &LiIonInputs::degradation_alpha &LiIonInputs::degradation_B_ha &LiIonInputs::degradation_Ea_cal_0 &LiIonInputs::degradation_s_cal def_readwrite ("gas_constant_JmolK", &LiIonInputs::gas_constant_JmolK) .def_readwrite("gas_constant_JmolK"
- &LiIonInputs::storage_inputs &LiIonInputs::operation_maintenance_cost_kWh &LiIonInputs::min_SOC &LiIonInputs::max_SOC &LiIonInputs::discharging_efficiency &LiIonInputs::degradation_alpha &LiIonInputs::degradation_B_ha &LiIonInputs::degradation_Ea_cal_0 &LiIonInputs::degradation_s_cal &LiIonInputs::gas_constant_JmolK def (pybind11::init())
- &LiIon::dynamic_energy_capacity_kWh def_readwrite ("SOH", &LiIon::SOH) .def_readwrite("replace_SOH"
- &LiIon::dynamic_energy_capacity_kWh &LiIon::replace_SOH def_readwrite ("degradation_alpha", &LiIon::degradation_alpha) .def_readwrite("degradation_beta"
- &LiIon::dynamic_energy_capacity_kWh &LiIon::replace_SOH &LiIon::degradation_beta def_readwrite ("degradation_B_hat_cal_0", &LiIon::degradation_B_hat_cal_0) .def_readwrite("degradation_r_cal"
- &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) .def_readwrite("degradation_a_cal"
- &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) .def_readwrite("gas_constant_JmolK"
- &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) .def_readwrite("init_SOC"
- &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) .def_readwrite("hysteresis_SOC"
- &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) .def_readwrite("charging_efficiency"
- &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) .def_readwrite("SOH_vec"

5.37.1 Detailed Description

Bindings file for the LiIon 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 LiIon class. Only public attributes/methods are bound!

5.37.2 Function Documentation

5.37.2.1 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()  )
```

5.37.2.2 def_readwrite() [1/18]

```
& LiIonInputs::storage_inputs def_readwrite (
            "capital_cost" ,
            &LiIonInputs::capital_cost  )
```

5.37.2.3 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  )
```

5.37.2.4 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  )
```

5.37.2.5 def_readwrite() [4/18]

```
& LiIon::dynamic_energy_capacity_kWh & LiIon::replace_SOH def_readwrite (
            "degradation_alpha" ,
            &LiIon::degradation_alpha  )
```

5.37.2.6 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 )
```

5.37.2.7 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 )
```

5.37.2.8 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 )
```

5.37.2.9 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 )
```

5.37.2.10 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 )
```

5.37.2.11 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 )
```

5.37.2.12 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 )
```

5.37.2.13 def_readwrite() [12/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh & LiIonInputs::min_SOC
def_readwrite (
            "hysteresis_SOC" ,
            &LiIonInputs::hysteresis_SOC )
```

5.37.2.14 def_readwrite() [13/18]

```
& LiIonInputs::storage_inputs & LiIonInputs::operation_maintenance_cost_kWh def_readwrite (
            "init_SOC" ,
            &LiIonInputs::init_SOC )
```

5.37.2.15 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 )
```

5.37.2.16 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 )
```

5.37.2.17 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 )
```

5.37.2.18 def_readwrite() [17/18]

```
& LiIon::dynamic_energy_capacity_kWh def_readwrite (
            "SOH" ,
            &LiIon::SOH )
```

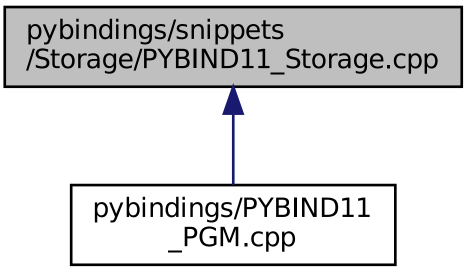
5.37.2.19 def_readwrite() [18/18]

```
& 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 )
```

5.38 pybindings/snippets/Storage/PYBIND11_Storage.cpp File Reference

Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp.

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



Functions

- StorageType::LIION value ("N_STORAGE_TYPES", StorageType::N_STORAGE_TYPES)
- &StorageInputs::print_flag def_readwrite ("is_sunk", &StorageInputs::is_sunk) .def_readwrite("power_↩ capacity_kW"
- &StorageInputs::print_flag &StorageInputs::power_capacity_kW def_readwrite ("energy_capacity_kWh", &StorageInputs::energy_capacity_kWh) .def_readwrite("nominal_inflation_annual"

Variables

- &StorageInputs::print_flag &StorageInputs::power_capacity_kW &StorageInputs::nominal_inflation_annual def_readwrite("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11 &Storage::type def_readwrite ("interpolator", &Storage::interpolator) .def_readwrite("print_flag"

5.38.1 Detailed Description

Bindings file for the Storage class. Intended to be #include'd in PYBIND11_PGM.cpp.

Ref: Jakob [2023]
A file which instructs pybind11 how to build Python bindings for the Storage class. Only public attributes/methods are bound!

5.38.2 Function Documentation

5.38.2.1 def_readwrite() [1/2]

```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW def_readwrite (
            "energy_capacity_kWh" ,
            &StorageInputs::energy_capacity_kWh )
```

5.38.2.2 def_readwrite() [2/2]

```
& StorageInputs::print_flag def_readwrite (
            "is_sunk" ,
            &StorageInputs::is_sunk )
```

5.38.2.3 value()

```
StorageType::LIION value (
            "N_STORAGE_TYPES" ,
            StorageType::N_STORAGE_TYPES )
```

5.38.3 Variable Documentation

5.38.3.1 def_readwrite

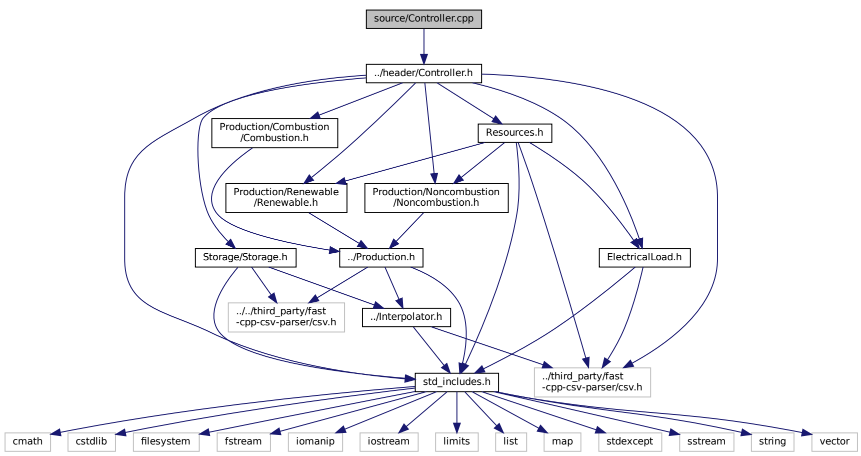
```
& StorageInputs::print_flag & StorageInputs::power_capacity_kW & StorageInputs::nominal_inflation_annual
def_readwrite ("nominal_discount_annual", &StorageInputs::nominal_discount_annual) .def(pybind11
& Storage::type & Storage::print_flag & Storage::is_sunk & Storage::n_replacements & Storage::power_capacity_k
& Storage::charge_kWh & Storage::nominal_inflation_annual & Storage::real_discount_annual &
Storage::operation_maintenance_cost_kWh & Storage::total_discharge_kWh & Storage::type_str &
Storage::charging_power_vec_kW def_readwrite("discharging_power_vec_kW", &Storage::discharging←
_power_vec_kW) .def_readwrite("capital_cost_vec" (
             "interpolator" ,
             &Storage::interpolator  )
```

5.39 source/Controller.cpp File Reference

Implementation file for the Controller class.

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

Include dependency graph for Controller.cpp:



5.39.1 Detailed Description

Implementation file for the Controller class.

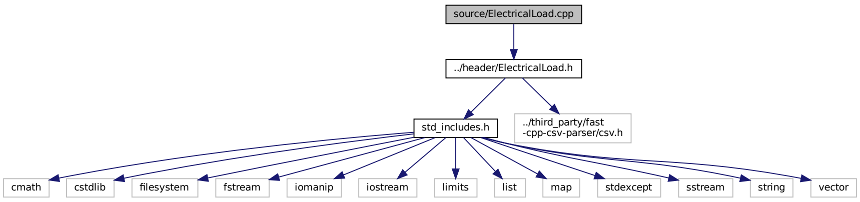
A class which contains a various dispatch control logic. Intended to serve as a component class of Controller.

5.40 source/ElectricalLoad.cpp File Reference

Implementation file for the ElectricalLoad class.

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

Include dependency graph for ElectricalLoad.cpp:



5.40.1 Detailed Description

Implementation file for the ElectricalLoad class.

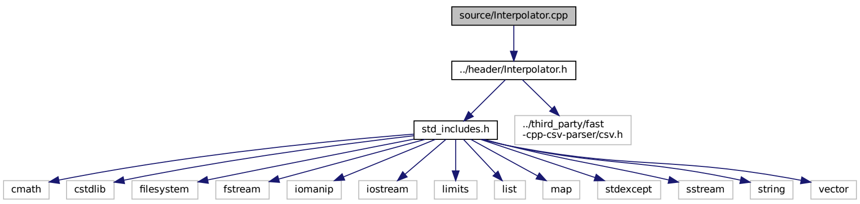
A class which contains time and electrical load data. Intended to serve as a component class of Model.

5.41 source/Interpolator.cpp File Reference

Implementation file for the Interpolator class.

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

Include dependency graph for Interpolator.cpp:



5.41.1 Detailed Description

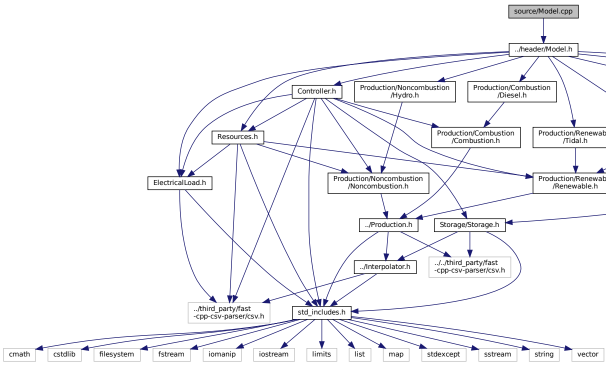
Implementation file for the Interpolator class.

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

5.42 source/Model.cpp File Reference

Implementation file for the Model class.

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



5.42.1 Detailed Description

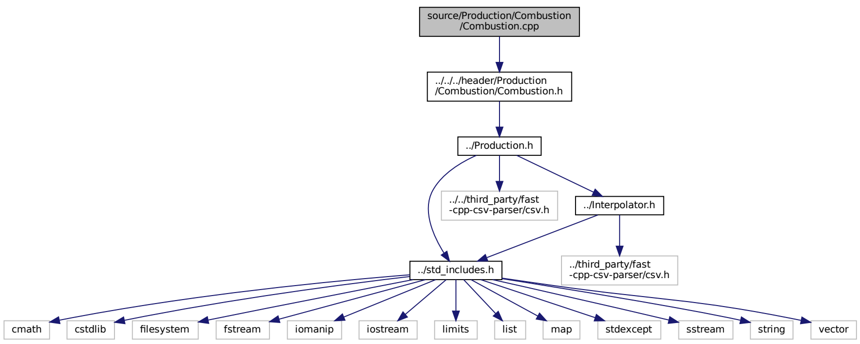
Implementation file for the Model class.

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

5.43 source/Production/Combustion/Combustion.cpp File Reference

Implementation file for the Combustion class.

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



5.43.1 Detailed Description

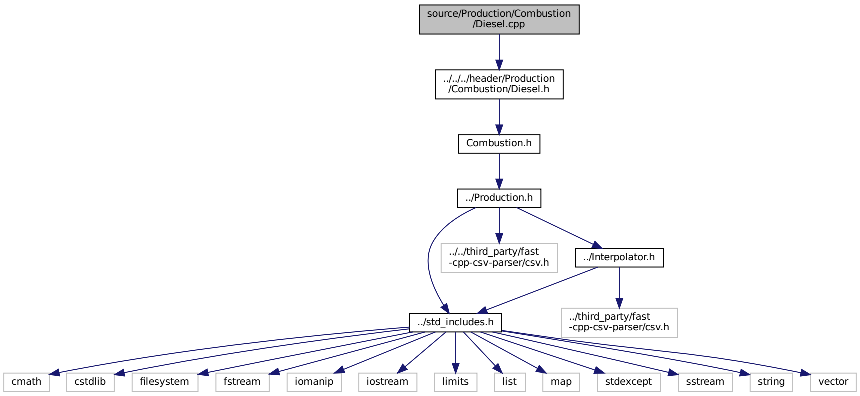
Implementation file for the Combustion class.

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

5.44 source/Production/Combustion/Diesel.cpp File Reference

Implementation file for the Diesel class.

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



5.44.1 Detailed Description

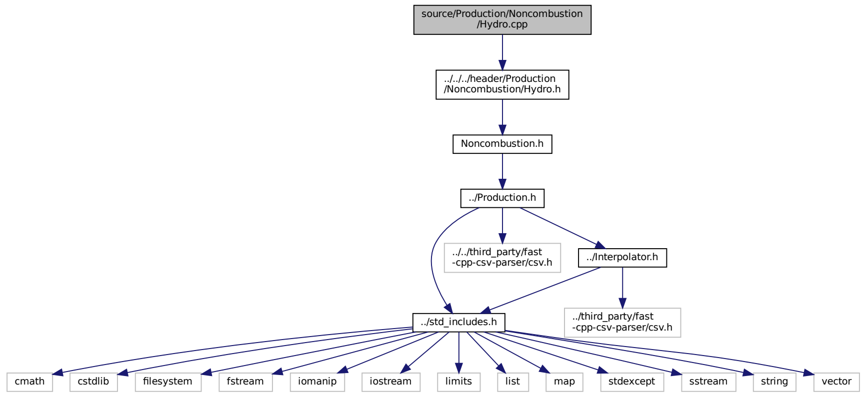
Implementation file for the Diesel class.

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

5.45 source/Production/Noncombustion/Hydro.cpp File Reference

Implementation file for the Hydro class.

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



5.45.1 Detailed Description

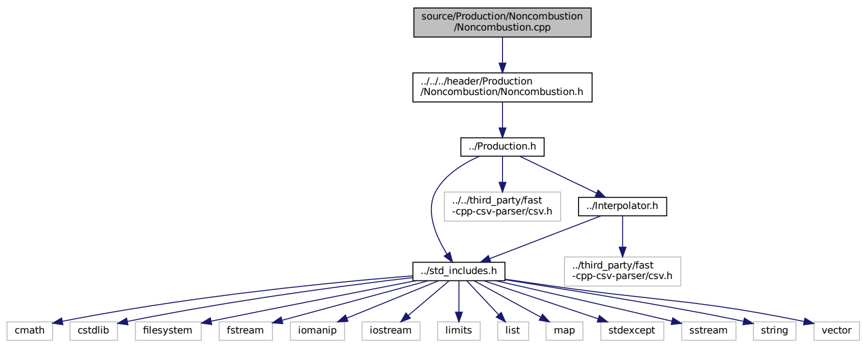
Implementation file for the Hydro class.

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

5.46 source/Production/Noncombustion/Noncombustion.cpp File Reference

Implementation file for the Noncombustion class.

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



5.46.1 Detailed Description

Implementation file for the Noncombustion class.

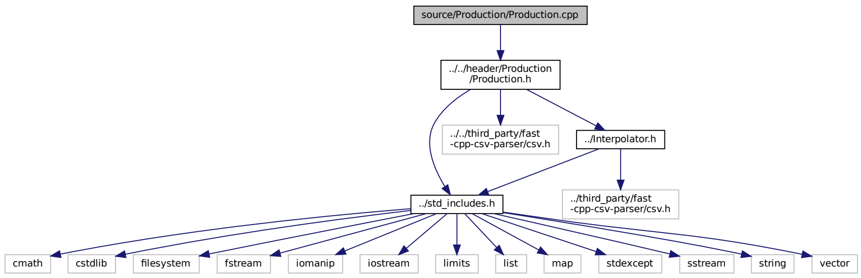
The root of the Noncombustion branch of the Production hierarchy. This branch contains derived classes which model controllable production which is not based on combustion.

5.47 source/Production/Production.cpp File Reference

Implementation file for the Production class.

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

Include dependency graph for Production.cpp:



5.47.1 Detailed Description

Implementation file for the Production class.

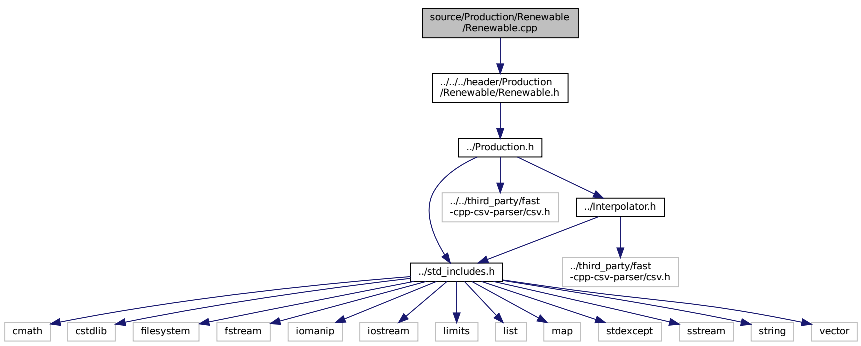
The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.48 source/Production/Renewable/Renewable.cpp File Reference

Implementation file for the Renewable class.

`#include "../../../header/Production/Renewable/Renewable.h"`

Include dependency graph for Renewable.cpp:



5.48.1 Detailed Description

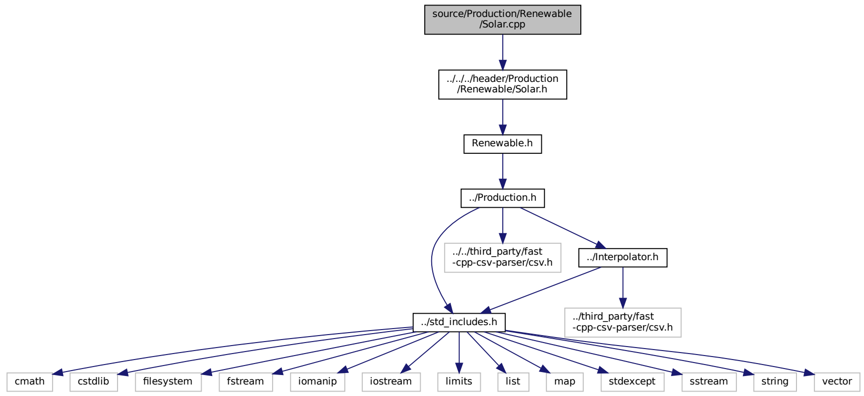
Implementation file for the Renewable class.

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

5.49 source/Production/Renewable/Solar.cpp File Reference

Implementation file for the Solar class.

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



5.49.1 Detailed Description

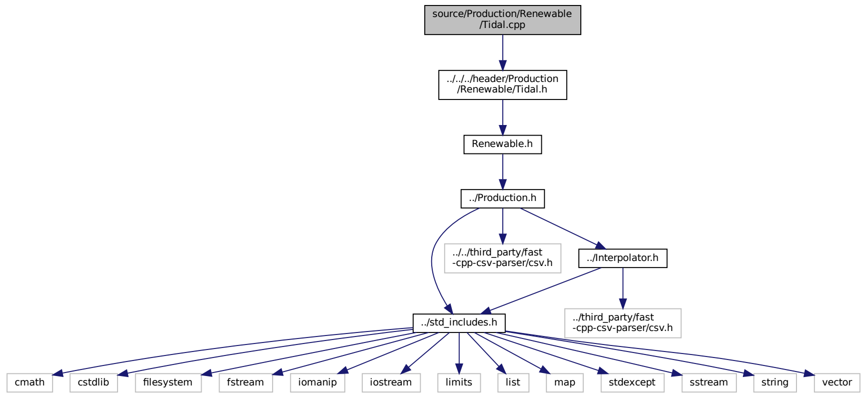
Implementation file for the Solar class.

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

5.50 source/Production/Renewable/Tidal.cpp File Reference

Implementation file for the Tidal class.

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



5.50.1 Detailed Description

Implementation file for the Tidal class.

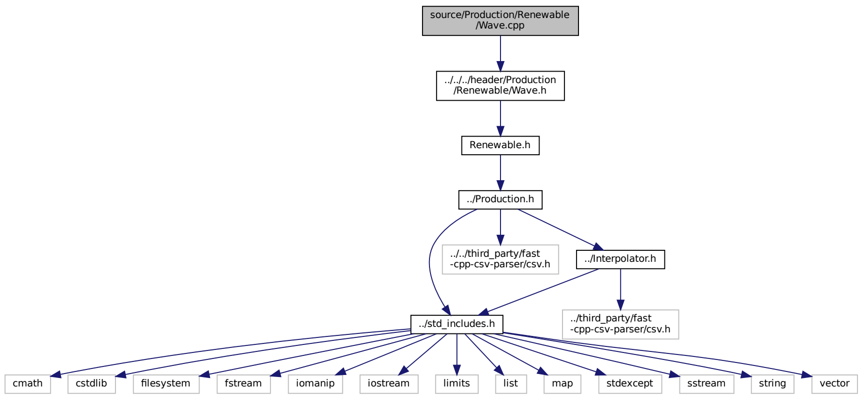
A derived class of the Renewable branch of Production which models tidal production.

5.51 source/Production/Renewable/Wave.cpp File Reference

Implementation file for the Wave class.

`#include "../../../header/Production/Renewable/Wave.h"`

Include dependency graph for Wave.cpp:



5.51.1 Detailed Description

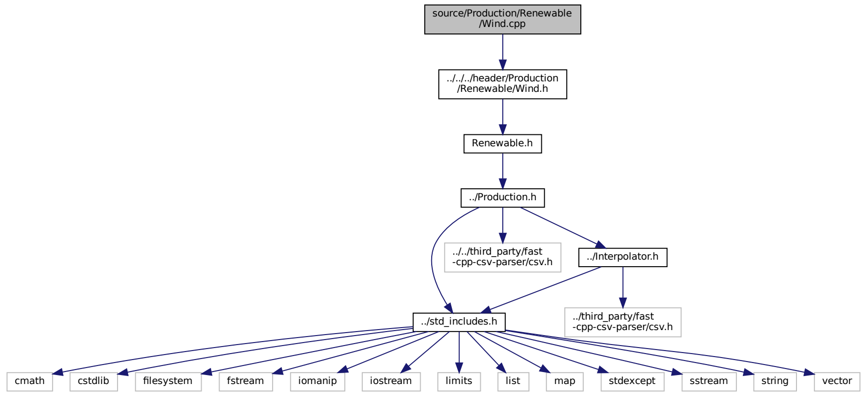
Implementation file for the Wave class.

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

5.52 source/Production/Renewable/Wind.cpp File Reference

Implementation file for the Wind class.

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



5.52.1 Detailed Description

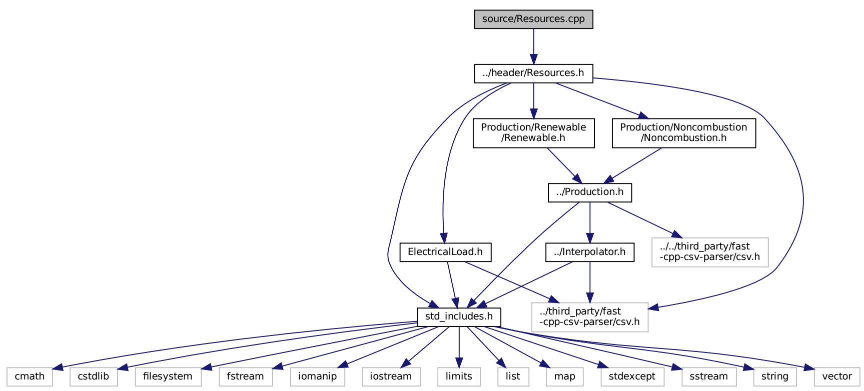
Implementation file for the Wind class.

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

5.53 source/Resources.cpp File Reference

Implementation file for the Resources class.

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



5.53.1 Detailed Description

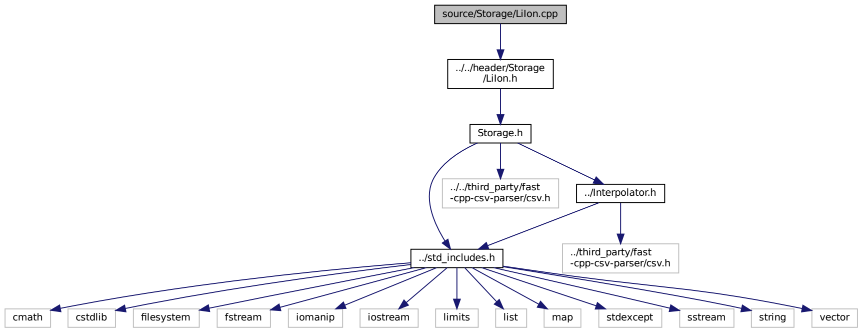
Implementation file for the Resources class.

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

5.54 source/Storage/LiIon.cpp File Reference

Implementation file for the LiIon class.

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



5.54.1 Detailed Description

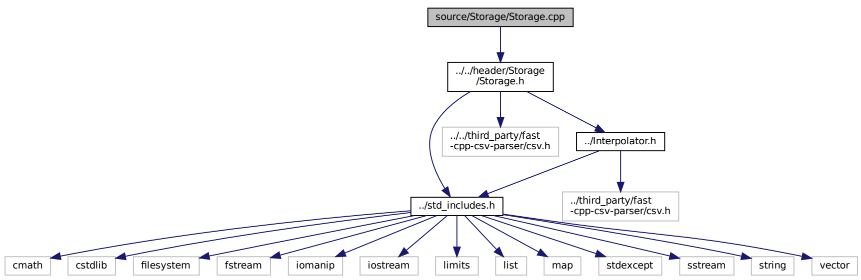
Implementation file for the LiIon class.

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

5.55 source/Storage/Storage.cpp File Reference

Implementation file for the Storage class.

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



5.55.1 Detailed Description

Implementation file for the Storage class.

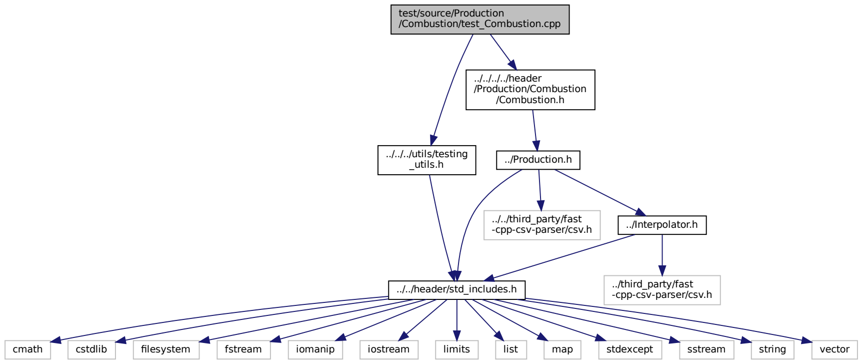
The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy.

5.56 test/source/Production/Combustion/test_Combustion.cpp File Reference

Testing suite for Combustion class.

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

Include dependency graph for test_Combustion.cpp:



Functions

- Combustion ∗ testConstruct_Combustion (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Combustion object and spot check some post-construction attributes.
- int main (int argc, char ∗∗argv)

5.56.1 Detailed Description

Testing suite for Combustion class.

A suite of tests for the Combustion class.

5.56.2 Function Documentation

5.56.2.1 main()

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

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Combustion");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Combustion* test_combustion_ptr = testConstruct_Combustion(&time_vec_hrs);


    try {
        //...
    }


    catch (...) {
        delete test_combustion_ptr;

        printGold(" ................ ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_combustion_ptr;

    printGold(" ................ ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.56.2.2 testConstruct_Combustion()

```
Combustion * testConstruct_Combustion (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Combustion object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Combustion object.

```
{
    CombustionInputs combustion_inputs;

    Combustion* test_combustion_ptr = new Combustion(
        8760,
        1,
        combustion_inputs,
```

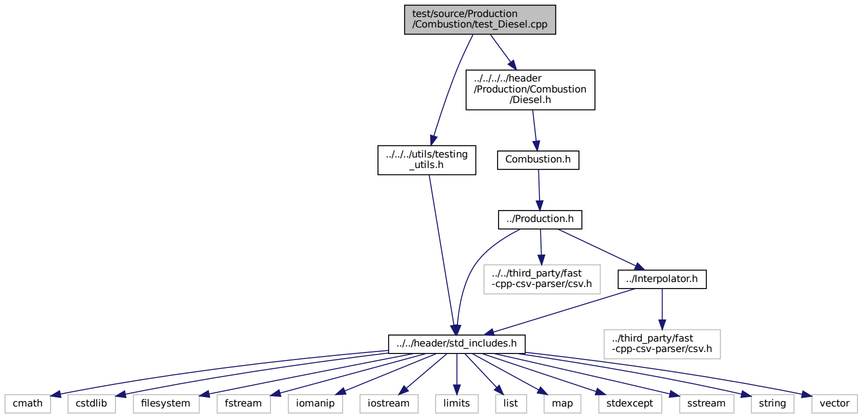
```
        time_vec_hrs_ptr
    );

    testTruth(
        not combustion_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->fuel_consumption_vec_L.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->fuel_cost_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->CO2_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->CO_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->NOx_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->SOx_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->CH4_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_combustion_ptr->PM_emissions_vec_kg.size(),
        8760,
        __FILE__,
        __LINE__
    );

    return test_combustion_ptr;
}   /* testConstruct_Combustion() */
```

5.57 test/source/Production/Combustion/test_Diesel.cpp File Reference

Testing suite for Diesel class.

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

Include dependency graph for test_Diesel.cpp:



Functions

- Combustion ∗ testConstruct_Diesel (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Diesel object and spot check some post-construction attributes.
- Combustion ∗ testConstructLookup_Diesel (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Diesel object using fuel consumption lookup.
- void testBadConstruct_Diesel (std::vector< double > ∗time_vec_hrs_ptr)

    Function to test the trying to construct a Diesel object given bad inputs is being handled as expected.
- void testCapacityConstraint_Diesel (Combustion ∗test_diesel_ptr)

    Test to check that the installed capacity constraint is active and behaving as expected.
- void testMinimumLoadRatioConstraint_Diesel (Combustion ∗test_diesel_ptr)

    Test to check that the minimum load ratio constraint is active and behaving as expected.
- void testCommit_Diesel (Combustion ∗test_diesel_ptr)

    Function to test if the commit method is working as expected, by checking some post-call attributes of the test Diesel object.
- void testMinimumRuntimeConstraint_Diesel (Combustion ∗test_diesel_ptr)

    Function to check that the minimum runtime constraint is active and behaving as expected.
- void testFuelConsumptionEmissions_Diesel (Combustion ∗test_diesel_ptr)

    Function to test that post-commit fuel consumption and emissions are > 0 when the test Diesel object is running, and = 0 when it is not (as expected).
- void testEconomics_Diesel (Combustion ∗test_diesel_ptr)

    Function to test that the post-commit model economics for the test Diesel object are as expected (> 0 when running, = 0 when not).
- void testFuelLookup_Diesel (Combustion ∗test_diesel_lookup_ptr)

    Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.
- int main (int argc, char ∗∗argv)

5.57.1 Detailed Description

Testing suite for Diesel class.

A suite of tests for the Diesel class.

5.57.2 Function Documentation

5.57.2.1 main()

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

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Combustion <-- Diesel");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }


    Combustion* test_diesel_ptr = testConstruct_Diesel(&time_vec_hrs);
    Combustion* test_diesel_lookup_ptr = testConstructLookup_Diesel(&time_vec_hrs);

    try {
        testBadConstruct_Diesel(&time_vec_hrs);

        testCapacityConstraint_Diesel(test_diesel_ptr);
        testMinimumLoadRatioConstraint_Diesel(test_diesel_ptr);

        testCommit_Diesel(test_diesel_ptr);

        testMinimumRuntimeConstraint_Diesel(test_diesel_ptr);

        testFuelConsumptionEmissions_Diesel(test_diesel_ptr);
        testEconomics_Diesel(test_diesel_ptr);

        testFuelLookup_Diesel(test_diesel_lookup_ptr);
    }


    catch (...) {
        delete test_diesel_ptr;
        delete test_diesel_lookup_ptr;

        printGold(" ..... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_diesel_ptr;
    delete test_diesel_lookup_ptr;

    printGold(" ..... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.57.2.2 testBadConstruct_Diesel()

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

Function to test the trying to construct a Diesel object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
        DieselInputs bad_diesel_inputs;
        bad_diesel_inputs.fuel_cost_L = -1;

        Diesel bad_diesel(
            8760,
            1,
            bad_diesel_inputs,
            time_vec_hrs_ptr
        );

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Diesel() */
```

5.57.2.3 testCapacityConstraint_Diesel()

```
void testCapacityConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Test to check that the installed capacity constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
{
    testFloatEquals(
        test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
        test_diesel_ptr->capacity_kW,
        __FILE__,
        __LINE__
    );

    return;
}   /* testCapacityConstraint_Diesel() */
```

5.57.2.4 testCommit_Diesel()

```
void testCommit_Diesel (
            Combustion * test_diesel_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Diesel object.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
{
    std::vector<double> dt_vec_hrs (48, 1);

    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    double load_kW = 0;
    double production_kW = 0;
    double roll = 0;

    for (int i = 0; i < 48; i++) {
        roll = (double)rand() / RAND_MAX;

        if (roll >= 0.95) {
            roll = 1.25;
        }

        load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
        load_kW = load_vec_kW[i];

        production_kW = test_diesel_ptr->requestProductionkW(
            i,
            dt_vec_hrs[i],
            load_kW
        );

        load_kW = test_diesel_ptr->commit(
            i,
            dt_vec_hrs[i],
            production_kW,
            load_kW
        );

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_diesel_ptr->production_vec_kW[i] -
            test_diesel_ptr->dispatch_vec_kW[i] -
            test_diesel_ptr->storage_vec_kW[i] -
            test_diesel_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );

        // capacity constraint
        if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
            testFloatEquals(
                test_diesel_ptr->production_vec_kW[i],
                test_diesel_ptr->capacity_kW,
                __FILE__,
                __LINE__
            );
        }

        // minimum load ratio constraint
        else if (
            test_diesel_ptr->is_running and
            test_diesel_ptr->production_vec_kW[i] > 0 and
            load_vec_kW[i] <
            ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
        ) {
            testFloatEquals(
                test_diesel_ptr->production_vec_kW[i],
                ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
                    test_diesel_ptr->capacity_kW,
                __FILE__,
                __LINE__
            );
        }
    }
```

```
    return;
} /* testCommit_Diesel() */
```

5.57.2.5 testConstruct_Diesel()

```
Combustion * testConstruct_Diesel (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Diesel object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Combustion pointer to a test Diesel object.

```
{
    DieselInputs diesel_inputs;

    Combustion* test_diesel_ptr =  new Diesel(
        8760,
        1,
        diesel_inputs,
        time_vec_hrs_ptr
    );

    testTruth(
        not diesel_inputs.combustion_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->type,
        CombustionType :: DIESEL,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_diesel_ptr->type_str == "DIESEL",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->linear_fuel_slope_LkWh,
        0.265675,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->linear_fuel_intercept_LkWh,
        0.026676,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->capital_cost,
        94125.375446,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->operation_maintenance_cost_kWh,
        0.069905,
```

```
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
        0.2,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
        4,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_diesel_ptr->replace_running_hrs,
        30000,
        __FILE__,
        __LINE__
    );

    return test_diesel_ptr;
}   /* testConstruct_Diesel() */
```

5.57.2.6 testConstructLookup_Diesel()

```
Combustion * testConstructLookup_Diesel (
             std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Diesel object using fuel consumption lookup.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Combustion pointer to a test Diesel object.

```
{
    DieselInputs diesel_inputs;

    diesel_inputs.combustion_inputs.fuel_mode = FuelMode :: FUEL_MODE_LOOKUP;
    diesel_inputs.combustion_inputs.path_2_fuel_interp_data =
        "data/test/interpolation/diesel_fuel_curve.csv";

    Combustion* test_diesel_lookup_ptr = new Diesel(
        8760,
        1,
        diesel_inputs,
        time_vec_hrs_ptr
    );

    return test_diesel_lookup_ptr;
}   /* testConstructLookup_Diesel() */
```

5.57.2.7 testEconomics_Diesel()

```
void testEconomics_Diesel (
             Combustion * test_diesel_ptr )
```

Function to test that the post-commit model economics for the test Diesel object are as expected ($>$ 0 when running, = 0 when not).

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
{
    std::vector<bool> expected_is_running_vec = {
        1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
    };

    bool is_running = false;

    for (int i = 0; i < 48; i++) {
        is_running = test_diesel_ptr->is_running_vec[i];

        testFloatEquals(
            is_running,
            expected_is_running_vec[i],
            __FILE__,
            __LINE__
        );

        // O&M, fuel consumption, and emissions > 0 whenever diesel is running
        if (is_running) {
            testGreaterThan(
                test_diesel_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
        else {
            testFloatEquals(
                test_diesel_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testEconomics_Diesel() */
```

5.57.2.8 testFuelConsumptionEmissions_Diesel()

```
void testFuelConsumptionEmissions_Diesel (
            Combustion * test_diesel_ptr )
```

Function to test that post-commit fuel consumption and emissions are $>$ 0 when the test Diesel object is running, and = 0 when it is not (as expected).

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
{
    std::vector<bool> expected_is_running_vec = {
        1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
    };

    bool is_running = false;

    for (int i = 0; i < 48; i++) {
        is_running = test_diesel_ptr->is_running_vec[i];
```

```

        testFloatEquals(
            is_running,
            expected_is_running_vec[i],
            __FILE__,
            __LINE__
        );

        // O&M, fuel consumption, and emissions > 0 whenever diesel is running
        if (is_running) {
            testGreaterThan(
                test_diesel_ptr->fuel_consumption_vec_L[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->fuel_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->CO2_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->CO_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->NOx_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->SOx_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->CH4_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testGreaterThan(
                test_diesel_ptr->PM_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
        else {
            testFloatEquals(
                test_diesel_ptr->fuel_consumption_vec_L[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->fuel_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->CO2_emissions_vec_kg[i],
                0,
                __FILE__,
```

```
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->CO_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->NOx_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->SOx_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->CH4_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );

            testFloatEquals(
                test_diesel_ptr->PM_emissions_vec_kg[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testFuelConsumptionEmissions_Diesel() */
```

5.57.2.9 testFuelLookup_Diesel()

```
void testFuelLookup_Diesel (
            Combustion * test_diesel_lookup_ptr )
```

Function to test that fuel consumption lookup (i.e., interpolation) is returning the expected values.

Parameters

test_diesel_lookup_ptr	A Combustion pointer to the test Diesel object using fuel consumption lookup.

```
{
    std::vector<double> load_ratio_vec = {
        0,
        0.170812859791767,
        0.322739274162545,
        0.369750203682042,
        0.443532869135929,
        0.471567864244626,
        0.536513734479662,
        0.586125806988674,
        0.601101175455075,
        0.658356862575221,
        0.70576929893201,
        0.784069734739331,
        0.805765927542453,
        0.884747873186048,
        0.930870496062112,
        0.979415217694769,
        1
    };

```

```
    std::vector<double> expected_fuel_consumption_vec_L = {
        4.68079520372916,
        8.35159603357656,
        11.7422361561399,
        12.9931187917615,
        14.8786636301325,
        15.5746957307243,
        17.1419229487141,
        18.3041866133728,
        18.6530540913696,
        19.9569217633299,
        21.012354614584,
        22.7142305879957,
        23.1916726441968,
        24.8602332554707,
        25.8172124624032,
        26.8256741279932,
        27.254952
    };

    for (size_t i = 0; i < load_ratio_vec.size(); i++) {
        testFloatEquals(
            test_diesel_lookup_ptr->getFuelConsumptionL(
                1, load_ratio_vec[i] * test_diesel_lookup_ptr->capacity_kW
            ),
            expected_fuel_consumption_vec_L[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testFuelLookup_Diesel() */
```

5.57.2.10 testMinimumLoadRatioConstraint_Diesel()

```
void testMinimumLoadRatioConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Test to check that the minimum load ratio constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

```
{
    testFloatEquals(
        test_diesel_ptr->requestProductionkW(
            0,
            1,
            0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
                test_diesel_ptr->capacity_kW
        ),
        ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
        __FILE__,
        __LINE__
    );

    return;
}   /* testMinimumLoadRatioConstraint_Diesel() */
```

5.57.2.11 testMinimumRuntimeConstraint_Diesel()

```
void testMinimumRuntimeConstraint_Diesel (
            Combustion * test_diesel_ptr )
```

Function to check that the minimum runtime constraint is active and behaving as expected.

Parameters

test_diesel_ptr	A Combustion pointer to the test Diesel object.

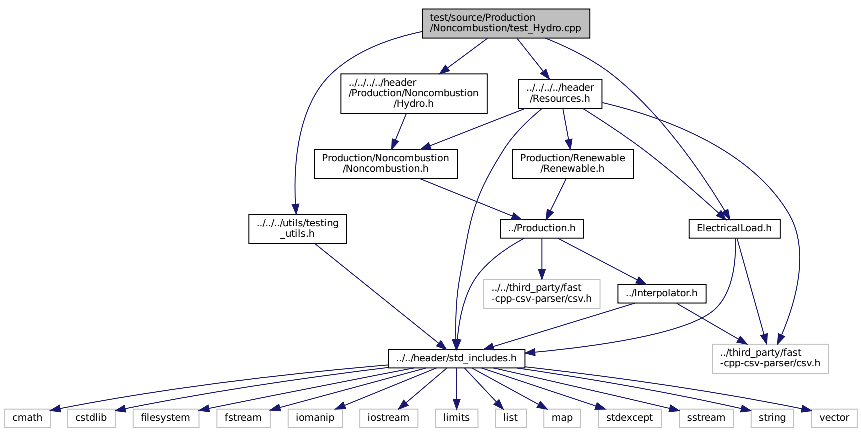
```
{
    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    std::vector<bool> expected_is_running_vec = {
        1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
    };

    for (int i = 0; i < 48; i++) {
        testFloatEquals(
            test_diesel_ptr->is_running_vec[i],
            expected_is_running_vec[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testMinimumRuntimeConstraint_Diesel() */
```

5.58 test/source/Production/Noncombustion/test_Hydro.cpp File Reference

Testing suite for Hydro class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Resources.h"
#include "../../../../header/ElectricalLoad.h"
#include "../../../../header/Production/Noncombustion/Hydro.h"
```

Include dependency graph for test_Hydro.cpp:



Functions

- Noncombustion ∗ testConstruct_Hydro (HydroInputs hydro_inputs, std::vector< double > ∗time_vec_hrs_↩ ptr)

A function to construct a Hydro object and spot check some post-construction attributes.

- void testEfficiencyInterpolation_Hydro (Noncombustion ∗test_hydro_ptr)

  Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.
- void testCommit_Hydro (Noncombustion ∗test_hydro_ptr, Resources ∗test_resources_ptr)
- int main (int argc, char ∗∗argv)

5.58.1 Detailed Description

Testing suite for Hydro class.

A suite of tests for the Hydro class.

5.58.2 Function Documentation

5.58.2.1 main()

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

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Noncombustion <-- Hydro");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

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

    ElectricalLoad* test_electrical_load_ptr =
        new ElectricalLoad(path_2_electrical_load_time_series);

    Resources* test_resources_ptr = new Resources();

    HydroInputs hydro_inputs;
    int hydro_resource_key = 0;

    hydro_inputs.reservoir_capacity_m3 = 10000;
    hydro_inputs.resource_key = hydro_resource_key;

    Noncombustion* test_hydro_ptr = testConstruct_Hydro(hydro_inputs, &time_vec_hrs);

    std::string path_2_hydro_resource_data =
        "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";

    test_resources_ptr->addResource(
        NoncombustionType::HYDRO,
        path_2_hydro_resource_data,
        hydro_resource_key,
        test_electrical_load_ptr
    );


    try {
        testEfficiencyInterpolation_Hydro(test_hydro_ptr);
        testCommit_Hydro(test_hydro_ptr, test_resources_ptr);
    }
```

```


    catch (...) {
        delete test_electrical_load_ptr;
        delete test_resources_ptr;
        delete test_hydro_ptr;

        printGold(" ... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_electrical_load_ptr;
    delete test_resources_ptr;
    delete test_hydro_ptr;

    printGold(" ... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.58.2.2 testCommit_Hydro()

```
void testCommit_Hydro (
            Noncombustion * test_hydro_ptr,
            Resources * test_resources_ptr )
{
    double load_kW = 100 * (double)rand() / RAND_MAX;
    double production_kW = 0;

    for (int i = 0; i < 8760; i++) {
        production_kW = test_hydro_ptr->requestProductionkW(
            i,
            1,
            load_kW,
            test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
        );

        load_kW = test_hydro_ptr->commit(
            i,
            1,
            production_kW,
            load_kW,
            test_resources_ptr->resource_map_1D[test_hydro_ptr->resource_key][i]
        );

        testGreaterThanOrEqualTo(
            test_hydro_ptr->production_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );

        testLessThanOrEqualTo(
            test_hydro_ptr->production_vec_kW[i],
            test_hydro_ptr->capacity_kW,
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_hydro_ptr->production_vec_kW[i] -
            test_hydro_ptr->dispatch_vec_kW[i] -
            test_hydro_ptr->curtailment_vec_kW[i] -
            test_hydro_ptr->storage_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );

        testGreaterThanOrEqualTo(
            ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
            0,
            __FILE__,
            __LINE__
```

```
        );

        testLessThanOrEqualTo(
            ((Hydro*)test_hydro_ptr)->turbine_flow_vec_m3hr[i],
            ((Hydro*)test_hydro_ptr)->maximum_flow_m3hr,
            __FILE__,
            __LINE__
        );

        testGreaterThanOrEqualTo(
            ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
            0,
            __FILE__,
            __LINE__
        );

        testLessThanOrEqualTo(
            ((Hydro*)test_hydro_ptr)->stored_volume_vec_m3[i],
            ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testCommit_Hydro() */
```

5.58.2.3 testConstruct_Hydro()

```
Hydro *Noncombustion * testConstruct_Hydro (
            HydroInputs hydro_inputs,
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Hydro object and spot check some post-construction attributes.

Returns

A Noncombustion pointer to a test Hydro object.

```
{
    Noncombustion* test_hydro_ptr =  new Hydro(
        8760,
        1,
        hydro_inputs,
        time_vec_hrs_ptr
    );

    testTruth(
        not hydro_inputs.noncombustion_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_hydro_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_hydro_ptr->type,
        NoncombustionType :: HYDRO,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_hydro_ptr->type_str == "HYDRO",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((Hydro*)test_hydro_ptr)->turbine_type,
```

```
        HydroTurbineType :: HYDRO_TURBINE_PELTON,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((Hydro*)test_hydro_ptr)->reservoir_capacity_m3,
        10000,
        __FILE__,
        __LINE__
    );

    return test_hydro_ptr;
}   /* testConstruct_Hydro() */
```

5.58.2.4 testEfficiencyInterpolation_Hydro()

```
void testEfficiencyInterpolation_Hydro (
            Noncombustion * test_hydro_ptr )
```

Function to test that the generator and turbine efficiency maps are being initialized as expected, and that efficiency interpolation is returning the expected values.

Parameters

test_hydro_ptr	A Noncombustion pointer to the test Hydro object.

```
{
    std::vector<double> expected_gen_power_ratios = {
        0,   0.1, 0.2,  0.3, 0.4, 0.5,
        0.6, 0.7, 0.75, 0.8, 0.9, 1
    };

    std::vector<double> expected_gen_efficiencies = {
        0.000, 0.800, 0.900, 0.913,
        0.925, 0.943, 0.947, 0.950,
        0.953, 0.954, 0.956, 0.958
    };

    double query = 0;
    for (size_t i = 0; i < expected_gen_power_ratios.size(); i++) {
        testFloatEquals(
            test_hydro_ptr->interpolator.interp_map_1D[
                HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
            ].x_vec[i],
            expected_gen_power_ratios[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_hydro_ptr->interpolator.interp_map_1D[
                HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY
            ].y_vec[i],
            expected_gen_efficiencies[i],
            __FILE__,
            __LINE__
        );

        if (i < expected_gen_power_ratios.size() - 1) {
            query = expected_gen_power_ratios[i] + ((double)rand() / RAND_MAX) *
                (expected_gen_power_ratios[i + 1] - expected_gen_power_ratios[i]);

            test_hydro_ptr->interpolator.interp1D(
                HydroInterpKeys :: GENERATOR_EFFICIENCY_INTERP_KEY,
                query
            );
        }
    }

    std::vector<double> expected_turb_power_ratios = {
        0,   0.1, 0.2, 0.3, 0.4,
        0.5, 0.6, 0.7, 0.8, 0.9,
        1
```

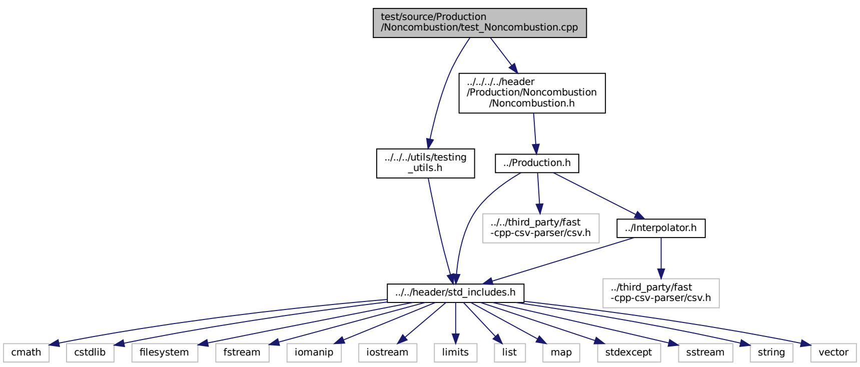
```
    };

    std::vector<double> expected_turb_efficiencies = {
        0.000, 0.780, 0.855, 0.875, 0.890,
        0.900, 0.908, 0.913, 0.918, 0.908,
        0.880
    };

    for (size_t i = 0; i < expected_turb_power_ratios.size(); i++) {
        testFloatEquals(
            test_hydro_ptr->interpolator.interp_map_1D[
                HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
            ].x_vec[i],
            expected_turb_power_ratios[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_hydro_ptr->interpolator.interp_map_1D[
                HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY
            ].y_vec[i],
            expected_turb_efficiencies[i],
            __FILE__,
            __LINE__
        );

        if (i < expected_turb_power_ratios.size() - 1) {
            query = expected_turb_power_ratios[i] + ((double)rand() / RAND_MAX) *
                (expected_turb_power_ratios[i + 1] - expected_turb_power_ratios[i]);

            test_hydro_ptr->interpolator.interp1D(
                HydroInterpKeys :: TURBINE_EFFICIENCY_INTERP_KEY,
                query
            );
        }
    }

    return;
}   /* testEfficiencyInterpolation_Hydro() */
```

5.59 test/source/Production/Noncombustion/test_Noncombustion.cpp File Reference

Testing suite for Noncombustion class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Noncombustion/Noncombustion.h"
```

Include dependency graph for test_Noncombustion.cpp:



Functions

- Noncombustion ∗ testConstruct_Noncombustion (std::vector< double > ∗time_vec_hrs_ptr)

  A function to construct a Noncombustion object and spot check some post-construction attributes.
- int main (int argc, char ∗∗argv)

5.59.1 Detailed Description

Testing suite for Noncombustion class.

A suite of tests for the Noncombustion class.

5.59.2 Function Documentation

5.59.2.1 main()

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

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Noncombustion");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Noncombustion* test_noncombustion_ptr = testConstruct_Noncombustion(&time_vec_hrs);


    try {
        //...
    }


    catch (...) {
        delete test_noncombustion_ptr;

        printGold(" ............. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_noncombustion_ptr;

    printGold(" ............. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.59.2.2 testConstruct_Noncombustion()

```
Noncombustion * testConstruct_Noncombustion (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Noncombustion object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Noncombustion object.

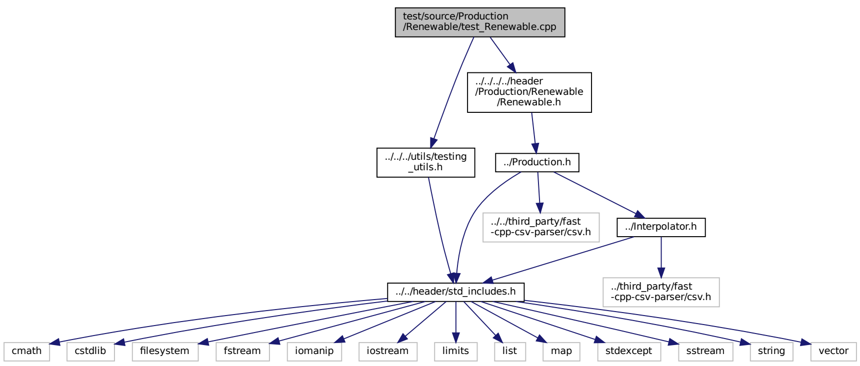
```
{
    NoncombustionInputs noncombustion_inputs;

    Noncombustion* test_noncombustion_ptr =
        new Noncombustion(
            8760,
            1,
            noncombustion_inputs,
            time_vec_hrs_ptr
        );

    testTruth(
        not noncombustion_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_noncombustion_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    return test_noncombustion_ptr;
}   /* testConstruct_Noncombustion() */
```

5.60 test/source/Production/Renewable/test_Renewable.cpp File Reference

Testing suite for Renewable class.

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

Include dependency graph for test_Renewable.cpp:



Functions

- Renewable * testConstruct_Renewable (std::vector< double > *time_vec_hrs_ptr)

    A function to construct a Renewable object and spot check some post-construction attributes.
- int main (int argc, char **argv)

5.60.1 Detailed Description

Testing suite for Renewable class.

A suite of tests for the Renewable class.

5.60.2 Function Documentation

5.60.2.1 main()

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

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_renewable_ptr = testConstruct_Renewable(&time_vec_hrs);


    try {
        //...
    }


    catch (...) {
        delete test_renewable_ptr;

        printGold(" ................. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_renewable_ptr;

    printGold(" ................. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.60.2.2 testConstruct_Renewable()

```
Renewable * testConstruct_Renewable (
        std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Renewable object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Renewable object.

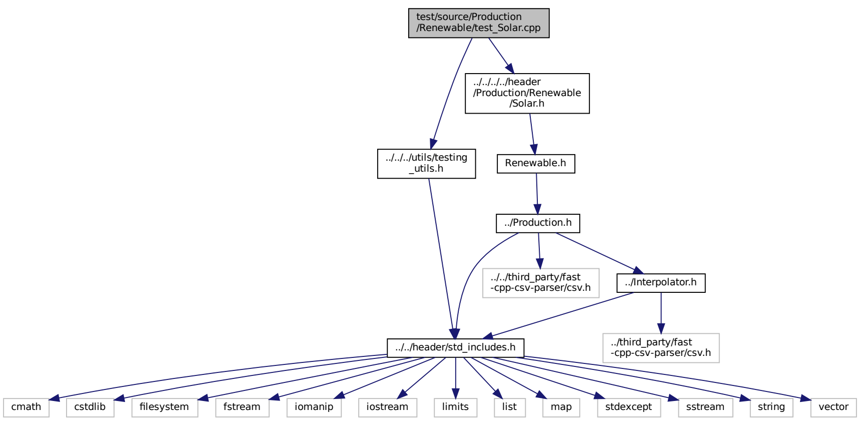
```
{
    RenewableInputs renewable_inputs;

    Renewable* test_renewable_ptr = new Renewable(
        8760,
        1,
        renewable_inputs,
        time_vec_hrs_ptr
    );

    testTruth(
        not renewable_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_renewable_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    return test_renewable_ptr;
}   /* testConstruct_Renewable() */
```

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

Testing suite for Solar class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Solar.h"
```

Include dependency graph for test_Solar.cpp:



Functions

- Renewable ∗ testConstruct_Solar (std::vector< double > ∗time_vec_hrs_ptr)

  A function to construct a Solar object and spot check some post-construction attributes.
- void testBadConstruct_Solar (std::vector< double > ∗time_vec_hrs_ptr)

  Function to test the trying to construct a Solar object given bad inputs is being handled as expected.
- void testProductionOverride_Solar (std::string path_2_normalized_production_time_series, std::vector< double > ∗time_vec_hrs_ptr)

  Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.
- void testProductionConstraint_Solar (Renewable ∗test_solar_ptr)

  Function to test that the production constraint is active and behaving as expected.
- void testCommit_Solar (Renewable ∗test_solar_ptr)

  Function to test if the commit method is working as expected, by checking some post-call attributes of the test Solar object. Uses a randomized resource input.
- void testEconomics_Solar (Renewable ∗test_solar_ptr)
- int main (int argc, char ∗∗argv)

5.61.1 Detailed Description

Testing suite for Solar class.

A suite of tests for the Solar class.

5.61.2 Function Documentation

5.61.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Solar");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_solar_ptr = testConstruct_Solar(&time_vec_hrs);


    try {
        testBadConstruct_Solar(&time_vec_hrs);

        std::string path_2_normalized_production_time_series =
            "data/test/normalized_production/normalized_solar_production.csv";

        testProductionOverride_Solar(
            path_2_normalized_production_time_series,
            &time_vec_hrs
        );
```

```
        testProductionConstraint_Solar(test_solar_ptr);

        testCommit_Solar(test_solar_ptr);
        testEconomics_Solar(test_solar_ptr);
    }


    catch (...) {
        delete test_solar_ptr;

        printGold(" ....... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_solar_ptr;

    printGold(" ....... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.61.2.2 testBadConstruct_Solar()

```
void testBadConstruct_Solar (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Solar object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
        SolarInputs bad_solar_inputs;
        bad_solar_inputs.derating = -1;

        Solar bad_solar(8760, 1, bad_solar_inputs, time_vec_hrs_ptr);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Solar() */
```

5.61.2.3 testCommit_Solar()

```
void testCommit_Solar (
            Renewable * test_solar_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Solar object. Uses a randomized resource input.

Parameters

test_solar_ptr	A Renewable pointer to the test Solar object.

```
{
    std::vector<double> dt_vec_hrs (48, 1);

    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    double load_kW = 0;
    double production_kW = 0;
    double roll = 0;
    double solar_resource_kWm2 = 0;

    for (int i = 0; i < 48; i++) {
        roll = (double)rand() / RAND_MAX;

        solar_resource_kWm2 = roll;

        roll = (double)rand() / RAND_MAX;

        if (roll <= 0.1) {
            solar_resource_kWm2 = 0;
        }

        else if (roll >= 0.95) {
            solar_resource_kWm2 = 1.25;
        }

        roll = (double)rand() / RAND_MAX;

        if (roll >= 0.95) {
            roll = 1.25;
        }

        load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
        load_kW = load_vec_kW[i];

        production_kW = test_solar_ptr->computeProductionkW(
            i,
            dt_vec_hrs[i],
            solar_resource_kWm2
        );

        load_kW = test_solar_ptr->commit(
            i,
            dt_vec_hrs[i],
            production_kW,
            load_kW
        );

        // is running (or not) as expected
        if (solar_resource_kWm2 > 0) {
            testTruth(
                test_solar_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        else {
            testTruth(
                not test_solar_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_solar_ptr->production_vec_kW[i] -
            test_solar_ptr->dispatch_vec_kW[i] -
            test_solar_ptr->storage_vec_kW[i] -
```

```
            test_solar_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );

        // capacity constraint
        if (solar_resource_kWm2 > 1) {
            testFloatEquals(
                test_solar_ptr->production_vec_kW[i],
                test_solar_ptr->capacity_kW,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testCommit_Solar() */
```

5.61.2.4 testConstruct_Solar()

```
Renewable * testConstruct_Solar (
        std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Solar object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Solar object.

```
{
    SolarInputs solar_inputs;

    Renewable* test_solar_ptr = new Solar(
        8760,
        1,
        solar_inputs,
        time_vec_hrs_ptr
    );

    testTruth(
        not solar_inputs.renewable_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_solar_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_solar_ptr->type,
        RenewableType :: SOLAR,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_solar_ptr->type_str == "SOLAR",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
```

```
        test_solar_ptr->capital_cost,
        350118.723363,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_solar_ptr->operation_maintenance_cost_kWh,
        0.01,
        __FILE__,
        __LINE__
    );

    return test_solar_ptr;
}   /* testConstruct_Solar() */
```

5.61.2.5 testEconomics_Solar()

```
void testEconomics_Solar (
            Renewable * test_solar_ptr )
```

```
{
    for (int i = 0; i < 48; i++) {
        // resource, O&M > 0 whenever solar is running (i.e., producing)
        if (test_solar_ptr->is_running_vec[i]) {
            testGreaterThan(
                test_solar_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M = 0 whenever solar is not running (i.e., not producing)
        else {
            testFloatEquals(
                test_solar_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testEconomics_Solar() */
```

5.61.2.6 testProductionConstraint_Solar()

```
void testProductionConstraint_Solar (
            Renewable * test_solar_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_solar_ptr	A Renewable pointer to the test Solar object.

```
{
    testFloatEquals(
        test_solar_ptr->computeProductionkW(0, 1, 2),
        100,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
```

```
        test_solar_ptr->computeProductionkW(0, 1, -1),
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testProductionConstraint_Solar() */
```

5.61.2.7 testProductionOverride_Solar()

```
void testProductionOverride_Solar (
            std::string path_2_normalized_production_time_series,
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test that normalized production data is being read in correctly, and that the associated production override feature is behaving as expected.

Parameters

path_2_normalized_production_time_series	A path (either relative or absolute) to the given normalized production time series data.
time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    SolarInputs solar_inputs;

    solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
        path_2_normalized_production_time_series;

    Solar test_solar_override(
        time_vec_hrs_ptr->size(),
        1,
        solar_inputs,
        time_vec_hrs_ptr
    );


    std::vector<double> expected_normalized_production_vec = {
        0.916955708517556,
        0.90947506148393,
        0.38425267564517,
        0.191510884037643,
        0.803361391862077,
        0.261511294927198,
        0.221944653883198,
        0.858495335855501,
        0.0162863861443092,
        0.774345409915512,
        0.354898664149867,
        0.11158009453439,
        0.191670176408956,
        0.0149072402795702,
        0.30174228469322,
        0.0815062957850151,
        0.776404660266821,
        0.207069187162109,
        0.518926216750454,
        0.148538109788597,
        0.443035200791027,
        0.62119079547209,
        0.270792717524391,
        0.761074879460849,
        0.0545251308358993,
        0.0895417089500092,
        0.21787190761933,
        0.834403724509682,
        0.908807953036246,
        0.815888965292123,
        0.416663215314571,
        0.523649705576525,
        0.490890480401437,
```

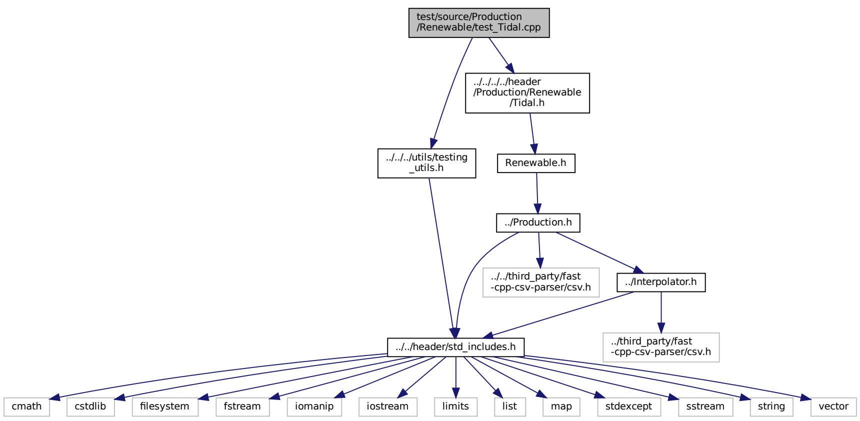
```
        0.28317138282312,
        0.877382682055847,
        0.14972090597986,
        0.480161632646382,
        0.0655830129932816,
        0.41802666403448,
        0.48692477737368,
        0.275957323208066,
        0.228651250718341,
        0.574371311550247,
        0.251872481275769,
        0.802697508767121,
        0.00130607304363551,
        0.481240172488057,
        0.702527508293784
    };

    for (size_t i = 0; i < expected_normalized_production_vec.size(); i++) {
        testFloatEquals(
            test_solar_override.normalized_production_vec[i],
            expected_normalized_production_vec[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_solar_override.computeProductionkW(i, rand(), rand()),
            test_solar_override.capacity_kW * expected_normalized_production_vec[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testProductionOverride_Solar() */
```

5.62 test/source/Production/Renewable/test_Tidal.cpp File Reference

Testing suite for Tidal class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Tidal.h"
```

Include dependency graph for test_Tidal.cpp:



Functions

- Renewable ∗ testConstruct_Tidal (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Tidal object and spot check some post-construction attributes.

- void testBadConstruct_Tidal (std::vector< double > ∗time_vec_hrs_ptr)

    Function to test the trying to construct a Tidal object given bad inputs is being handled as expected.
- void testProductionConstraint_Tidal (Renewable ∗test_tidal_ptr)

    Function to test that the production constraint is active and behaving as expected.
- void testCommit_Tidal (Renewable ∗test_tidal_ptr)

    Function to test if the commit method is working as expected, by checking some post-call attributes of the test Tidal object. Uses a randomized resource input.
- void testEconomics_Tidal (Renewable ∗test_tidal_ptr)
- int main (int argc, char ∗∗argv)

5.62.1 Detailed Description

Testing suite for Tidal class.

A suite of tests for the Tidal class.

5.62.2 Function Documentation

5.62.2.1 main()

```
int main (
        int argc,
        char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Tidal");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_tidal_ptr = testConstruct_Tidal(&time_vec_hrs);


    try {
        testBadConstruct_Tidal(&time_vec_hrs);

        testProductionConstraint_Tidal(test_tidal_ptr);

        testCommit_Tidal(test_tidal_ptr);
        testEconomics_Tidal(test_tidal_ptr);
    }


    catch (...) {
        delete test_tidal_ptr;

        printGold(" ....... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_tidal_ptr;

    printGold(" ....... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.62.2.2 testBadConstruct_Tidal()

```
void testBadConstruct_Tidal (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Tidal object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
        TidalInputs bad_tidal_inputs;
        bad_tidal_inputs.design_speed_ms = -1;

        Tidal bad_tidal(8760, 1, bad_tidal_inputs, time_vec_hrs_ptr);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}    /* testBadConstruct_Tidal() */
```

5.62.2.3 testCommit_Tidal()

```
void testCommit_Tidal (
            Renewable * test_tidal_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Tidal object. Uses a randomized resource input.

Parameters

test_tidal_ptr	A Renewable pointer to the test Tidal object.

```
{
    std::vector<double> dt_vec_hrs (48, 1);

    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    double load_kW = 0;
    double production_kW = 0;
    double roll = 0;
    double tidal_resource_ms = 0;

    for (int i = 0; i < 48; i++) {
        roll = (double)rand() / RAND_MAX;

        tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;

        roll = (double)rand() / RAND_MAX;

        if (roll <= 0.1) {
            tidal_resource_ms = 0;
        }
```

```
        else if (roll >= 0.95) {
            tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
        }

        roll = (double)rand() / RAND_MAX;

        if (roll >= 0.95) {
            roll = 1.25;
        }

        load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
        load_kW = load_vec_kW[i];

        production_kW = test_tidal_ptr->computeProductionkW(
            i,
            dt_vec_hrs[i],
            tidal_resource_ms
        );

        load_kW = test_tidal_ptr->commit(
            i,
            dt_vec_hrs[i],
            production_kW,
            load_kW
        );

        // is running (or not) as expected
        if (production_kW > 0) {
            testTruth(
                test_tidal_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        else {
            testTruth(
                not test_tidal_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_tidal_ptr->production_vec_kW[i] -
            test_tidal_ptr->dispatch_vec_kW[i] -
            test_tidal_ptr->storage_vec_kW[i] -
            test_tidal_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testCommit_Tidal() */
```

5.62.2.4 testConstruct_Tidal()

```
Renewable * testConstruct_Tidal (
        std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Tidal object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Tidal object.

```
{
    TidalInputs tidal_inputs;

    Renewable* test_tidal_ptr = new Tidal(8760, 1, tidal_inputs, time_vec_hrs_ptr);

    testTruth(
        not tidal_inputs.renewable_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->type,
        RenewableType :: TIDAL,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_tidal_ptr->type_str == "TIDAL",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->capital_cost,
        500237.446725,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->operation_maintenance_cost_kWh,
        0.069905,
        __FILE__,
        __LINE__
    );

    return test_tidal_ptr;
}   /* testConstruct_Tidal() */
```

5.62.2.5 testEconomics_Tidal()

```
void testEconomics_Tidal (
            Renewable * test_tidal_ptr )
```

```
{
    for (int i = 0; i < 48; i++) {
        // resource, O&M > 0 whenever tidal is running (i.e., producing)
        if (test_tidal_ptr->is_running_vec[i]) {
            testGreaterThan(
                test_tidal_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M = 0 whenever tidal is not running (i.e., not producing)
        else {
            testFloatEquals(
                test_tidal_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

```

```
    return;
}   /* testEconomics_Tidal() */
```

5.62.2.6 testProductionConstraint_Tidal()

```
void testProductionConstraint_Tidal (
            Renewable * test_tidal_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_tidal_ptr	A Renewable pointer to the test Tidal object.

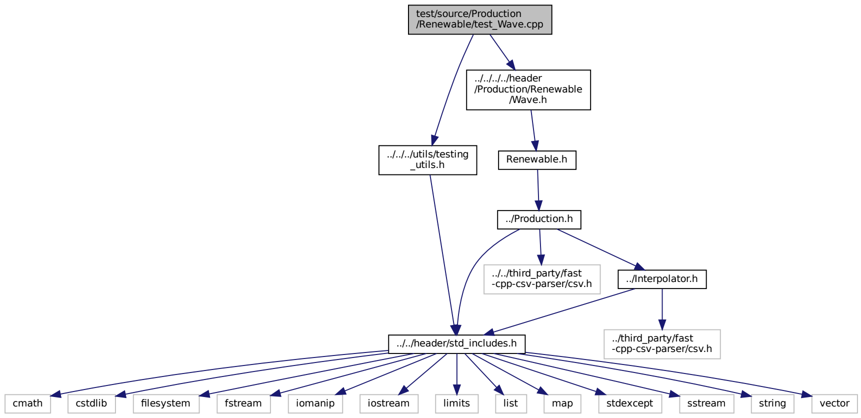
```
{
    testFloatEquals(
        test_tidal_ptr->computeProductionkW(0, 1, 1e6),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->computeProductionkW(
            0,
            1,
            ((Tidal*)test_tidal_ptr)->design_speed_ms
        ),
        test_tidal_ptr->capacity_kW,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_tidal_ptr->computeProductionkW(0, 1, -1),
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testProductionConstraint_Tidal() */
```

5.63 test/source/Production/Renewable/test_Wave.cpp File Reference

Testing suite for Wave class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Wave.h"
```

Include dependency graph for test_Wave.cpp:



Functions

- Renewable ∗ testConstruct_Wave (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Wave object and spot check some post-construction attributes.
- Renewable ∗ testConstructLookup_Wave (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Wave object using production lookup.
- void testBadConstruct_Wave (std::vector< double > ∗time_vec_hrs_ptr)

    Function to test the trying to construct a Wave object given bad inputs is being handled as expected.
- void testProductionConstraint_Wave (Renewable ∗test_wave_ptr)

    Function to test that the production constraint is active and behaving as expected.
- void testCommit_Wave (Renewable ∗test_wave_ptr)

    Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wave object. Uses a randomized resource input.
- void testEconomics_Wave (Renewable ∗test_wave_ptr)
- void testProductionLookup_Wave (Renewable ∗test_wave_lookup_ptr)

    Function to test that production lookup (i.e., interpolation) is returning the expected values.
- int main (int argc, char ∗∗argv)

5.63.1 Detailed Description

Testing suite for Wave class.

A suite of tests for the Wave class.

5.63.2 Function Documentation

5.63.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Wave");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_wave_ptr = testConstruct_Wave(&time_vec_hrs);
    Renewable* test_wave_lookup_ptr = testConstructLookup_Wave(&time_vec_hrs);


    try {
        testBadConstruct_Wave(&time_vec_hrs);

        testProductionConstraint_Wave(test_wave_ptr);

        testCommit_Wave(test_wave_ptr);
        testEconomics_Wave(test_wave_ptr);

        testProductionLookup_Wave(test_wave_lookup_ptr);
    }


    catch (...) {
        delete test_wave_ptr;
        delete test_wave_lookup_ptr;

        printGold(" ........ ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_wave_ptr;
    delete test_wave_lookup_ptr;

    printGold(" ........ ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.63.2.2 testBadConstruct_Wave()

```
void testBadConstruct_Wave (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Wave object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
```

```
        WaveInputs bad_wave_inputs;
        bad_wave_inputs.design_significant_wave_height_m = -1;

        Wave bad_wave(8760, 1, bad_wave_inputs, time_vec_hrs_ptr);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Wave() */
```

5.63.2.3 testCommit_Wave()

```
void testCommit_Wave (
            Renewable * test_wave_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wave object. Uses a randomized resource input.

Parameters

test_wave_ptr	A Renewable pointer to the test Wave object.

```
{
    std::vector<double> dt_vec_hrs (48, 1);

    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    double load_kW = 0;
    double production_kW = 0;
    double roll = 0;
    double significant_wave_height_m = 0;
    double energy_period_s = 0;

    for (int i = 0; i < 48; i++) {
        roll = (double)rand() / RAND_MAX;

        if (roll <= 0.05) {
            roll = 0;
        }

        significant_wave_height_m = roll *
            ((Wave*)test_wave_ptr)->design_significant_wave_height_m;

        roll = (double)rand() / RAND_MAX;

        if (roll <= 0.05) {
            roll = 0;
        }

        energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;

        roll = (double)rand() / RAND_MAX;

        if (roll >= 0.95) {
            roll = 1.25;
        }

        load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
        load_kW = load_vec_kW[i];

        production_kW = test_wave_ptr->computeProductionkW(
            i,
            dt_vec_hrs[i],
```

```
            significant_wave_height_m,
            energy_period_s
        );

        load_kW = test_wave_ptr->commit(
            i,
            dt_vec_hrs[i],
            production_kW,
            load_kW
        );

        // is running (or not) as expected
        if (production_kW > 0) {
            testTruth(
                test_wave_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        else {
            testTruth(
                not test_wave_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_wave_ptr->production_vec_kW[i] -
            test_wave_ptr->dispatch_vec_kW[i] -
            test_wave_ptr->storage_vec_kW[i] -
            test_wave_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testCommit_Wave() */
```

5.63.2.4 testConstruct_Wave()

Renewable * testConstruct_Wave (
    std::vector< double > * *time_vec_hrs_ptr*)

A function to construct a Wave object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wave object.

```
{
    WaveInputs wave_inputs;

    Renewable* test_wave_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);

    testTruth(
```

```
        not wave_inputs.renewable_inputs.production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wave_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wave_ptr->type,
        RenewableType :: WAVE,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_wave_ptr->type_str == "WAVE",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wave_ptr->capital_cost,
        850831.063539,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wave_ptr->operation_maintenance_cost_kWh,
        0.069905,
        __FILE__,
        __LINE__
    );

    return test_wave_ptr;
}   /* testConstruct_Wave() */
```

5.63.2.5 testConstructLookup_Wave()

```
Renewable * testConstructLookup_Wave (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Wave object using production lookup.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wave object.

```
{
    WaveInputs wave_inputs;

    wave_inputs.power_model = WavePowerProductionModel :: WAVE_POWER_LOOKUP;
    wave_inputs.path_2_normalized_performance_matrix =
        "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";

    Renewable* test_wave_lookup_ptr = new Wave(8760, 1, wave_inputs, time_vec_hrs_ptr);

    return test_wave_lookup_ptr;
}   /* testConstructLookup_Wave() */
```

5.63.2.6 testEconomics_Wave()

```
void testEconomics_Wave (
            Renewable * test_wave_ptr )
```

```
{
    for (int i = 0; i < 48; i++) {
        // resource, O&M > 0 whenever wave is running (i.e., producing)
        if (test_wave_ptr->is_running_vec[i]) {
            testGreaterThan(
                test_wave_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M = 0 whenever wave is not running (i.e., not producing)
        else {
            testFloatEquals(
                test_wave_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testEconomics_Wave() */
```

5.63.2.7 testProductionConstraint_Wave()

```
void testProductionConstraint_Wave (
            Renewable * test_wave_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_wave_ptr	A Renewable pointer to the test Wave object.

```
{
    testFloatEquals(
        test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testProductionConstraint_Wave() */
```

5.63.2.8 testProductionLookup_Wave()

```
void testProductionLookup_Wave (
            Renewable * test_wave_lookup_ptr )
```

Function to test that production lookup (i.e., interpolation) is returning the expected values.

Parameters

test_wave_lookup_ptr	A Renewable pointer to the test Wave object using production lookup.

```
{
    std::vector<double> significant_wave_height_vec_m = {
        0.389211848822208,
        0.836477431896843,
        1.52738334015579,
        1.92640601114508,
        2.27297317532019,
        2.87416589636605,
        3.72275770908175,
        3.95063175885536,
        4.68097139867404,
        4.97775020449812,
        5.55184219980547,
        6.06566629451658,
        6.27927876785062,
        6.96218133671013,
        7.51754442460228
    };

    std::vector<double> energy_period_vec_s = {
        5.45741899698926,
        6.00101329139007,
        7.50567689404182,
        8.77681262912881,
        9.45143678206774,
        10.7767876462885,
        11.4795760857165,
        12.9430684577599,
        13.303544885703,
        14.5069863517863,
        15.1487890438045,
        16.086524049077,
        17.176609978648,
        18.4155153740256,
        19.1704554940162
    };

    std::vector<std::vector<double» expected_normalized_performance_matrix = {

   {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.8996148

   {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.8820580

   {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608

   {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240

   {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061

   {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811

   {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511

   {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657

   {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855

   {0,0.0157252942367668,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442

   {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265

   {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010

   {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272

   {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51

   {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
    };

    for (size_t i = 0; i < energy_period_vec_s.size(); i++) {
        for (size_t j = 0; j < significant_wave_height_vec_m.size(); j++) {
            testFloatEquals(
                test_wave_lookup_ptr->computeProductionkW(
                    0,
                    1,
                    significant_wave_height_vec_m[j],
                    energy_period_vec_s[i]
                ),
                expected_normalized_performance_matrix[i][j] *
                test_wave_lookup_ptr->capacity_kW,
                __FILE__,
```

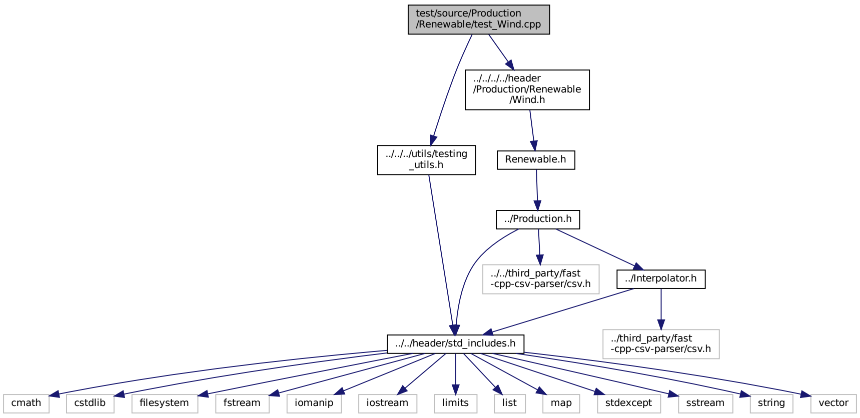
```
                __LINE__
            );
        }
    }

    return;
}   /* testProductionLookup_Wave() */
```

5.64 test/source/Production/Renewable/test_Wind.cpp File Reference

Testing suite for Wind class.

```
#include "../../../utils/testing_utils.h"
#include "../../../../header/Production/Renewable/Wind.h"
```

Include dependency graph for test_Wind.cpp:



Functions

- Renewable ∗ testConstruct_Wind (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Wind object and spot check some post-construction attributes.
- void testBadConstruct_Wind (std::vector< double > ∗time_vec_hrs_ptr)

    Function to test the trying to construct a Wind object given bad inputs is being handled as expected.
- void testProductionConstraint_Wind (Renewable ∗test_wind_ptr)

    Function to test that the production constraint is active and behaving as expected.
- void testCommit_Wind (Renewable ∗test_wind_ptr)

    Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wind object. Uses a randomized resource input.
- void testEconomics_Wind (Renewable ∗test_wind_ptr)
- int main (int argc, char ∗∗argv)

5.64.1 Detailed Description

Testing suite for Wind class.

A suite of tests for the Wind class.

5.64.2 Function Documentation

5.64.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Wind");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Renewable* test_wind_ptr = testConstruct_Wind(&time_vec_hrs);


    try {
        testBadConstruct_Wind(&time_vec_hrs);

        testProductionConstraint_Wind(test_wind_ptr);

        testCommit_Wind(test_wind_ptr);
        testEconomics_Wind(test_wind_ptr);
    }


    catch (...) {
        delete test_wind_ptr;

        printGold(" ........ ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_wind_ptr;

    printGold(" ........ ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.64.2.2 testBadConstruct_Wind()

```
void testBadConstruct_Wind (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Wind object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
        WindInputs bad_wind_inputs;
        bad_wind_inputs.design_speed_ms = -1;

        Wind bad_wind(8760, 1, bad_wind_inputs, time_vec_hrs_ptr);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Wind() */
```

5.64.2.3 testCommit_Wind()

```
void testCommit_Wind (
            Renewable * test_wind_ptr )
```

Function to test if the commit method is working as expected, by checking some post-call attributes of the test Wind object. Uses a randomized resource input.

Parameters

test_wind_ptr	A Renewable pointer to the test Wind object.

```
{
    std::vector<double> dt_vec_hrs (48, 1);

    std::vector<double> load_vec_kW = {
        1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
        1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
        1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
    };

    double load_kW = 0;
    double production_kW = 0;
    double roll = 0;
    double wind_resource_ms = 0;

    for (int i = 0; i < 48; i++) {
        roll = (double)rand() / RAND_MAX;

        wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;

        roll = (double)rand() / RAND_MAX;

        if (roll <= 0.1) {
            wind_resource_ms = 0;
        }

        else if (roll >= 0.95) {
            wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
        }

        roll = (double)rand() / RAND_MAX;

        if (roll >= 0.95) {
            roll = 1.25;
        }

        load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
        load_kW = load_vec_kW[i];

        production_kW = test_wind_ptr->computeProductionkW(
            i,
            dt_vec_hrs[i],
```

```
            wind_resource_ms
        );

        load_kW = test_wind_ptr->commit(
            i,
            dt_vec_hrs[i],
            production_kW,
            load_kW
        );

        // is running (or not) as expected
        if (production_kW > 0) {
            testTruth(
                test_wind_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        else {
            testTruth(
                not test_wind_ptr->is_running,
                __FILE__,
                __LINE__
            );
        }

        // load_kW <= load_vec_kW (i.e., after vs before)
        testLessThanOrEqualTo(
            load_kW,
            load_vec_kW[i],
            __FILE__,
            __LINE__
        );

        // production = dispatch + storage + curtailment
        testFloatEquals(
            test_wind_ptr->production_vec_kW[i] -
            test_wind_ptr->dispatch_vec_kW[i] -
            test_wind_ptr->storage_vec_kW[i] -
            test_wind_ptr->curtailment_vec_kW[i],
            0,
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testCommit_Wind() */
```

5.64.2.4 testConstruct_Wind()

Renewable * testConstruct_Wind (
std::vector< double > * *time_vec_hrs_ptr*)

A function to construct a Wind object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A Renewable pointer to a test Wind object.

```
{
    WindInputs wind_inputs;

    Renewable* test_wind_ptr = new Wind(8760, 1, wind_inputs, time_vec_hrs_ptr);

    testTruth(
        not wind_inputs.renewable_inputs.production_inputs.print_flag,
```

```
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->type,
        RenewableType :: WIND,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_wind_ptr->type_str == "WIND",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->capital_cost,
        450356.170088,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->operation_maintenance_cost_kWh,
        0.034953,
        __FILE__,
        __LINE__
    );

    return test_wind_ptr;
}   /* testConstruct_Wind() */
```

5.64.2.5 testEconomics_Wind()

```
void testEconomics_Wind (
        Renewable * test_wind_ptr )
{
    for (int i = 0; i < 48; i++) {
        // resource, O&M > 0 whenever wind is running (i.e., producing)
        if (test_wind_ptr->is_running_vec[i]) {
            testGreaterThan(
                test_wind_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }

        // resource, O&M = 0 whenever wind is not running (i.e., not producing)
        else {
            testFloatEquals(
                test_wind_ptr->operation_maintenance_cost_vec[i],
                0,
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testEconomics_Wind() */
```

5.64.2.6 testProductionConstraint_Wind()

```
void testProductionConstraint_Wind (
            Renewable * test_wind_ptr )
```

Function to test that the production constraint is active and behaving as expected.

Parameters

test_wind_ptr	A Renewable pointer to the test Wind object.

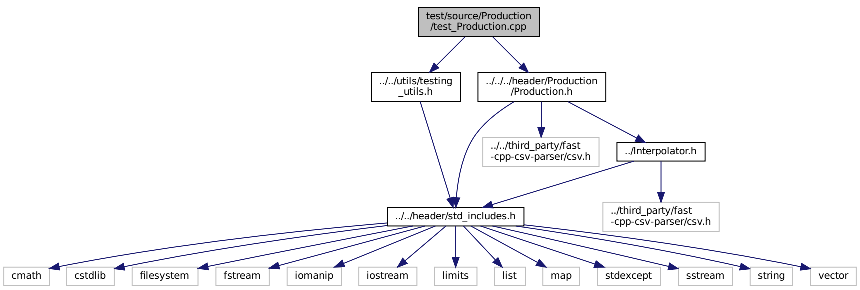
```
{
    testFloatEquals(
        test_wind_ptr->computeProductionkW(0, 1, 1e6),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->computeProductionkW(
            0,
            1,
            ((Wind*)test_wind_ptr)->design_speed_ms
        ),
        test_wind_ptr->capacity_kW,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_wind_ptr->computeProductionkW(0, 1, -1),
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testProductionConstraint_Wind() */
```

5.65 test/source/Production/test_Production.cpp File Reference

Testing suite for Production class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Production/Production.h"
```

Include dependency graph for test_Production.cpp:



Functions

- Production ∗ testConstruct_Production (std::vector< double > ∗time_vec_hrs_ptr)

    A function to construct a Production object and spot check some post-construction attributes.
- void testBadConstruct_Production (std::vector< double > ∗time_vec_hrs_ptr)

    Function to test the trying to construct a Production object given bad inputs is being handled as expected.
- int main (int argc, char ∗∗argv)

5.65.1 Detailed Description

Testing suite for Production class.

A suite of tests for the Production class.

5.65.2 Function Documentation

5.65.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Production");

    srand(time(NULL));


    std::vector<double> time_vec_hrs (8760, 0);
    for (size_t i = 0; i < time_vec_hrs.size(); i++) {
        time_vec_hrs[i] = i;
    }

    Production* test_production_ptr = testConstruct_Production(&time_vec_hrs);


    try {
        testBadConstruct_Production(&time_vec_hrs);
    }


    catch (...) {
        delete test_production_ptr;

        printGold(" .............................. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_production_ptr;

    printGold(" .............................. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.65.2.2 testBadConstruct_Production()

```
void testBadConstruct_Production (
            std::vector< double > * time_vec_hrs_ptr )
```

Function to test the trying to construct a Production object given bad inputs is being handled as expected.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

```
{
    bool error_flag = true;

    try {
        ProductionInputs production_inputs;

        Production bad_production(0, 1, production_inputs, time_vec_hrs_ptr);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Production() */
```

5.65.2.3 testConstruct_Production()

```
Production * testConstruct_Production (
            std::vector< double > * time_vec_hrs_ptr )
```

A function to construct a Production object and spot check some post-construction attributes.

Parameters

time_vec_hrs_ptr	A pointer to the vector containing the modelling time series.

Returns

A pointer to a test Production object.

```
{
    ProductionInputs production_inputs;

    Production* test_production_ptr = new Production(
        8760,
        1,
        production_inputs,
        time_vec_hrs_ptr
    );

    testTruth(
        not production_inputs.print_flag,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        production_inputs.nominal_inflation_annual,
        0.02,
        __FILE__,
```

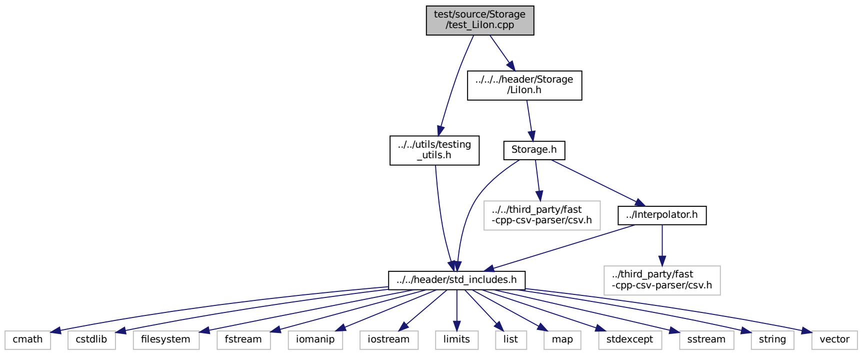
```
        __LINE__
    );

    testFloatEquals(
        production_inputs.nominal_discount_annual,
        0.04,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->capacity_kW,
        100,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->real_discount_annual,
        0.0196078431372549,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->production_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->dispatch_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->storage_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->curtailment_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->capital_cost_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_production_ptr->operation_maintenance_cost_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    return test_production_ptr;
}   /* testConstruct_Production() */
```

5.66 test/source/Storage/test_LiIon.cpp File Reference

Testing suite for LiIon class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Storage/LiIon.h"
```

Include dependency graph for test_LiIon.cpp:



Functions

- Storage * testConstruct_LiIon (void)

  A function to construct a LiIon object and spot check some post-construction attributes.
- void testBadConstruct_LiIon (void)

  Function to test the trying to construct a LiIon object given bad inputs is being handled as expected.
- void testCommitCharge_LiIon (Storage *test_liion_ptr)

  A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.
- void testCommitDischarge_LiIon (Storage *test_liion_ptr)

  A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.
- int main (int argc, char **argv)

5.66.1 Detailed Description

Testing suite for LiIon class.

A suite of tests for the LiIon class.

5.66.2 Function Documentation

5.66.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Storage <-- LiIon");

    srand(time(NULL));


    Storage* test_liion_ptr = testConstruct_LiIon();


    try {
        testBadConstruct_LiIon();

        testCommitCharge_LiIon(test_liion_ptr);
        testCommitDischarge_LiIon(test_liion_ptr);
    }


    catch (...) {
        delete test_liion_ptr;

        printGold(" ........................ ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_liion_ptr;

    printGold(" ........................ ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.66.2.2 testBadConstruct_LiIon()

```
void testBadConstruct_LiIon (
            void  )
```

Function to test the trying to construct a LiIon object given bad inputs is being handled as expected.

```
{
    bool error_flag = true;

    try {
        LiIonInputs bad_liion_inputs;
        bad_liion_inputs.min_SOC = -1;

        LiIon bad_liion(8760, 1, bad_liion_inputs);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_LiIon() */
```

5.66.2.3 testCommitCharge_LiIon()

```
void testCommitCharge_LiIon (
            Storage * test_liion_ptr )
```

A function to test commitCharge() and ensure that its impact on acceptable and available power is as expected.

Parameters

test_liion_ptr	A Storage pointer to a test LiIon object.

```
{
    double dt_hrs = 1;

    testFloatEquals(
        test_liion_ptr->getAvailablekW(dt_hrs),
        100,    // hits power capacity constraint
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_liion_ptr->getAcceptablekW(dt_hrs),
        100,    // hits power capacity constraint
        __FILE__,
        __LINE__
    );

    test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing in

    testFloatEquals(
        test_liion_ptr->getAvailablekW(dt_hrs),
        0,    // is already hitting power capacity constraint
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_liion_ptr->getAcceptablekW(dt_hrs),
        0,    // is already hitting power capacity constraint
        __FILE__,
        __LINE__
    );

    test_liion_ptr->commitCharge(0, dt_hrs, 100);

    testFloatEquals(
        test_liion_ptr->power_kW,
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testCommitCharge_LiIon() */
```

5.66.2.4 testCommitDischarge_LiIon()

```
void testCommitDischarge_LiIon (
            Storage * test_liion_ptr )
```

A function to test commitDischarge() and ensure that its impact on acceptable and available power is as expected.

Parameters

test_liion_ptr	A Storage pointer to a test LiIon object.

```
{
```

```
    double dt_hrs = 1;
    double load_kW = 100;

    testFloatEquals(
        test_liion_ptr->getAvailablekW(dt_hrs),
        100,    // hits power capacity constraint
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_liion_ptr->getAcceptablekW(dt_hrs),
        100,    // hits power capacity constraint
        __FILE__,
        __LINE__
    );

    test_liion_ptr->power_kW = 1e6; // as if a massive amount of power is already flowing out

    testFloatEquals(
        test_liion_ptr->getAvailablekW(dt_hrs),
        0,    // is already hitting power capacity constraint
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_liion_ptr->getAcceptablekW(dt_hrs),
        0,    // is already hitting power capacity constraint
        __FILE__,
        __LINE__
    );

    load_kW = test_liion_ptr->commitDischarge(0, dt_hrs, 100, load_kW);

    testFloatEquals(
        load_kW,
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_liion_ptr->power_kW,
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testCommitDischarge_LiIon() */
```

5.66.2.5 testConstruct_LiIon()

```
Storage * testConstruct_LiIon (
            void )
```

A function to construct a LiIon object and spot check some post-construction attributes.

Returns

A Storage pointer to a test LiIon object.

```
{
    LiIonInputs liion_inputs;

    Storage* test_liion_ptr = new LiIon(8760, 1, liion_inputs);

    testTruth(
        test_liion_ptr->type_str == "LIION",
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->init_SOC,
```

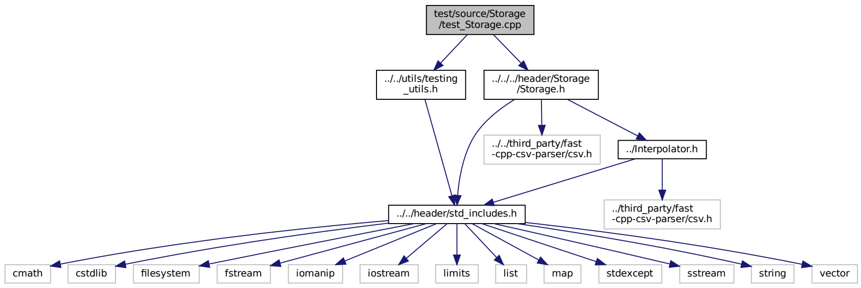
```
        0.5,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->min_SOC,
        0.15,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->hysteresis_SOC,
        0.5,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->max_SOC,
        0.9,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->charging_efficiency,
        0.9,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->discharging_efficiency,
        0.9,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->replace_SOH,
        0.8,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->power_kW,
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        ((LiIon*)test_liion_ptr)->SOH_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    return test_liion_ptr;
}   /* testConstruct_LiIon() */
```

5.67 test/source/Storage/test_Storage.cpp File Reference

Testing suite for Storage class.

```
#include "../../utils/testing_utils.h"
#include "../../../header/Storage/Storage.h"
```

Include dependency graph for test_Storage.cpp:



Functions

- Storage ∗ testConstruct_Storage (void)

    A function to construct a Storage object and spot check some post-construction attributes.
- void testBadConstruct_Storage (void)

    Function to test the trying to construct a Storage object given bad inputs is being handled as expected.
- int main (int argc, char ∗∗argv)

5.67.1 Detailed Description

Testing suite for Storage class.

A suite of tests for the Storage class.

5.67.2 Function Documentation

5.67.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Storage");

    srand(time(NULL));


    Storage* test_storage_ptr = testConstruct_Storage();


    try {
        testBadConstruct_Storage();
    }


    catch (...) {
```

```
        delete test_storage_ptr;

        printGold(" .................................. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_storage_ptr;

    printGold(" .................................. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;

}   /* main() */
```

5.67.2.2 testBadConstruct_Storage()

```
void testBadConstruct_Storage (
            void )
```

Function to test the trying to construct a Storage object given bad inputs is being handled as expected.

```
{
    bool error_flag = true;

    try {
        StorageInputs bad_storage_inputs;
        bad_storage_inputs.energy_capacity_kWh = 0;

        Storage bad_storage(8760, 1, bad_storage_inputs);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadConstruct_Storage() */
```

5.67.2.3 testConstruct_Storage()

```
Storage * testConstruct_Storage (
            void )
```

A function to construct a Storage object and spot check some post-construction attributes.

Returns

A Renewable pointer to a test Storage object.

```
{
    StorageInputs storage_inputs;

    Storage* test_storage_ptr = new Storage(8760, 1, storage_inputs);

    testFloatEquals(
        test_storage_ptr->power_capacity_kW,
        100,
        __FILE__,
        __LINE__
    );

```

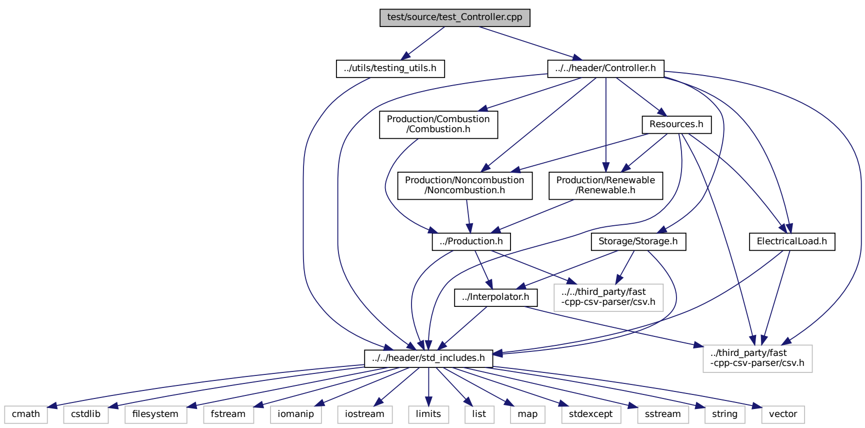
```
    testFloatEquals(
        test_storage_ptr->energy_capacity_kWh,
        1000,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_storage_ptr->charge_vec_kWh.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_storage_ptr->charging_power_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_storage_ptr->discharging_power_vec_kW.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_storage_ptr->capital_cost_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_storage_ptr->operation_maintenance_cost_vec.size(),
        8760,
        __FILE__,
        __LINE__
    );

    return test_storage_ptr;
}   /* testConstruct_Storage() */
```

5.68 test/source/test_Controller.cpp File Reference

Testing suite for Controller class.

```
#include "../utils/testing_utils.h"
#include "../../header/Controller.h"
```

Include dependency graph for test_Controller.cpp:



Functions

- Controller * testConstruct_Controller (void)

    A function to construct a Controller object.
- int main (int argc, char **argv)

5.68.1 Detailed Description

Testing suite for Controller class.

A suite of tests for the Controller class.

5.68.2 Function Documentation

5.68.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Controller");

    srand(time(NULL));


    Controller* test_controller_ptr = testConstruct_Controller();


    try {
        //...
    }


    catch (...) {
        delete test_controller_ptr;

        printGold(" .............................. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_controller_ptr;

    printGold(" .............................. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.68.2.2 testConstruct_Controller()

```
Controller * testConstruct_Controller (
             void  )
```

A function to construct a Controller object.

Returns

A pointer to a test Controller object.

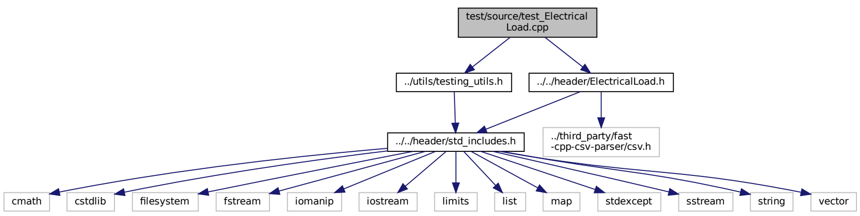
```
{
    Controller* test_controller_ptr = new Controller();

    return test_controller_ptr;
}   /* testConstruct_Controller() */
```

5.69 test/source/test_ElectricalLoad.cpp File Reference

Testing suite for ElectricalLoad class.

```
#include "../utils/testing_utils.h"
#include "../../header/ElectricalLoad.h"
```

Include dependency graph for test_ElectricalLoad.cpp:



Functions

- ElectricalLoad * testConstruct_ElectricalLoad (void)

    A function to construct an ElectricalLoad object.
- void testPostConstructionAttributes_ElectricalLoad (ElectricalLoad *test_electrical_load_ptr)

    A function to check the values of various post-construction attributes.
- void testDataRead_ElectricalLoad (ElectricalLoad *test_electrical_load_ptr)

    A function to check the values read into the test ElectricalLoad object.
- int main (int argc, char **argv)

5.69.1 Detailed Description

Testing suite for ElectricalLoad class.

A suite of tests for the ElectricalLoad class.

5.69.2 Function Documentation

5.69.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting ElectricalLoad");

    srand(time(NULL));


    ElectricalLoad* test_electrical_load_ptr = testConstruct_ElectricalLoad();


    try {
        testPostConstructionAttributes_ElectricalLoad(test_electrical_load_ptr);
        testDataRead_ElectricalLoad(test_electrical_load_ptr);
    }


    catch (...) {
        delete test_electrical_load_ptr;

        printGold(" .......................... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_electrical_load_ptr;

    printGold(" .......................... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.69.2.2 testConstruct_ElectricalLoad()

```
ElectricalLoad * testConstruct_ElectricalLoad (
            void  )
```

A function to construct an ElectricalLoad object.

Returns

A pointer to a test ElectricalLoad object.

```
{
    std::string path_2_electrical_load_time_series =
        "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

    ElectricalLoad* test_electrical_load_ptr =
        new ElectricalLoad(path_2_electrical_load_time_series);

    testTruth(
        test_electrical_load_ptr->path_2_electrical_load_time_series ==
        path_2_electrical_load_time_series,
        __FILE__,
        __LINE__
    );

    return test_electrical_load_ptr;
}   /* testConstruct_ElectricalLoad() */
```

5.69.2.3 testDataRead_ElectricalLoad()

```
void testDataRead_ElectricalLoad (
            ElectricalLoad * test_electrical_load_ptr )
```

A function to check the values read into the test ElectricalLoad object.

Parameters

test_electrical_load_ptr	A pointer to the test ElectricalLoad object.

```
{
    std::vector<double> expected_dt_vec_hrs (48, 1);

    std::vector<double> expected_time_vec_hrs = {
         0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
        12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
        24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
    };

    std::vector<double> expected_load_vec_kW = {
        360.253836463674,
        355.171277826775,
        353.776453532298,
        353.75405737934,
        346.592867404975,
        340.132411175118,
        337.354867340578,
        340.644115618736,
        363.639028500678,
        378.787797779238,
        372.215798201712,
        395.093925731298,
        402.325427142659,
        386.907725462306,
        380.709170928091,
        372.062070914977,
        372.328646856954,
        391.841444284136,
        394.029351759596,
        383.369407765254,
        381.093099675206,
        382.604158946193,
        390.744843709034,
        383.13949492437,
        368.150393976985,
        364.629744480226,
        363.572736804082,
        359.854924202248,
        355.207590170267,
        349.094656012401,
        354.365935871597,
        343.380608328546,
        404.673065729266,
        486.296896820126,
        480.225974100847,
        457.318764401085,
        418.177339948609,
        414.399018364126,
        409.678420185754,
        404.768766016563,
        401.699589920585,
        402.44339040654,
        398.138372541906,
        396.010498627646,
        390.165117432277,
        375.850429417013,
        365.567100746484,
        365.429624610923
    };

    for (int i = 0; i < 48; i++) {
        testFloatEquals(
            test_electrical_load_ptr->dt_vec_hrs[i],
            expected_dt_vec_hrs[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
```

```
            test_electrical_load_ptr->time_vec_hrs[i],
            expected_time_vec_hrs[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_electrical_load_ptr->load_vec_kW[i],
            expected_load_vec_kW[i],
            __FILE__,
            __LINE__
        );

    }

    return;
}   /* testDataRead_ElectricalLoad() */
```

5.69.2.4 testPostConstructionAttributes_ElectricalLoad()

```
void testPostConstructionAttributes_ElectricalLoad (
            ElectricalLoad * test_electrical_load_ptr )
```

A function to check the values of various post-construction attributes.

Parameters

test_electrical_load_ptr	A pointer to the test ElectricalLoad object.

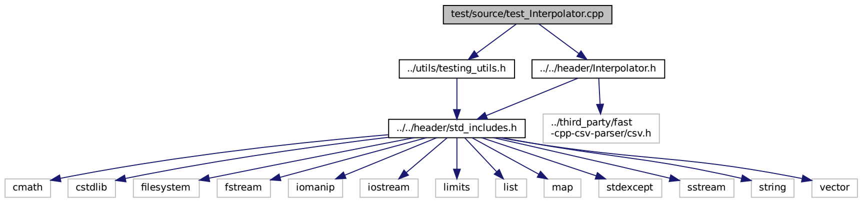
```
{
    testFloatEquals(
        test_electrical_load_ptr->n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_electrical_load_ptr->n_years,
        0.999886,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_electrical_load_ptr->min_load_kW,
        82.1211213927802,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_electrical_load_ptr->mean_load_kW,
        258.373472633202,
        __FILE__,
        __LINE__
    );


    testFloatEquals(
        test_electrical_load_ptr->max_load_kW,
        500,
        __FILE__,
        __LINE__
    );

    return;
}   /* testPostConstructionAttributes_ElectricalLoad() */
```

5.70 test/source/test_Interpolator.cpp File Reference

Testing suite for Interpolator class.

```
#include "../utils/testing_utils.h"
#include "../../header/Interpolator.h"
```

Include dependency graph for test_Interpolator.cpp:



Functions

- Interpolator ∗ testConstruct_Interpolator (void)

    A function to construct an Interpolator object.
- void testDataRead1D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_1D, std::string path_2←_data_1D)

    A function to check the 1D data values read into the Interpolator object.
- void testBadIndexing1D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_bad)

    A function to check if bad key errors are being handled properly.
- void testInvalidInterpolation1D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_1D)

    Function to check if attempting to interpolate outside the given 1D data domain is handled properly.
- void testInterpolation1D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_1D)

    Function to check that the Interpolator object is returning the expected 1D interpolation values.
- void testDataRead2D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_2D, std::string path_2←_data_2D)

    A function to check the 2D data values read into the Interpolator object.
- void testInvalidInterpolation2D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_2D)

    Function to check if attempting to interpolate outside the given 2D data domain is handled properly.
- void testInterpolation2D_Interpolator (Interpolator ∗test_interpolator_ptr, int data_key_2D)

    Function to check that the Interpolator object is returning the expected 2D interpolation values.
- int main (int argc, char ∗∗argv)

5.70.1 Detailed Description

Testing suite for Interpolator class.

A suite of tests for the Interpolator class.

5.70.2 Function Documentation

5.70.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\n\tTesting Interpolator");

    srand(time(NULL));


    Interpolator* test_interpolator_ptr = testConstruct_Interpolator();


    try {
        int data_key_1D = 1;
        std::string path_2_data_1D =
            "data/test/interpolation/diesel_fuel_curve.csv";

        testDataRead1D_Interpolator(test_interpolator_ptr, data_key_1D, path_2_data_1D);
        testBadIndexing1D_Interpolator(test_interpolator_ptr, -99);
        testInvalidInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);
        testInterpolation1D_Interpolator(test_interpolator_ptr, data_key_1D);


        int data_key_2D = 2;
        std::string path_2_data_2D =
            "data/test/interpolation/wave_energy_converter_normalized_performance_matrix.csv";

        testDataRead2D_Interpolator(test_interpolator_ptr, data_key_2D, path_2_data_2D);
        testInvalidInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
        testInterpolation2D_Interpolator(test_interpolator_ptr, data_key_2D);
    }


    catch (...) {
        delete test_interpolator_ptr;

        printGold(" ............................. ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_interpolator_ptr;

    printGold(" ............................. ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.70.2.2 testBadIndexing1D_Interpolator()

```
void testBadIndexing1D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_bad )
```

A function to check if bad key errors are being handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_bad	A key used to index into the Interpolator object.

```
{
```

```
    bool error_flag = true;

    try {
        test_interpolator_ptr->interp1D(data_key_bad, 0);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testBadIndexing1D_Interpolator() */
```

5.70.2.3 testConstruct_Interpolator()

```
Interpolator * testConstruct_Interpolator (
                void )
```

A function to construct an Interpolator object.

Returns

A pointer to a test Interpolator object.

```
{
    Interpolator* test_interpolator_ptr = new Interpolator();

    return test_interpolator_ptr;
}   /* testConstruct_Interpolator() */
```

5.70.2.4 testDataRead1D_Interpolator()

```
void testDataRead1D_Interpolator (
                Interpolator * test_interpolator_ptr,
                int data_key,
                std::string path_2_data_1D )
```

A function to check the 1D data values read into the Interpolator object.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.
path_2_data_1D	A path (either relative or absolute) to the interpolation data.

```
{
    test_interpolator_ptr->addData1D(data_key_1D, path_2_data_1D);

    testTruth(
        test_interpolator_ptr->path_map_1D[data_key_1D] == path_2_data_1D,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_1D[data_key_1D].n_points,
        16,
        __FILE__,
```

```
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec.size(),
        16,
        __FILE__,
        __LINE__
    );

    std::vector<double> expected_x_vec = {
        0,
        0.3,
        0.35,
        0.4,
        0.45,
        0.5,
        0.55,
        0.6,
        0.65,
        0.7,
        0.75,
        0.8,
        0.85,
        0.9,
        0.95,
        1
    };

    std::vector<double> expected_y_vec = {
        4.68079520372916,
        11.1278522361839,
        12.4787834830748,
        13.7808847600209,
        15.0417468303382,
        16.277263,
        17.4612831516442,
        18.6279054806525,
        19.7698039220515,
        20.8893499214868,
        21.955378,
        23.0690535155297,
        24.1323614374927,
        25.1797231192866,
        26.2122451458747,
        27.254952
    };

    for (int i = 0; i < test_interpolator_ptr->interp_map_1D[data_key_1D].n_points; i++) {
        testFloatEquals(
            test_interpolator_ptr->interp_map_1D[data_key_1D].x_vec[i],
            expected_x_vec[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_interpolator_ptr->interp_map_1D[data_key_1D].y_vec[i],
            expected_y_vec[i],
            __FILE__,
            __LINE__
        );
    }

    testFloatEquals(
        test_interpolator_ptr->interp_map_1D[data_key_1D].min_x,
        expected_x_vec[0],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_1D[data_key_1D].max_x,
        expected_x_vec[expected_x_vec.size() - 1],
        __FILE__,
        __LINE__
    );

    return;
}   /* testDataRead1D_Interpolator() */
```

5.70.2.5 testDataRead2D_Interpolator()

```
void testDataRead2D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key,
            std::string path_2_data_2D )
```

A function to check the 2D data values read into the Interpolator object.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.
path_2_data_2D	A path (either relative or absolute) to the interpolation data.

```
{
    test_interpolator_ptr->addData2D(data_key_2D, path_2_data_2D);

    testTruth(
        test_interpolator_ptr->path_map_2D[data_key_2D] == path_2_data_2D,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows,
        16,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols,
        16,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec.size(),
        16,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec.size(),
        16,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix.size(),
        16,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[0].size(),
        16,
        __FILE__,
        __LINE__
    );

    std::vector<double> expected_x_vec = {
        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
    };

    std::vector <double> expected_y_vec = {
        5,
        6,
        7,
        8,
        9,
        10,
        11,
```

```
        12,
        13,
        14,
        15,
        16,
        17,
        18,
        19,
        20
    };

    for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; i++) {
        testFloatEquals(
            test_interpolator_ptr->interp_map_2D[data_key_2D].x_vec[i],
            expected_x_vec[i],
            __FILE__,
            __LINE__
        );
    }

    for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
        testFloatEquals(
            test_interpolator_ptr->interp_map_2D[data_key_2D].y_vec[i],
            expected_y_vec[i],
            __FILE__,
            __LINE__
        );
    }

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].min_x,
        expected_x_vec[0],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].max_x,
        expected_x_vec[expected_x_vec.size() - 1],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].min_y,
        expected_y_vec[0],
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_interpolator_ptr->interp_map_2D[data_key_2D].max_y,
        expected_y_vec[expected_y_vec.size() - 1],
        __FILE__,
        __LINE__
    );

    std::vector<std::vector<double» expected_z_matrix = {
        {0, 0.129128125, 0.268078125, 0.404253125, 0.537653125, 0.668278125, 0.796128125, 0.921203125,
    1, 1, 1, 0, 0, 0, 0, 0},
        {0, 0.11160375, 0.24944375, 0.38395375, 0.51513375, 0.64298375, 0.76750375, 0.88869375, 1, 1, 1,
    1, 1, 1, 1, 1},
        {0, 0.094079375, 0.230809375, 0.363654375, 0.492614375, 0.617689375, 0.738879375, 0.856184375,
    0.969604375, 1, 1, 1, 1, 1, 1, 1},
        {0, 0.076555, 0.212175, 0.343355, 0.470095, 0.592395, 0.710255, 0.823675, 0.932655, 1, 1, 1, 1,
    1, 1, 1},
        {0, 0.059030625, 0.193540625, 0.323055625, 0.447575625, 0.567100625, 0.681630625, 0.791165625,
    0.895705625, 0.995250625, 1, 1, 1, 1, 1, 1},
        {0, 0.04150625, 0.17490625, 0.30275625, 0.42505625, 0.54180625, 0.65300625, 0.75865625,
    0.85875625, 0.95330625, 1, 1, 1, 1, 1, 1},
        {0, 0.023981875, 0.156271875, 0.282456875, 0.402536875, 0.516511875, 0.624381875, 0.726146875,
    0.821806875, 0.911361875, 0.994811875, 1, 1, 1, 1, 1},
        {0, 0.0064575, 0.1376375, 0.2621575, 0.3800175, 0.4912175, 0.5957575, 0.6936375, 0.7848575,
    0.8694175, 0.9473175, 1, 1, 1, 1, 1},
        {0, 0, 0.119003125, 0.241858125, 0.357498125, 0.465923125, 0.567133125, 0.661128125,
    0.747908125, 0.827473125, 0.899823125, 0.964958125, 1, 1, 1, 1},
        {0, 0, 0.10036875, 0.22155875, 0.33497875, 0.44062875, 0.53850875, 0.62861875, 0.71095875,
    0.78552875, 0.85232875, 0.91135875, 0.96261875, 1, 1, 1},
        {0, 0, 0.081734375, 0.201259375, 0.312459375, 0.415334375, 0.509884375, 0.596109375,
    0.674009375, 0.743584375, 0.804834375, 0.857759375, 0.902359375, 0.938634375, 0.966584375,
    0.986209375},
        {0, 0, 0.0631, 0.18096, 0.28994, 0.39004, 0.48126, 0.5636, 0.63706, 0.70164, 0.75734, 0.80416,
    0.8421, 0.87116, 0.89134, 0.90264},
        {0, 0, 0.044465625, 0.160660625, 0.267420625, 0.364745625, 0.452635625, 0.531090625,
    0.600110625, 0.659695625, 0.709845625, 0.750560625, 0.781840625, 0.803685624999999, 0.816095625,
    0.819070625},
        {0, 0, 0.02583125, 0.14036125, 0.24490125, 0.33945125, 0.42401125, 0.49858125, 0.56316125,
```

```
    0.61775125, 0.66235125, 0.69696125, 0.72158125, 0.73621125, 0.74085125, 0.73550125},
        {0, 0, 0.007196875, 0.120061875, 0.222381875, 0.314156875, 0.395386875, 0.466071875,
    0.526211875, 0.575806875, 0.614856875, 0.643361875, 0.661321875, 0.668736875, 0.665606875,
    0.651931875},
        {0, 0, 0, 0.0997625, 0.1998625, 0.2888625, 0.3667625, 0.4335625, 0.4892625, 0.5338625,
    0.5673625, 0.5897625, 0.6010625, 0.6012625, 0.5903625, 0.5683625}
    };

    for (int i = 0; i < test_interpolator_ptr->interp_map_2D[data_key_2D].n_rows; i++) {
        for (int j = 0; j < test_interpolator_ptr->interp_map_2D[data_key_2D].n_cols; j++) {
            testFloatEquals(
                test_interpolator_ptr->interp_map_2D[data_key_2D].z_matrix[i][j],
                expected_z_matrix[i][j],
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testDataRead2D_Interpolator() */
```

5.70.2.6 testInterpolation1D_Interpolator()

```
void testInterpolation1D_Interpolator (
        Interpolator * test_interpolator_ptr,
        int data_key_1D )
```

Function to check that the Interpolator object is returning the expected 1D interpolation values.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.

```
{
    std::vector<double> interp_x_vec = {
        0,
        0.170812859791767,
        0.322739274162545,
        0.369750203682042,
        0.443532869135929,
        0.471567864244626,
        0.536513734479662,
        0.586125806988674,
        0.601101175455075,
        0.658356862575221,
        0.70576929893201,
        0.784069734739331,
        0.805765927542453,
        0.884747873186048,
        0.930870496062112,
        0.979415217694769,
        1
    };

    std::vector<double> expected_interp_y_vec = {
        4.68079520372916,
        8.35159603357656,
        11.7422361561399,
        12.9931187917615,
        14.8786636301325,
        15.5746957307243,
        17.1419229487141,
        18.3041866133728,
        18.6530540913696,
        19.9569217633299,
        21.012354614584,
        22.7142305879957,
        23.1916726441968,
        24.8602332554707,
        25.8172124624032,
        26.8256741279932,
        27.254952
```

```
    };

    for (size_t i = 0; i < interp_x_vec.size(); i++) {
        testFloatEquals(
            test_interpolator_ptr->interp1D(data_key_1D, interp_x_vec[i]),
            expected_interp_y_vec[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testInterpolation1D_Interpolator() */
```

5.70.2.7 testInterpolation2D_Interpolator()

```
void testInterpolation2D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_2D )
```

Function to check that the Interpolator object is returning the expected 2D interpolation values.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.

```
{
    std::vector<double> interp_x_vec = {
        0.389211848822208,
        0.836477431896843,
        1.52738334015579,
        1.92640601114508,
        2.27297317532019,
        2.87416589636605,
        3.72275770908175,
        3.95063175885536,
        4.68097139867404,
        4.97775020449812,
        5.55184219980547,
        6.06566629451658,
        6.27927876785062,
        6.96218133671013,
        7.51754442460228
    };

    std::vector<double> interp_y_vec = {
        5.45741899698926,
        6.00101329139007,
        7.50567689404182,
        8.77681262912881,
        9.45143678206774,
        10.7767876462885,
        11.4795760857165,
        12.9430684577599,
        13.303544885703,
        14.5069863517863,
        15.1487890438045,
        16.086524049077,
        17.176609978648,
        18.4155153740256,
        19.1704554940162
    };

    std::vector<std::vector<double» expected_interp_z_matrix = {

   {0.0337204906738533,0.145056406036013,0.334677248806653,0.441674658936075,0.533295755691263,0.68807895676592,0.89961488

   {0.0310681846933292,0.135425896595439,0.324045598153363,0.430214268249038,0.520985043044784,0.673879556322479,0.8820580

   {0.0237266281076604,0.108768742207538,0.294617294841705,0.398492020763049,0.486909112828702,0.63457575706117,0.8334608

   {0.0175245009938255,0.0862488504001753,0.269756343931147,0.371693152028768,0.458121859300634,0.601372013927032,0.79240
```

```
    {0.0142328739589644,0.0742969694833995,0.256562003243255,0.357470308928265,0.442843729679424,0.583749940636223,0.77061
    {0.0077662203173173,0.0508165832074184,0.230640709501637,0.329528443353471,0.41282867283787,0.549130026772199,0.727811
    {0.00433717405958826,0.0383657337957315,0.21689552996585,0.314711823368423,0.396912710109449,0.530772265145106,0.70511
    {0.000102358416923608,0.0210697053701168,0.188272456115393,0.283857573197153,0.363769179652786,0.492543912767949,0.657
    {0,0.0196038727057393,0.181222235960193,0.276257786480759,0.355605514643888,0.483127792688125,0.646203044346932,0.6855
    {0,0.015725294236768,0.157685253727545,0.250886090139653,0.328351324840186,0.451692313207986,0.607334650020078,0.6442
    {0,0.0136568246246201,0.145132837191606,0.23735520935175,0.313816498778623,0.43492757979648,0.586605897674033,0.622265
    {0,0.0106345930466366,0.12679255826648,0.217585300741544,0.292579730277991,0.410432703770651,0.556319211544087,0.59010
    {0,0.00712134879261874,0.10547259059088,0.194603435839713,0.267892689267542,0.381958220518761,0.52111194060085,0.55272
    {0,0.00312847342058727,0.0812420026472571,0.168484067035528,0.239835352250276,0.349596376397684,0.481098142839729,0.51
    {0,0.00103256269522045,0.0673448574082101,0.152567953107312,0.222738316872545,0.329876344040866,0.456715311514779,0.48
    };

    for (size_t i = 0; i < interp_y_vec.size(); i++) {
        for (size_t j = 0; j < interp_x_vec.size(); j++) {
            testFloatEquals(
                test_interpolator_ptr->interp2D(data_key_2D, interp_x_vec[j], interp_y_vec[i]),
                expected_interp_z_matrix[i][j],
                __FILE__,
                __LINE__
            );
        }
    }

    return;
}   /* testInterpolation2D_Interpolator() */
```

5.70.2.8 testInvalidInterpolation1D_Interpolator()

```
void testInvalidInterpolation1D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_1D )
```

Function to check if attempting to interpolate outside the given 1D data domain is handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_1D	A key used to index into the Interpolator object.

```
{
    bool error_flag = true;

    try {
        test_interpolator_ptr->interp1D(data_key_1D, -1);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        test_interpolator_ptr->interp1D(data_key_1D, 2);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

```

```
    try {
        test_interpolator_ptr->interp1D(data_key_1D, 0 - FLOAT_TOLERANCE);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        test_interpolator_ptr->interp1D(data_key_1D, 1 + FLOAT_TOLERANCE);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testInvalidInterpolation1D_Interpolator() */
```

5.70.2.9 testInvalidInterpolation2D_Interpolator()

```
void testInvalidInterpolation2D_Interpolator (
            Interpolator * test_interpolator_ptr,
            int data_key_2D )
```

Function to check if attempting to interpolate outside the given 2D data domain is handled properly.

Parameters

test_interpolator_ptr	A pointer to the test Interpolator object.
data_key_2D	A key used to index into the Interpolator object.

```
{
    bool error_flag = true;

    try {
        test_interpolator_ptr->interp2D(data_key_2D, -1, 6);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        test_interpolator_ptr->interp2D(data_key_2D, 99, 6);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        test_interpolator_ptr->interp2D(data_key_2D, 0.75, -1);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        test_interpolator_ptr->interp2D(data_key_2D, 0.75, 99);
        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
```

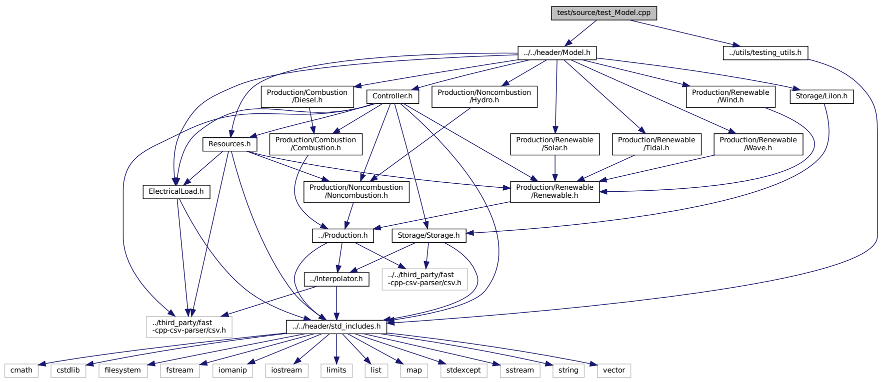
```
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}   /* testInvalidInterpolation2D_Interpolator() */
```

5.71 test/source/test_Model.cpp File Reference

Testing suite for Model class.

```
#include "../utils/testing_utils.h"
#include "../../header/Model.h"
```

Include dependency graph for test_Model.cpp:



Functions

- Model ∗ testConstruct_Model (ModelInputs test_model_inputs)
- void testBadConstruct_Model (void)

    Function to check if passing bad ModelInputs to the Model constructor is handled appropriately.
- void testPostConstructionAttributes_Model (Model ∗test_model_ptr)

    A function to check the values of various post-construction attributes.
- void testElectricalLoadData_Model (Model ∗test_model_ptr)

    Function to check the values read into the ElectricalLoad component of the test Model object.
- void testAddSolarResource_Model (Model ∗test_model_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

    Function to test adding a solar resource and then check the values read into the Resources component of the test Model object.
- void testAddTidalResource_Model (Model ∗test_model_ptr, std::string path_2_tidal_resource_data, int tidal↩_resource_key)

    Function to test adding a tidal resource and then check the values read into the Resources component of the test Model object.
- void testAddWaveResource_Model (Model ∗test_model_ptr, std::string path_2_wave_resource_data, int wave_resource_key)

    Function to test adding a wave resource and then check the values read into the Resources component of the test Model object.
- void testAddWindResource_Model (Model ∗test_model_ptr, std::string path_2_wind_resource_data, int wind_resource_key)

Function to test adding a wind resource and then check the values read into the Resources component of the test Model object.

- void testAddHydroResource_Model (Model *test_model_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)

  Function to test adding a hydro resource and then check the values read into the Resources component of the test Model object.
- void testAddHydro_Model (Model *test_model_ptr, int hydro_resource_key)

  Function to test adding a hydroelectric asset to the test Model object, and then spot check some post-add attributes.
- void testAddDiesel_Model (Model *test_model_ptr)

  Function to test adding a suite of diesel generators to the test Model object, and then spot check some post-add attributes.
- void testAddSolar_Model (Model *test_model_ptr, int solar_resource_key)

  Function to test adding a solar PV array to the test Model object and then spot check some post-add attributes.
- void testAddSolar_productionOverride_Model (Model *test_model_ptr, std::string path_2_normalized_↩ production_time_series)

  Function to test adding a solar PV array to the test Model object using the production override feature, and then spot check some post-add attributes.
- void testAddTidal_Model (Model *test_model_ptr, int tidal_resource_key)

  Function to test adding a tidal turbine to the test Model object and then spot check some post-add attributes.
- void testAddWave_Model (Model *test_model_ptr, int wave_resource_key)

  Function to test adding a wave energy converter to the test Model object and then spot check some post-add attributes.
- void testAddWind_Model (Model *test_model_ptr, int wind_resource_key)

  Function to test adding a wind turbine to the test Model object and then spot check some post-add attributes.
- void testAddLiIon_Model (Model *test_model_ptr)

  Function to test adding a lithium ion battery energy storage system to the test Model object and then spot check some post-add attributes.
- void testLoadBalance_Model (Model *test_model_ptr)

  Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the Controller as expected.
- void testEconomics_Model (Model *test_model_ptr)

  Function to check that the modelled economic metrics are > 0.
- void testFuelConsumptionEmissions_Model (Model *test_model_ptr)

  Function to check that the modelled fuel consumption and emissions are > 0.
- int main (int argc, char **argv)

5.71.1 Detailed Description

Testing suite for Model class.

A suite of tests for the Model class.

5.71.2 Function Documentation

5.71.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

```
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Model");
    std::cout « std::flush;

    srand(time(NULL));


    std::string path_2_electrical_load_time_series =
        "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

    ModelInputs test_model_inputs;
    test_model_inputs.path_2_electrical_load_time_series =
        path_2_electrical_load_time_series;

    Model* test_model_ptr = testConstruct_Model(test_model_inputs);


    try {
        testBadConstruct_Model();
        testPostConstructionAttributes_Model(test_model_ptr);
        testElectricalLoadData_Model(test_model_ptr);


        int solar_resource_key = 0;
        std::string path_2_solar_resource_data =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

        testAddSolarResource_Model(
            test_model_ptr,
            path_2_solar_resource_data,
            solar_resource_key
        );


        int tidal_resource_key = 1;
        std::string path_2_tidal_resource_data =
            "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

        testAddTidalResource_Model(
            test_model_ptr,
            path_2_tidal_resource_data,
            tidal_resource_key
        );


        int wave_resource_key = 2;
        std::string path_2_wave_resource_data =
            "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

        testAddWaveResource_Model(
            test_model_ptr,
            path_2_wave_resource_data,
            wave_resource_key
        );


        int wind_resource_key = 3;
        std::string path_2_wind_resource_data =
            "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";

        testAddWindResource_Model(
            test_model_ptr,
            path_2_wind_resource_data,
            wind_resource_key
        );


        int hydro_resource_key = 4;
        std::string path_2_hydro_resource_data =
            "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";

        testAddHydroResource_Model(
            test_model_ptr,
            path_2_hydro_resource_data,
            hydro_resource_key
        );
```

```
        std::string path_2_normalized_production_time_series =
                "data/test/normalized_production/normalized_solar_production.csv";

        // looping solely for the sake of profiling (also tests reset(), which is
        // needed for wrapping PGMcpp in an optimizer)
        for (int i = 0; i < 1000; i++) {
            test_model_ptr->reset();


            testAddHydro_Model(test_model_ptr, hydro_resource_key);
            testAddDiesel_Model(test_model_ptr);
            testAddSolar_Model(test_model_ptr, solar_resource_key);

            testAddSolar_productionOverride_Model(
                test_model_ptr,
                path_2_normalized_production_time_series
            );

            testAddTidal_Model(test_model_ptr, tidal_resource_key);
            testAddWave_Model(test_model_ptr, wave_resource_key);
            testAddWind_Model(test_model_ptr, wind_resource_key);


            test_model_ptr->run();
        }


        testLoadBalance_Model(test_model_ptr);
        testEconomics_Model(test_model_ptr);
        testFuelConsumptionEmissions_Model(test_model_ptr);

        test_model_ptr->writeResults("test/test_results/");
    }


    catch (...) {
        delete test_model_ptr;

        printGold(" ................................... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_model_ptr;

    printGold(" ................................... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.71.2.2 testAddDiesel_Model()

```
void testAddDiesel_Model (
            Model * test_model_ptr )
```

Function to test adding a suite of diesel generators to the test Model object, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    DieselInputs diesel_inputs;
    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
    diesel_inputs.combustion_inputs.production_inputs.is_sunk = true;

    test_model_ptr->addDiesel(diesel_inputs);

    testFloatEquals(
```

```
        test_model_ptr->combustion_ptr_vec.size(),
        1,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->combustion_ptr_vec[0]->type,
        CombustionType :: DIESEL,
        __FILE__,
        __LINE__
    );

    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;

    test_model_ptr->addDiesel(diesel_inputs);

    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;

    test_model_ptr->addDiesel(diesel_inputs);

    testFloatEquals(
        test_model_ptr->combustion_ptr_vec.size(),
        3,
        __FILE__,
        __LINE__
    );

    std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};

    for (int i = 0; i < 3; i++) {
        testFloatEquals(
            test_model_ptr->combustion_ptr_vec[i]->capacity_kW,
            expected_diesel_capacity_vec_kW[i],
            __FILE__,
            __LINE__
        );
    }

    diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;

    for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
        test_model_ptr->addDiesel(diesel_inputs);
    }

    return;
}   /* testAddDiesel_Model() */
```

5.71.2.3 testAddHydro_Model()

```
void testAddHydro_Model (
            Model * test_model_ptr,
            int hydro_resource_key )
```

Function to test adding a hydroelectric asset to the test Model object, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
hydro_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    HydroInputs hydro_inputs;
    hydro_inputs.noncombustion_inputs.production_inputs.capacity_kW = 300;
    hydro_inputs.reservoir_capacity_m3 = 100000;
    hydro_inputs.init_reservoir_state = 0.5;
    hydro_inputs.noncombustion_inputs.production_inputs.is_sunk = true;
    hydro_inputs.resource_key = hydro_resource_key;

    test_model_ptr->addHydro(hydro_inputs);

    testFloatEquals(
        test_model_ptr->noncombustion_ptr_vec.size(),
```

```
        1,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->noncombustion_ptr_vec[0]->type,
        NoncombustionType :: HYDRO,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->noncombustion_ptr_vec[0]->resource_key,
        hydro_resource_key,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddHydro_Model() */
```

5.71.2.4 testAddHydroResource_Model()

```
void testAddHydroResource_Model (
            Model * test_model_ptr,
            std::string path_2_hydro_resource_data,
            int hydro_resource_key )
```

Function to test adding a hydro resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_hydro_resource_data	A path (either relative or absolute) to the hydro resource data.
hydro_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    test_model_ptr->addResource(
        NoncombustionType :: HYDRO,
        path_2_hydro_resource_data,
        hydro_resource_key
    );

    std::vector<double> expected_hydro_resource_vec_ms = {
        2167.91531556942,
        2046.58261560569,
        2007.85941123153,
        2000.11477247929,
        1917.50527264453,
        1963.97311577093,
        1908.46985899809,
        1886.5267112678,
        1965.26388854254,
        1953.64692935289,
        2084.01504296306,
        2272.46796101188,
        2520.29645627096,
        2715.203242423,
        2720.36633563203,
        3130.83228077221,
        3289.59741021591,
        3981.45195965772,
        5295.45929491303,
        7084.47124360523,
        7709.20557708454,
        7436.85238642936,
        7235.49173429668,
        6710.14695517339,
        6015.71085806577,
        5279.97001316337,
```

```
        4877.24870889801,
        4421.60569340303,
        3919.49483690424,
        3498.70270322341,
        3274.10813058883,
        3147.61233529349,
        2904.94693324343,
        2805.55738101,
        2418.32535637171,
        2398.96375630723,
        2260.85100182222,
        2157.58912702878,
        2019.47637254377,
        1913.63295220712,
        1863.29279076589,
        1748.41395678279,
        1695.49224555317,
        1599.97501375715,
        1559.96103873397,
        1505.74855473274,
        1438.62833664765,
        1384.41585476901
    };

    for (size_t i = 0; i < expected_hydro_resource_vec_ms.size(); i++) {
        testFloatEquals(
            test_model_ptr->resources.resource_map_1D[hydro_resource_key][i],
            expected_hydro_resource_vec_ms[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddHydroResource_Model() */
```

5.71.2.5 testAddLiIon_Model()

```
void testAddLiIon_Model (
            Model * test_model_ptr )
```

Function to test adding a lithium ion battery energy storage system to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    LiIonInputs liion_inputs;

    test_model_ptr->addLiIon(liion_inputs);

    testFloatEquals(
        test_model_ptr->storage_ptr_vec.size(),
        1,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->storage_ptr_vec[0]->type,
        StorageType :: LIION,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddLiIon_Model() */
```

5.71.2.6 testAddSolar_Model()

```
void testAddSolar_Model (
            Model * test_model_ptr,
            int solar_resource_key )
```

Function to test adding a solar PV array to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
solar_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    SolarInputs solar_inputs;
    solar_inputs.resource_key = solar_resource_key;

    test_model_ptr->addSolar(solar_inputs);

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec.size(),
        1,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec[0]->type,
        RenewableType :: SOLAR,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddSolar_Model() */
```

5.71.2.7 testAddSolar_productionOverride_Model()

```
void testAddSolar_productionOverride_Model (
            Model * test_model_ptr,
            std::string path_2_normalized_production_time_series )
```

Function to test adding a solar PV array to the test Model object using the production override feature, and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_normalized_production_time_series	A path (either relative or absolute) to the given normalized production time series data.

```
{
    SolarInputs solar_inputs;
    solar_inputs.renewable_inputs.production_inputs.path_2_normalized_production_time_series =
        path_2_normalized_production_time_series;

    test_model_ptr->addSolar(solar_inputs);

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec.size(),
        2,
        __FILE__,
        __LINE__
    );

```

```
    testFloatEquals(
        test_model_ptr->renewable_ptr_vec[1]->type,
        RenewableType :: SOLAR,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_model_ptr->renewable_ptr_vec[1]->normalized_production_series_given,
        __FILE__,
        __LINE__
    );

    testTruth(
        test_model_ptr->renewable_ptr_vec[1]->path_2_normalized_production_time_series ==
        path_2_normalized_production_time_series,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddSolar_productionOverride_Model() */
```

5.71.2.8 testAddSolarResource_Model()

```
void testAddSolarResource_Model (
            Model * test_model_ptr,
            std::string path_2_solar_resource_data,
            int solar_resource_key )
```

Function to test adding a solar resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_solar_resource_data	A path (either relative or absolute) to the solar resource data.
solar_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    test_model_ptr->addResource(
        RenewableType :: SOLAR,
        path_2_solar_resource_data,
        solar_resource_key
    );

    std::vector<double> expected_solar_resource_vec_kWm2 = {
        0,
        0,
        0,
        0,
        0,
        0,
        8.51702662684015E-05,
        0.000348341567045,
        0.00213793728593,
        0.004099863613322,
        0.000997135230553,
        0.009534527624657,
        0.022927996790616,
        0.0136071715294,
        0.002535134127751,
        0.005206897515821,
        0.005627658648597,
        0.000701186722215,
        0.00017119827089,
        0,
        0,
        0,
        0,
        0,
        0,
```

```
        0,
        0,
        0,
        0,
        0,
        0,
        0.000141055102242,
        0.00084525014743,
        0.024893647822702,
        0.091245556190749,
        0.158722176731637,
        0.152859680515876,
        0.149922903895116,
        0.13049996570866,
        0.03081254222795,
        0.001218928911125,
        0.000206092647423,
        0,
        0,
        0,
        0,
        0,
        0
    };

    for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
        testFloatEquals(
            test_model_ptr->resources.resource_map_1D[solar_resource_key][i],
            expected_solar_resource_vec_kWm2[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddSolarResource_Model() */
```

5.71.2.9 testAddTidal_Model()

```
void testAddTidal_Model (
        Model * test_model_ptr,
        int tidal_resource_key )
```

Function to test adding a tidal turbine to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
tidal_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    TidalInputs tidal_inputs;
    tidal_inputs.resource_key = tidal_resource_key;

    test_model_ptr->addTidal(tidal_inputs);

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec.size(),
        3,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec[2]->type,
        RenewableType :: TIDAL,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddTidal_Model() */
```

5.71.2.10 testAddTidalResource_Model()

```
void testAddTidalResource_Model (
            Model * test_model_ptr,
            std::string path_2_tidal_resource_data,
            int tidal_resource_key )
```

Function to test adding a tidal resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_tidal_resource_data	A path (either relative or absolute) to the tidal resource data.
tidal_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    test_model_ptr->addResource(
        RenewableType :: TIDAL,
        path_2_tidal_resource_data,
        tidal_resource_key
    );

    std::vector<double> expected_tidal_resource_vec_ms = {
        0.347439913040533,
        0.770545522195602,
        0.731352084836198,
        0.293389814389542,
        0.209959110813115,
        0.610609623896497,
        1.78067162013604,
        2.53522775118089,
        2.75966627832024,
        2.52101111143895,
        2.05389330201031,
        1.3461515862445,
        0.28909254878384,
        0.897754086048563,
        1.71406453837407,
        1.85047408742869,
        1.71507908595979,
        1.33540349705416,
        0.434586143463003,
        0.500623815700637,
        1.37172172646733,
        1.68294125491228,
        1.56101300975417,
        1.04925834219412,
        0.211395463930223,
        1.03720048903385,
        1.85059536356448,
        1.85203242794517,
        1.4091471616277,
        0.767776539039899,
        0.251464906990961,
        1.47018469375652,
        2.36260493698197,
        2.46653750048625,
        2.12851908739291,
        1.62783753197988,
        0.734594890957439,
        0.441886297300355,
        1.6574418350918,
        2.0684558286637,
        1.87717416992136,
        1.58871262337931,
        1.03451227609235,
        0.193371305159817,
        0.976400122458815,
        1.6583227369707,
        1.76690616570953,
        1.54801328553115
    };
```

```

    for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
        testFloatEquals(
            test_model_ptr->resources.resource_map_1D[tidal_resource_key][i],
            expected_tidal_resource_vec_ms[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddTidalResource_Model() */
```

5.71.2.11 testAddWave_Model()

```
void testAddWave_Model (
            Model * test_model_ptr,
            int wave_resource_key )
```

Function to test adding a wave energy converter to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
wave_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    WaveInputs wave_inputs;
    wave_inputs.resource_key = wave_resource_key;

    test_model_ptr->addWave(wave_inputs);

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec.size(),
        4,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec[3]->type,
        RenewableType :: WAVE,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddWave_Model() */
```

5.71.2.12 testAddWaveResource_Model()

```
void testAddWaveResource_Model (
            Model * test_model_ptr,
            std::string path_2_wave_resource_data,
            int wave_resource_key )
```

Function to test adding a wave resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_wave_resource_data	A path (either relative or absolute) to the wave resource data.
wave_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    test_model_ptr->addResource(
        RenewableType :: WAVE,
        path_2_wave_resource_data,
        wave_resource_key
    );

    std::vector<double> expected_significant_wave_height_vec_m = {
        4.26175222125028,
        4.25020976167872,
        4.25656524330349,
        4.27193854786718,
        4.28744955711233,
        4.29421815278154,
        4.2839937266082,
        4.25716982457976,
        4.22419391611483,
        4.19588925217606,
        4.17338788587412,
        4.14672746914214,
        4.10560041173665,
        4.05074966447193,
        3.9953696962433,
        3.95316976150866,
        3.92771018142378,
        3.91129562488595,
        3.89558312094911,
        3.87861093931749,
        3.86538307240754,
        3.86108961027929,
        3.86459448853189,
        3.86796474016882,
        3.86357412779993,
        3.85554872014731,
        3.86044266668675,
        3.89445961915999,
        3.95554798115731,
        4.02265508610476,
        4.07419587011404,
        4.10314247143958,
        4.11738045085928,
        4.12554995596708,
        4.12923992001675,
        4.1229292327442,
        4.10123955307441,
        4.06748827895363,
        4.0336230651344,
        4.01134236393876,
        4.00136570034559,
        3.99368787690411,
        3.97820924247644,
        3.95369335178055,
        3.92742545608532,
        3.90683362771686,
        3.89331520944006,
        3.88256045801583
    };

    std::vector<double> expected_energy_period_vec_s = {
        10.4456008226821,
        10.4614151137651,
        10.4462827795433,
        10.4127692097884,
        10.3734397942723,
        10.3408599227669,
        10.32637292093,
        10.3245412676322,
        10.310409818185,
        10.2589529840966,
        10.1728100603103,
        10.0862908658929,
        10.03480243813,
        10.023673635806,
        10.0243418565116,
        10.0063487117653,
        9.96050302286607,
        9.9011999635568,
        9.84451822125472,
```

```
        9.79726875879626,
        9.75614594835158,
        9.7173447961368,
        9.68342904390577,
        9.66380508567062,
        9.6674009575699,
        9.68927134575103,
        9.70979984863046,
        9.70967357906908,
        9.68983025704562,
        9.6722855524805,
        9.67973599910003,
        9.71977125328293,
        9.78450442291421,
        9.86532355233449,
        9.96158937600019,
        10.0807018356507,
        10.2291022504937,
        10.39458528356,
        10.5464393581004,
        10.6553277500484,
        10.7245553190084,
        10.7893127285064,
        10.8846512240849,
        11.0148158739075,
        11.1544325654719,
        11.2772785848343,
        11.3744362756187,
        11.4533643503183
    };

    for (size_t i = 0; i < expected_energy_period_vec_s.size(); i++) {
        testFloatEquals(
            test_model_ptr->resources.resource_map_2D[wave_resource_key][i][0],
            expected_significant_wave_height_vec_m[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_model_ptr->resources.resource_map_2D[wave_resource_key][i][1],
            expected_energy_period_vec_s[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddWaveResource_Model() */
```

5.71.2.13 testAddWind_Model()

```
void testAddWind_Model (
            Model * test_model_ptr,
            int wind_resource_key )
```

Function to test adding a wind turbine to the test Model object and then spot check some post-add attributes.

Parameters

test_model_ptr	A pointer to the test Model object.
wind_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    WindInputs wind_inputs;
    wind_inputs.resource_key = wind_resource_key;

    test_model_ptr->addWind(wind_inputs);

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec.size(),
        5,
        __FILE__,
```

```
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->renewable_ptr_vec[4]->type,
        RenewableType :: WIND,
        __FILE__,
        __LINE__
    );

    return;
}   /* testAddWind_Model() */
```

5.71.2.14 testAddWindResource_Model()

```
void testAddWindResource_Model (
            Model * test_model_ptr,
            std::string path_2_wind_resource_data,
            int wind_resource_key )
```

Function to test adding a wind resource and then check the values read into the Resources component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.
path_2_wind_resource_data	A path (either relative or absolute) to the wind resource data.
wind_resource_key	A key used to index into the Resources component of the test Model object.

```
{
    test_model_ptr->addResource(
        RenewableType :: WIND,
        path_2_wind_resource_data,
        wind_resource_key
    );

    std::vector<double> expected_wind_resource_vec_ms = {
        6.88566688469997,
        5.02177105466549,
        3.74211715899568,
        5.67169579985362,
        4.90670669971858,
        4.29586955031368,
        7.41155377205065,
        10.2243290476943,
        13.1258696725555,
        13.7016198628274,
        16.2481482330233,
        16.5096744355418,
        13.4354482206162,
        14.0129230731609,
        14.5554549260515,
        13.4454539065912,
        13.3447169512094,
        11.7372615098554,
        12.7200070078013,
        10.6421127908149,
        6.09869498990661,
        5.66355596602321,
        4.97316966910831,
        3.48937138360567,
        2.15917470979169,
        1.29061103587027,
        3.43475751425219,
        4.11706326260927,
        4.28905275747408,
        5.75850263196241,
        8.98293663055264,
        11.7069822941315,
        12.4031987075858,
        15.4096570910089,
        16.6210843829552,
```

```
        13.3421219142573,
        15.2112831900548,
        18.350864533037,
        15.8751799822971,
        15.3921198799796,
        15.9729192868434,
        12.4728950178772,
        10.177050481096,
        10.7342247355551,
        8.98846695631389,
        4.14671169124739,
        3.17256452697149,
        3.40036336968628
    };

    for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
        testFloatEquals(
            test_model_ptr->resources.resource_map_1D[wind_resource_key][i],
            expected_wind_resource_vec_ms[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddWindResource_Model() */
```

5.71.2.15 testBadConstruct_Model()

```
void testBadConstruct_Model (
            void )
```

Function to check if passing bad ModelInputs to the Model constructor is handled appropriately.

```
{
    bool error_flag = true;

    try {
        ModelInputs bad_model_inputs;   // path_2_electrical_load_time_series left empty

        Model bad_model(bad_model_inputs);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    try {
        ModelInputs bad_model_inputs;
        bad_model_inputs.path_2_electrical_load_time_series =
            "data/test/electrical_load/bad_path_";
        bad_model_inputs.path_2_electrical_load_time_series += std::to_string(rand());
        bad_model_inputs.path_2_electrical_load_time_series += ".csv";

        Model bad_model(bad_model_inputs);

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
}
```

5.71.2.16 testConstruct_Model()

```
Model* testConstruct_Model (
            ModelInputs test_model_inputs )
```

```
{
    Model* test_model_ptr = new Model(test_model_inputs);

    testTruth(
        test_model_ptr->electrical_load.path_2_electrical_load_time_series ==
        test_model_inputs.path_2_electrical_load_time_series,
        __FILE__,
        __LINE__
    );

    return test_model_ptr;
}   /* testConstruct_Model() */
```

5.71.2.17 testEconomics_Model()

```
void testEconomics_Model (
            Model * test_model_ptr )
```

Function to check that the modelled economic metrics are $>$ 0.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    testGreaterThan(
        test_model_ptr->net_present_cost,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->levellized_cost_of_energy_kWh,
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testEconomics_Model() */
```

5.71.2.18 testElectricalLoadData_Model()

```
void testElectricalLoadData_Model (
            Model * test_model_ptr )
```

Function to check the values read into the ElectricalLoad component of the test Model object.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    std::vector<double> expected_dt_vec_hrs (48, 1);

```

```
    std::vector<double> expected_time_vec_hrs = {
         0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
        12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
        24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
    };

    std::vector<double> expected_load_vec_kW = {
        360.253836463674,
        355.171277826775,
        353.776453532298,
        353.75405737934,
        346.592867404975,
        340.132411175118,
        337.354867340578,
        340.644115618736,
        363.639028500678,
        378.787797779238,
        372.215798201712,
        395.093925731298,
        402.325427142659,
        386.907725462306,
        380.709170928091,
        372.062070914977,
        372.328646856954,
        391.841444284136,
        394.029351759596,
        383.369407765254,
        381.093099675206,
        382.604158946193,
        390.744843709034,
        383.13949492437,
        368.150393976985,
        364.629744480226,
        363.572736804082,
        359.854924202248,
        355.207590170267,
        349.094656012401,
        354.365935871597,
        343.380608328546,
        404.673065729266,
        486.296896820126,
        480.225974100847,
        457.318764401085,
        418.177339948609,
        414.399018364126,
        409.678420185754,
        404.768766016563,
        401.699589920585,
        402.44339040654,
        398.138372541906,
        396.010498627646,
        390.165117432277,
        375.850429417013,
        365.567100746484,
        365.429624610923
    };

    for (int i = 0; i < 48; i++) {
        testFloatEquals(
            test_model_ptr->electrical_load.dt_vec_hrs[i],
            expected_dt_vec_hrs[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_model_ptr->electrical_load.time_vec_hrs[i],
            expected_time_vec_hrs[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_model_ptr->electrical_load.load_vec_kW[i],
            expected_load_vec_kW[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testElectricalLoadData_Model() */
```

5.71.2.19 testFuelConsumptionEmissions_Model()

```
void testFuelConsumptionEmissions_Model (
            Model * test_model_ptr )
```

Function to check that the modelled fuel consumption and emissions are $>$ 0.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    testGreaterThan(
        test_model_ptr->total_fuel_consumed_L,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.CO2_kg,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.CO_kg,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.NOx_kg,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.SOx_kg,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.CH4_kg,
        0,
        __FILE__,
        __LINE__
    );

    testGreaterThan(
        test_model_ptr->total_emissions.PM_kg,
        0,
        __FILE__,
        __LINE__
    );

    return;
}   /* testFuelConsumptionEmissions_Model() */
```

5.71.2.20 testLoadBalance_Model()

```
void testLoadBalance_Model (
            Model * test_model_ptr )
```

Function to check that the post-run load data is as expected. That is, the added renewable, production, and storage assets are handled by the Controller as expected.

Parameters

test_model_ptr	A pointer to the test Model object.

```
{
    double net_load_kW = 0;

    Combustion* combustion_ptr;
    Noncombustion* noncombustion_ptr;
    Renewable* renewable_ptr;
    Storage* storage_ptr;

    for (int i = 0; i < test_model_ptr->electrical_load.n_points; i++) {
        net_load_kW = test_model_ptr->controller.net_load_vec_kW[i];

        testLessThanOrEqualTo(
            test_model_ptr->controller.net_load_vec_kW[i],
            test_model_ptr->electrical_load.max_load_kW,
            __FILE__,
            __LINE__
        );

        for (size_t j = 0; j < test_model_ptr->combustion_ptr_vec.size(); j++) {
            combustion_ptr = test_model_ptr->combustion_ptr_vec[j];

            testFloatEquals(
                combustion_ptr->production_vec_kW[i] -
                combustion_ptr->dispatch_vec_kW[i] -
                combustion_ptr->curtailment_vec_kW[i] -
                combustion_ptr->storage_vec_kW[i],
                0,
                __FILE__,
                __LINE__
            );

            net_load_kW -= combustion_ptr->production_vec_kW[i];
        }

        for (size_t j = 0; j < test_model_ptr->noncombustion_ptr_vec.size(); j++) {
            noncombustion_ptr = test_model_ptr->noncombustion_ptr_vec[j];

            testFloatEquals(
                noncombustion_ptr->production_vec_kW[i] -
                noncombustion_ptr->dispatch_vec_kW[i] -
                noncombustion_ptr->curtailment_vec_kW[i] -
                noncombustion_ptr->storage_vec_kW[i],
                0,
                __FILE__,
                __LINE__
            );

            net_load_kW -= noncombustion_ptr->production_vec_kW[i];
        }

        for (size_t j = 0; j < test_model_ptr->renewable_ptr_vec.size(); j++) {
            renewable_ptr = test_model_ptr->renewable_ptr_vec[j];

            testFloatEquals(
                renewable_ptr->production_vec_kW[i] -
                renewable_ptr->dispatch_vec_kW[i] -
                renewable_ptr->curtailment_vec_kW[i] -
                renewable_ptr->storage_vec_kW[i],
                0,
                __FILE__,
                __LINE__
            );

            net_load_kW -= renewable_ptr->production_vec_kW[i];
        }

        for (size_t j = 0; j < test_model_ptr->storage_ptr_vec.size(); j++) {
            storage_ptr = test_model_ptr->storage_ptr_vec[j];

            testTruth(
                not (
                    storage_ptr->charging_power_vec_kW[i] > 0 and
                    storage_ptr->discharging_power_vec_kW[i] > 0
                ),
                __FILE__,
                __LINE__
            );

            net_load_kW -= storage_ptr->discharging_power_vec_kW[i];
        }

        testLessThanOrEqualTo(
```

```
            net_load_kW,
            0,
            __FILE__,
            __LINE__
        );
    }

    testFloatEquals(
        test_model_ptr->total_dispatch_discharge_kWh,
        2263351.62026685,
        __FILE__,
        __LINE__
    );

    return;
}   /* testLoadBalance_Model() */
```

5.71.2.21 testPostConstructionAttributes_Model()

```
void testPostConstructionAttributes_Model (
            Model * test_model_ptr )
```

A function to check the values of various post-construction attributes.

Parameters

test_model_ptr	A pointer to the test Model object.

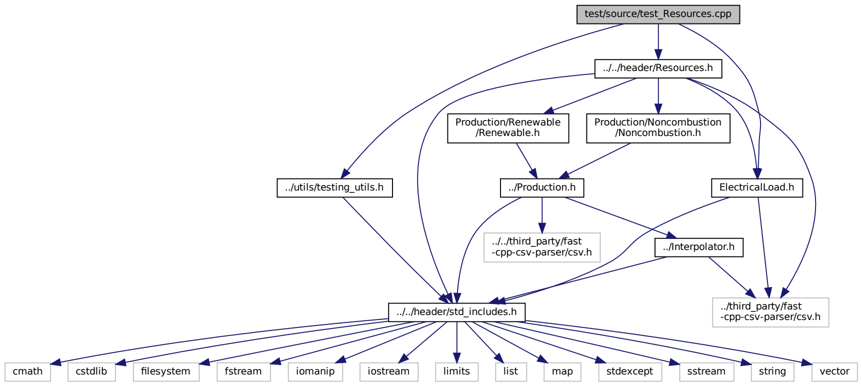
```
{
    testFloatEquals(
        test_model_ptr->electrical_load.n_points,
        8760,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->electrical_load.n_years,
        0.999886,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->electrical_load.min_load_kW,
        82.1211213927802,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_model_ptr->electrical_load.mean_load_kW,
        258.373472633202,
        __FILE__,
        __LINE__
    );


    testFloatEquals(
        test_model_ptr->electrical_load.max_load_kW,
        500,
        __FILE__,
        __LINE__
    );

    return;
}   /* testPostConstructionAttributes_Model() */
```

5.72 test/source/test_Resources.cpp File Reference

Testing suite for Resources class.

```
#include "../utils/testing_utils.h"
#include "../../header/Resources.h"
#include "../../header/ElectricalLoad.h"
```

Include dependency graph for test_Resources.cpp:



Functions

- Resources ∗ testConstruct_Resources (void)

    A function to construct a Resources object and spot check some post-construction attributes.
- void testAddSolarResource_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

    Function to test adding a solar resource and then check the values read into the test Resources object.
- void testBadAdd_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_solar_resource_data, int solar_resource_key)

    Function to test that trying to add bad resource data is being handled as expected.
- void testAddTidalResource_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_tidal_resource_data, int tidal_resource_key)

    Function to test adding a tidal resource and then check the values read into the test Resources object.
- void testAddWaveResource_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_wave_resource_data, int wave_resource_key)

    Function to test adding a wave resource and then check the values read into the test Resources object.
- void testAddWindResource_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_wind_resource_data, int wind_resource_key)

    Function to test adding a wind resource and then check the values read into the test Resources object.
- void testAddHydroResource_Resources (Resources ∗test_resources_ptr, ElectricalLoad ∗test_electrical_load_ptr, std::string path_2_hydro_resource_data, int hydro_resource_key)

    Function to test adding a hydro resource and then check the values read into the test Resources object.
- int main (int argc, char ∗∗argv)

5.72.1 Detailed Description

Testing suite for Resources class.

A suite of tests for the Resources class.

5.72.2 Function Documentation

5.72.2.1 main()

```
int main (
            int argc,
            char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif  /* _WIN32 */

    printGold("\tTesting Resources");

    srand(time(NULL));


    std::string path_2_electrical_load_time_series =
        "data/test/electrical_load/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";

    ElectricalLoad* test_electrical_load_ptr =
        new ElectricalLoad(path_2_electrical_load_time_series);

    Resources* test_resources_ptr = testConstruct_Resources();


    try {
        int solar_resource_key = 0;
        std::string path_2_solar_resource_data =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";

        testAddSolarResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_solar_resource_data,
            solar_resource_key
        );

        testBadAdd_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_solar_resource_data,
            solar_resource_key
        );


        int tidal_resource_key = 1;
        std::string path_2_tidal_resource_data =
            "data/test/resources/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

        testAddTidalResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_tidal_resource_data,
            tidal_resource_key
        );


        int wave_resource_key = 2;
        std::string path_2_wave_resource_data =
            "data/test/resources/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";

        testAddWaveResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_wave_resource_data,
            wave_resource_key
        );


        int wind_resource_key = 3;
        std::string path_2_wind_resource_data =
            "data/test/resources/wind_speed_peak-25ms_1yr_dt-1hr.csv";

        testAddWindResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_wind_resource_data,
```

```
            wind_resource_key
        );


        int hydro_resource_key = 4;
        std::string path_2_hydro_resource_data =
            "data/test/resources/hydro_inflow_peak-20000m3hr_1yr_dt-1hr.csv";

        testAddHydroResource_Resources(
            test_resources_ptr,
            test_electrical_load_ptr,
            path_2_hydro_resource_data,
            hydro_resource_key
        );
    }


    catch (...) {
        delete test_electrical_load_ptr;
        delete test_resources_ptr;

        printGold(" ............................... ");
        printRed("FAIL");
        std::cout « std::endl;
        throw;
    }


    delete test_electrical_load_ptr;
    delete test_resources_ptr;

    printGold(" ............................... ");
    printGreen("PASS");
    std::cout « std::endl;
    return 0;
}   /* main() */
```

5.72.2.2 testAddHydroResource_Resources()

```
void testAddHydroResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_hydro_resource_data,
            int hydro_resource_key )
```

Function to test adding a hydro resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_hydro_resource_data	A path (either relative or absolute) to the hydro resource data.
hydro_resource_key	A key used to index into the Resources component of the test Resources object.

```
{
    test_resources_ptr->addResource(
        NoncombustionType::HYDRO,
        path_2_hydro_resource_data,
        hydro_resource_key,
        test_electrical_load_ptr
    );

    std::vector<double> expected_hydro_resource_vec_m3hr = {
        2167.91531556942,
        2046.58261560569,
        2007.85941123153,
        2000.11477247929,
        1917.50527264453,
        1963.97311577093,
        1908.46985899809,
        1886.5267112678,
```

```
        1965.26388854254,
        1953.64692935289,
        2084.01504296306,
        2272.46796101188,
        2520.29645627096,
        2715.203242423,
        2720.36633563203,
        3130.83228077221,
        3289.59741021591,
        3981.45195965772,
        5295.45929491303,
        7084.47124360523,
        7709.20557708454,
        7436.85238642936,
        7235.49173429668,
        6710.14695517339,
        6015.71085806577,
        5279.97001316337,
        4877.24870889801,
        4421.60569340303,
        3919.49483690424,
        3498.70270322341,
        3274.10813058883,
        3147.61233529349,
        2904.94693324343,
        2805.55738101,
        2418.32535637171,
        2398.96375630723,
        2260.85100182222,
        2157.58912702878,
        2019.47637254377,
        1913.63295220712,
        1863.29279076589,
        1748.41395678279,
        1695.49224555317,
        1599.97501375715,
        1559.96103873397,
        1505.74855473274,
        1438.62833664765,
        1384.41585476901
    };

    for (size_t i = 0; i < expected_hydro_resource_vec_m3hr.size(); i++) {
        testFloatEquals(
            test_resources_ptr->resource_map_1D[hydro_resource_key][i],
            expected_hydro_resource_vec_m3hr[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddHydroResource_Resources() */
```

5.72.2.3 testAddSolarResource_Resources()

```
void testAddSolarResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_solar_resource_data,
            int solar_resource_key )
```

Function to test adding a solar resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_solar_resource_data	A path (either relative or absolute) to the solar resource data.
solar_resource_key	A key used to index into the Resources component of the test Resources object.

```
{
    test_resources_ptr->addResource(
        RenewableType::SOLAR,
        path_2_solar_resource_data,
        solar_resource_key,
        test_electrical_load_ptr
    );

    std::vector<double> expected_solar_resource_vec_kWm2 = {
        0,
        0,
        0,
        0,
        0,
        0,
        8.51702662684015E-05,
        0.000348341567045,
        0.00213793728593,
        0.004099863613322,
        0.000997135230553,
        0.009534527624657,
        0.022927996790616,
        0.0136071715294,
        0.002535134127751,
        0.005206897515821,
        0.005627658648597,
        0.000701186722215,
        0.00017119827089,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0.000141055102242,
        0.00084525014743,
        0.024893647822702,
        0.091245556190749,
        0.158722176731637,
        0.152859680515876,
        0.149922903895116,
        0.13049996570866,
        0.03081254222795,
        0.001218928911125,
        0.000206092647423,
        0,
        0,
        0,
        0,
        0,
        0
    };

    for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
        testFloatEquals(
            test_resources_ptr->resource_map_1D[solar_resource_key][i],
            expected_solar_resource_vec_kWm2[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddSolarResource_Resources() */
```

5.72.2.4 testAddTidalResource_Resources()

```
void testAddTidalResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_tidal_resource_data,
            int tidal_resource_key )
```

Function to test adding a tidal resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_tidal_resource_data	A path (either relative or absolute) to the tidal resource data.
tidal_resource_key	A key used to index into the Resources component of the test Resources object.

```
{
    test_resources_ptr->addResource(
        RenewableType::TIDAL,
        path_2_tidal_resource_data,
        tidal_resource_key,
        test_electrical_load_ptr
    );

    std::vector<double> expected_tidal_resource_vec_ms = {
        0.347439913040533,
        0.770545522195602,
        0.731352084836198,
        0.293389814389542,
        0.209959110813115,
        0.610609623896497,
        1.78067162013604,
        2.53522775118089,
        2.75966627832024,
        2.52101111143895,
        2.05389330201031,
        1.3461515862445,
        0.28909254878384,
        0.897754086048563,
        1.71406453837407,
        1.85047408742869,
        1.71507908595979,
        1.33540349705416,
        0.434586143463003,
        0.500623815700637,
        1.37172172646733,
        1.68294125491228,
        1.56101300975417,
        1.04925834219412,
        0.211395463930223,
        1.03720048903385,
        1.85059536356448,
        1.85203242794517,
        1.4091471616277,
        0.767776539039899,
        0.251464906990961,
        1.47018469375652,
        2.36260493698197,
        2.46653750048625,
        2.12851908739291,
        1.62783753197988,
        0.734594890957439,
        0.441886297300355,
        1.6574418350918,
        2.0684558286637,
        1.87717416992136,
        1.58871262337931,
        1.03451227609235,
        0.193371305159817,
        0.976400122458815,
        1.6583227369707,
        1.76690616570953,
        1.54801328553115
    };

    for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
        testFloatEquals(
            test_resources_ptr->resource_map_1D[tidal_resource_key][i],
            expected_tidal_resource_vec_ms[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}    /* testAddTidalResource_Resources() */
```

5.72.2.5 testAddWaveResource_Resources()

```
void testAddWaveResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_wave_resource_data,
            int wave_resource_key )
```

Function to test adding a wave resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_wave_resource_data	A path (either relative or absolute) to the wave resource data.
wave_resource_key	A key used to index into the Resources component of the test Resources object.

```
{
    test_resources_ptr->addResource(
        RenewableType::WAVE,
        path_2_wave_resource_data,
        wave_resource_key,
        test_electrical_load_ptr
    );

    std::vector<double> expected_significant_wave_height_vec_m = {
        4.26175222125028,
        4.25020976167872,
        4.25656524330349,
        4.27193854786718,
        4.28744955711233,
        4.29421815278154,
        4.2839937266082,
        4.25716982457976,
        4.22419391611483,
        4.19588925217606,
        4.17338788587412,
        4.14672746914214,
        4.10560041173665,
        4.05074966447193,
        3.9953696962433,
        3.95316976150866,
        3.92771018142378,
        3.91129562488595,
        3.89558312094911,
        3.87861093931749,
        3.86538307240754,
        3.86108961027929,
        3.86459448853189,
        3.86796474016882,
        3.86357412779993,
        3.85554872014731,
        3.86044266668675,
        3.89445961915999,
        3.95554798115731,
        4.02265508610476,
        4.07419587011404,
        4.10314247143958,
        4.11738045085928,
        4.12554995596708,
        4.12923992001675,
        4.1229292327442,
        4.10123955307441,
        4.06748827895363,
        4.0336230651344,
        4.01134236393876,
        4.00136570034559,
        3.99368787690411,
        3.97820924247644,
        3.95369335178055,
        3.92742545608532,
        3.90683362771686,
        3.89331520944006,
        3.88256045801583
    };

    std::vector<double> expected_energy_period_vec_s = {
```

```
        10.4456008226821,
        10.4614151137651,
        10.4462827795433,
        10.4127692097884,
        10.3734397942723,
        10.3408599227669,
        10.32637292093,
        10.3245412676322,
        10.310409818185,
        10.2589529840966,
        10.1728100603103,
        10.0862908658929,
        10.03480243813,
        10.023673635806,
        10.0243418565116,
        10.0063487117653,
        9.96050302286607,
        9.9011999635568,
        9.84451822125472,
        9.79726875879626,
        9.75614594835158,
        9.7173447961368,
        9.68342904390577,
        9.66380508567062,
        9.6674009575699,
        9.68927134575103,
        9.70979984863046,
        9.70967357906908,
        9.68983025704562,
        9.6722855524805,
        9.67973599910003,
        9.71977125328293,
        9.78450442291421,
        9.86532355233449,
        9.96158937600019,
        10.0807018356507,
        10.2291022504937,
        10.39458528356,
        10.5464393581004,
        10.6553277500484,
        10.7245553190084,
        10.7893127285064,
        10.8846512240849,
        11.0148158739075,
        11.1544325654719,
        11.2772785848343,
        11.3744362756187,
        11.4533643503183
    };

    for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
        testFloatEquals(
            test_resources_ptr->resource_map_2D[wave_resource_key][i][0],
            expected_significant_wave_height_vec_m[i],
            __FILE__,
            __LINE__
        );

        testFloatEquals(
            test_resources_ptr->resource_map_2D[wave_resource_key][i][1],
            expected_energy_period_vec_s[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddWaveResource_Resources() */
```

5.72.2.6 testAddWindResource_Resources()

```
void testAddWindResource_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_wind_resource_data,
            int wind_resource_key )
```

Function to test adding a wind resource and then check the values read into the test Resources object.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_wind_resource_data	A path (either relative or absolute) to the wind resource data.
wind_resource_key	A key used to index into the Resources component of the test Resources object.

```
{
    test_resources_ptr->addResource(
        RenewableType::WIND,
        path_2_wind_resource_data,
        wind_resource_key,
        test_electrical_load_ptr
    );

    std::vector<double> expected_wind_resource_vec_ms = {
        6.88566688469997,
        5.02177105466549,
        3.74211715899568,
        5.67169579985362,
        4.90670669971858,
        4.29586955031368,
        7.41155377205065,
        10.2243290476943,
        13.1258696725555,
        13.7016198628274,
        16.2481482330233,
        16.5096744355418,
        13.4354482206162,
        14.0129230731609,
        14.5554549260515,
        13.4454539065912,
        13.3447169512094,
        11.7372615098554,
        12.7200070078013,
        10.6421127908149,
        6.09869498990661,
        5.66355596602321,
        4.97316966910831,
        3.48937138360567,
        2.15917470979169,
        1.29061103587027,
        3.43475751425219,
        4.11706326260927,
        4.28905275747408,
        5.75850263196241,
        8.98293663055264,
        11.7069822941315,
        12.4031987075858,
        15.4096570910089,
        16.6210843829552,
        13.3421219142573,
        15.2112831900548,
        18.350864533037,
        15.8751799822971,
        15.3921198799796,
        15.9729192868434,
        12.4728950178772,
        10.177050481096,
        10.7342247355551,
        8.98846695631389,
        4.14671169124739,
        3.17256452697149,
        3.40036336968628
    };

    for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
        testFloatEquals(
            test_resources_ptr->resource_map_1D[wind_resource_key][i],
            expected_wind_resource_vec_ms[i],
            __FILE__,
            __LINE__
        );
    }

    return;
}   /* testAddWindResource_Resources() */
```

5.72.2.7 testBadAdd_Resources()

```
void testBadAdd_Resources (
            Resources * test_resources_ptr,
            ElectricalLoad * test_electrical_load_ptr,
            std::string path_2_solar_resource_data,
            int solar_resource_key )
```

Function to test that trying to add bad resource data is being handled as expected.

Parameters

test_resources_ptr	A pointer to the test Resources object.
test_electrical_load_ptr	A pointer to the test ElectricalLoad object.
path_2_solar_resource_data	A path (either relative or absolute) to the given solar resource data.
solar_resource_key	A key for indexing into the test Resources object.

```
{
    bool error_flag = true;

    try {
        test_resources_ptr->addResource(
            RenewableType::SOLAR,
            path_2_solar_resource_data,
            solar_resource_key,
            test_electrical_load_ptr
        );

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }


    try {
        std::string path_2_solar_resource_data_BAD_TIMES =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_TIMES.csv";

        test_resources_ptr->addResource(
            RenewableType::SOLAR,
            path_2_solar_resource_data_BAD_TIMES,
            -1,
            test_electrical_load_ptr
        );

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }


    try {
        std::string path_2_solar_resource_data_BAD_LENGTH =
            "data/test/resources/solar_GHI_peak-1kWm2_1yr_dt-1hr_BAD_LENGTH.csv";

        test_resources_ptr->addResource(
            RenewableType::SOLAR,
            path_2_solar_resource_data_BAD_LENGTH,
            -2,
            test_electrical_load_ptr
        );

        error_flag = false;
    } catch (...) {
        // Task failed successfully! =P
    }
    if (not error_flag) {
        expectedErrorNotDetected(__FILE__, __LINE__);
    }

    return;
```

```
}    /* testBadAdd_Resources() */
```

5.72.2.8 testConstruct_Resources()

```
Resources * testConstruct_Resources (
        void )
```

A function to construct a Resources object and spot check some post-construction attributes.

Returns

A pointer to a test Resources object.

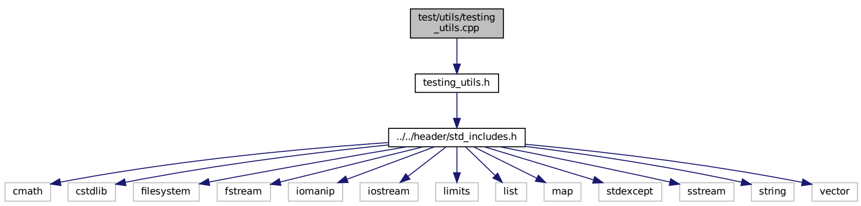
```
{
    Resources* test_resources_ptr = new Resources();

    testFloatEquals(
        test_resources_ptr->resource_map_1D.size(),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_resources_ptr->path_map_1D.size(),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_resources_ptr->resource_map_2D.size(),
        0,
        __FILE__,
        __LINE__
    );

    testFloatEquals(
        test_resources_ptr->path_map_2D.size(),
        0,
        __FILE__,
        __LINE__
    );

    return test_resources_ptr;
}   /* testConstruct_Resources() */
```

5.73 test/utils/testing_utils.cpp File Reference

Implementation file for various PGMcpp testing utilities.

```
#include "testing_utils.h"
```

Include dependency graph for testing_utils.cpp:



Functions

- void printGreen (std::string input_str)

  A function that sends green text to std::cout.
- void printGold (std::string input_str)

  A function that sends gold text to std::cout.
- void printRed (std::string input_str)

  A function that sends red text to std::cout.
- void testFloatEquals (double x, double y, std::string file, int line)

  Tests for the equality of two floating point numbers x and y (to within FLOAT_TOLERANCE).
- void testGreaterThan (double x, double y, std::string file, int line)

  Tests if $x > y$.
- void testGreaterThanOrEqualTo (double x, double y, std::string file, int line)

  Tests if $x >= y$.
- 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 $x <= y$.
- void testTruth (bool statement, std::string file, int line)

  Tests if the given statement is true.
- void expectedErrorNotDetected (std::string file, int line)

  A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.73.1 Detailed Description

Implementation file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.73.2 Function Documentation

5.73.2.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
            std::string file,
            int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
    error_str += std::to_string(line);
```

```
    error_str += " of ";
    error_str += file;

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* expectedErrorNotDetected() */
```

5.73.2.2 printGold()

void printGold (
    std::string *input_str*)

A function that sends gold text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[33m" « input_str « "\033[0m";
    return;
}   /* printGold() */
```

5.73.2.3 printGreen()

void printGreen (
    std::string *input_str*)

A function that sends green text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[32m" « input_str « "\033[0m";
    return;
}   /* printGreen() */
```

5.73.2.4 printRed()

void printRed (
    std::string *input_str*)

A function that sends red text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[31m" « input_str « "\033[0m";
    return;
}   /* printRed() */
```

5.73.2.5 testFloatEquals()

```
void testFloatEquals (
            double x,
            double y,
            std::string file,
            int line )
```

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT_TOLERANCE).

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (fabs(x - y) <= FLOAT_TOLERANCE) {
        return;
    }

    std::string error_str = "ERROR: testFloatEquals():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " and ";
    error_str += std::to_string(y);
    error_str += " are not equal to within +/- ";
    error_str += std::to_string(FLOAT_TOLERANCE);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testFloatEquals() */
```

5.73.2.6 testGreaterThan()

```
void testGreaterThan (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x > y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x > y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testGreaterThan() */
```

5.73.2.7 testGreaterThanOrEqualTo()

```
void testGreaterThanOrEqualTo (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x >= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x >= y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
```

```
    return;
}   /* testGreaterThanOrEqualTo() */
```

5.73.2.8 testLessThan()

```
void testLessThan (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x < y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x < y) {
        return;
    }

    std::string error_str = "ERROR: testLessThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testLessThan() */
```

5.73.2.9 testLessThanOrEqualTo()

```
void testLessThanOrEqualTo (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x <= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x <= y) {
        return;
    }

    std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testLessThanOrEqualTo() */
```

5.73.2.10 testTruth()

```
void testTruth (
            bool statement,
            std::string file,
            int line )
```

Tests if the given statement is true.

Parameters

statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

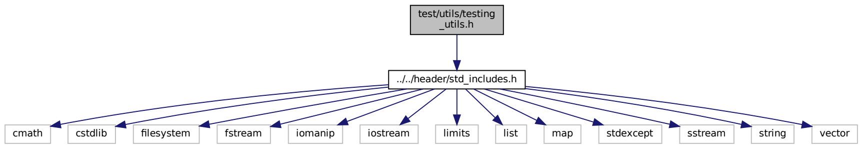
```
{
    if (statement) {
        return;
    }

    std::string error_str = "ERROR: testTruth():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += "Given statement is not true";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testTruth() */
```

5.74 test/utils/testing_utils.h File Reference

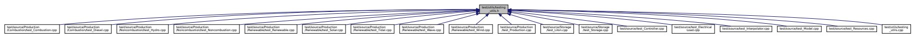
Header file for various PGMcpp testing utilities.

`#include "../../header/std_includes.h"`

Include dependency graph for testing_utils.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define FLOAT_TOLERANCE 1e-6

  A tolerance for application to floating point equality tests.

Functions

- void printGreen (std::string)

  A function that sends green text to std::cout.
- void printGold (std::string)

  A function that sends gold text to std::cout.
- void printRed (std::string)

  A function that sends red text to std::cout.
- void testFloatEquals (double, double, std::string, int)

  Tests for the equality of two floating point numbers x and y (to within FLOAT_TOLERANCE).
- void testGreaterThan (double, double, std::string, int)

  Tests if x > y.
- void testGreaterThanOrEqualTo (double, double, std::string, int)

  Tests if x >= y.
- void testLessThan (double, double, std::string, int)

  Tests if x < y.
- void testLessThanOrEqualTo (double, double, std::string, int)

  Tests if x <= y.
- void testTruth (bool, std::string, int)

  Tests if the given statement is true.
- void expectedErrorNotDetected (std::string, int)

  A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

5.74.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.74.2 Macro Definition Documentation

5.74.2.1 FLOAT_TOLERANCE

```
#define FLOAT_TOLERANCE 1e-6
```

A tolerance for application to floating point equality tests.

5.74.3 Function Documentation

5.74.3.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
            std::string file,
            int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    std::string error_str = "\n ERROR  failed to throw expected error prior to line ";
    error_str += std::to_string(line);
    error_str += " of ";
    error_str += file;

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* expectedErrorNotDetected() */
```

5.74.3.2 printGold()

```
void printGold (
            std::string input_str )
```

A function that sends gold text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[33m" « input_str « "\033[0m";
    return;
}   /* printGold() */
```

5.74.3.3 printGreen()

```
void printGreen (
            std::string input_str )
```

A function that sends green text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[32m" « input_str « "\033[0m";
    return;
}   /* printGreen() */
```

5.74.3.4 printRed()

```
void printRed (
            std::string input_str )
```

A function that sends red text to std::cout.

Parameters

input_str	The text of the string to be sent to std::cout.

```
{
    std::cout « "\x1B[31m" « input_str « "\033[0m";
    return;
}   /* printRed() */
```

5.74.3.5 testFloatEquals()

```
void testFloatEquals (
            double x,
            double y,
            std::string file,
            int line )
```

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT_TOLERANCE).

Parameters

x	The first of two numbers to test.

Parameters

y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (fabs(x - y) <= FLOAT_TOLERANCE) {
        return;
    }

    std::string error_str = "ERROR: testFloatEquals():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " and ";
    error_str += std::to_string(y);
    error_str += " are not equal to within +/- ";
    error_str += std::to_string(FLOAT_TOLERANCE);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testFloatEquals() */
```

5.74.3.6 testGreaterThan()

```
void testGreaterThan (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x > y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x > y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

```

```
    throw std::runtime_error(error_str);
    return;
}   /* testGreaterThan() */
```

5.74.3.7 testGreaterThanOrEqualTo()

```
void testGreaterThanOrEqualTo (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x >= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x >= y) {
        return;
    }

    std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not greater than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testGreaterThanOrEqualTo() */
```

5.74.3.8 testLessThan()

```
void testLessThan (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x < y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x < y) {
        return;
    }

    std::string error_str = "ERROR: testLessThan():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testLessThan() */
```

5.74.3.9 testLessThanOrEqualTo()

```
void testLessThanOrEqualTo (
            double x,
            double y,
            std::string file,
            int line )
```

Tests if x <= y.

Parameters

x	The first of two numbers to test.
y	The second of two numbers to test.
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (x <= y) {
        return;
    }

    std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += std::to_string(x);
    error_str += " is not less than or equal to ";
    error_str += std::to_string(y);
    error_str += "\n";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* testLessThanOrEqualTo() */
```

5.74.3.10 testTruth()

```
void testTruth (
```

```
bool statement,
std::string file,
int line )
```

Tests if the given statement is true.

Parameters

statement	The statement whose truth is to be tested ("1 == 0", for example).
file	The file in which the test is applied (you should be able to just pass in "__FILE__").
line	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
{
    if (statement) {
        return;
    }

    std::string error_str = "ERROR: testTruth():\t in ";
    error_str += file;
    error_str += "\tline ";
    error_str += std::to_string(line);
    error_str += ":\t\n";
    error_str += "Given statement is not true";

    #ifdef _WIN32
        std::cout « error_str « std::endl;
    #endif

    throw std::runtime_error(error_str);
    return;
}   /* 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. 233

CIMAC. Guide to Diesel Exhaust Emissions Control of NOx, SOx, Particulates, Smoke, and CO2. Technical report, Conseil International des Machines à Combustion, 2008. Included: docs/refs/diesel_emissions_ref_2.pdf. 60

HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. 171, 220

HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. 16, 158, 171, 172, 219, 220

HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. 51, 60

HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. 51, 60

HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. 51, 60

HOMER. How HOMER Calculates the PV Array Power Output, 2023f. URL https://www.homerenergy.com/products/pro/docs/latest/how_homer_calculates_the_pv_array_power_output.html. 210

HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. 171, 220

HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. 172, 219

HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. 171, 220

W. Jakob. pybind11 — Seamless operability between C++11 and Python, 2023. URL https://pybind11.readthedocs.io/en/stable/. 303, 304, 306, 309, 312, 314, 319, 320, 322, 324, 326, 328, 330, 331, 333, 334, 336, 341

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. 235, 236, 251

P. Milan, M. Wächter, S. Barth, and J. Peinke. Power curves for wind turbines. *Wind Power Generation and Wind Turbine Design*, page 595–612, 2010. doi: 10.2495/978-1-84564-205-1/18. 264

NRCan. Auto$mart Learn the facts: Emissions from your vehicle. Technical report, Natural Resources Canada, 2014. Included: docs/refs/diesel_emissions_ref_1.pdf. 60

Dr. B. Robertson, Dr. H. Bailey, M. Leary, and Dr. B. Buckham. A methodology for architecture agnostic and time flexible representations of wave energy converter performance. *Applied Energy*, 287, 2021. doi:10.1016/j.apenergy.2021.116588. 250

A. Truelove. Battery Degradation Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023a. Included: docs/refs/battery_degradation.pdf. 115, 116, 118, 129

A. Truelove. Hydro Modelling For Implementation in PGMcpp. Technical report, PRIMED, 2023b. Included: docs/refs/hydro.pdf. 75, 77, 78, 79, 81

A. Truelove, Dr. B. Buckham, Dr. C. Crawford, and C. Hiles. Scaling Technology Models for HOMER Pro: Wind, Tidal Stream, and Wave. Technical report, PRIMED, 2019. Included: docs/refs/wind_tidal_wave.pdf. 234, 249, 265

D. van Heesch. Doxygen: Generate documentation from source code, 2023. URL https://www.doxygen.nl. 276

Index

__applyCycleChargingControl_CHARGING
Controller, 27
__applyCycleChargingControl_DISCHARGING
Controller, 28
__applyLoadFollowingControl_CHARGING
Controller, 29
__applyLoadFollowingControl_DISCHARGING
Controller, 30
__checkBounds1D
Interpolator, 93
__checkBounds2D
Interpolator, 94
__checkDataKey1D
Interpolator, 95
__checkDataKey2D
Interpolator, 95
__checkInputs
Combustion, 14
Diesel, 48
Hydro, 74
LiIon, 113
Model, 136
Noncombustion, 156
Production, 166
Renewable, 183
Solar, 206
Storage, 217
Tidal, 233
Wave, 248
Wind, 263
__checkNormalizedProduction
Production, 167
__checkResourceKey1D
Resources, 191
__checkResourceKey2D
Resources, 192
__checkTimePoint
Production, 168
Resources, 193
__computeCubicProductionkW
Tidal, 233
Wind, 264
__computeEconomics
Model, 137
__computeExponentialProductionkW
Tidal, 234
Wind, 265
__computeFuelAndEmissions
Model, 137
__computeGaussianProductionkW
Wave, 248
__computeLevellizedCostOfEnergy
Model, 138
__computeLookupProductionkW
Tidal, 235
Wave, 249
Wind, 265
__computeNetLoad
Controller, 31
__computeNetPresentCost
Model, 138
__computeParaboloidProductionkW
Wave, 250
__computeRealDiscountAnnual
Storage, 218
__constructCombustionMap
Controller, 32
__flowToPower
Hydro, 75
__getAcceptableFlow
Hydro, 75
__getAvailableFlow
Hydro, 76
__getBcal
LiIon, 115
__getDataStringMatrix
Interpolator, 96
__getEacal
LiIon, 116
__getEfficiencyFactor
Hydro, 76
__getGenericCapitalCost
Diesel, 50
Hydro, 77
LiIon, 116
Solar, 206
Tidal, 235
Wave, 250
Wind, 266
__getGenericFuelIntercept
Diesel, 50
__getGenericFuelSlope
Diesel, 51
__getGenericOpMaintCost
Diesel, 51
Hydro, 77
LiIon, 116
Solar, 207

Tidal, 235
Wave, 251
Wind, 266
__getInterpolationIndex
Interpolator, 96
__getMaximumFlowm3hr
Hydro, 78
__getMinimumFlowm3hr
Hydro, 78
__getRenewableProduction
Controller, 34
__handleCombustionDispatch
Controller, 35
__handleDegradation
LiIon, 117
__handleNoncombustionDispatch
Controller, 36
__handleStartStop
Diesel, 52
Noncombustion, 156
Renewable, 183
__handleStorageCharging
Controller, 37, 39
__handleStorageDischarging
Controller, 40
__initInterpolator
Hydro, 78
__isNonNumeric
Interpolator, 97
__modelDegradation
LiIon, 117
__powerToFlow
Hydro, 80
__readData1D
Interpolator, 97
__readData2D
Interpolator, 98
__readHydroResource
Resources, 193
__readNormalizedProductionData
Production, 168
__readSolarResource
Resources, 194
__readTidalResource
Resources, 195
__readWaveResource
Resources, 196
__readWindResource
Resources, 197
__splitCommaSeparatedString
Interpolator, 100
__throwLengthError
Production, 169
Resources, 198
__throwReadError
Interpolator, 101
__toggleDepleted
LiIon, 119
__updateState
Hydro, 81
__writeSummary
Combustion, 14
Diesel, 53
Hydro, 82
LiIon, 119
Model, 139
Noncombustion, 157
Renewable, 184
Solar, 207
Storage, 219
Tidal, 236
Wave, 251
Wind, 266
__writeTimeSeries
Combustion, 14
Diesel, 55
Hydro, 83
LiIon, 121
Model, 142
Noncombustion, 157
Renewable, 184
Solar, 208
Storage, 219
Tidal, 237
Wave, 253
Wind, 268
∼Combustion
Combustion, 13
∼Controller
Controller, 27
∼Diesel
Diesel, 48
∼ElectricalLoad
ElectricalLoad, 64
∼Hydro
Hydro, 74
∼Interpolator
Interpolator, 93
∼LiIon
LiIon, 112
∼Model
Model, 136
∼Noncombustion
Noncombustion, 156
∼Production
Production, 166
∼Renewable
Renewable, 183
∼Resources
Resources, 190
∼Solar
Solar, 206
∼Storage
Storage, 217
∼Tidal
Tidal, 233

~Wave
  Wave, 247
~Wind
  Wind, 263

addData1D
  Interpolator, 101
addData2D
  Interpolator, 102
addDiesel
  Model, 143
addHydro
  Model, 143
addLiIon
  Model, 144
addResource
  Model, 144, 145
  Resources, 199, 200
addSolar
  Model, 145
addTidal
  Model, 146
addWave
  Model, 146
addWind
  Model, 146
applyDispatchControl
  Controller, 41

capacity_kW
  Production, 173
  ProductionInputs, 179
capital_cost
  DieselInputs, 60
  HydroInputs, 90
  LiIonInputs, 130
  Production, 173
  SolarInputs, 213
  Storage, 222
  TidalInputs, 242
  WaveInputs, 258
  WindInputs, 273
capital_cost_vec
  Production, 173
  Storage, 223
CH4_emissions_intensity_kgL
  Combustion, 19
  DieselInputs, 60
CH4_emissions_vec_kg
  Combustion, 19
CH4_kg
  Emissions, 68
charge_kWh
  Storage, 223
charge_vec_kWh
  Storage, 223
charging_efficiency
  LiIon, 125
  LiIonInputs, 130
charging_power_vec_kW
  Storage, 223
clear
  Controller, 42
  ElectricalLoad, 64
  Model, 147
  Resources, 201
CO2_emissions_intensity_kgL
  Combustion, 20
  DieselInputs, 60
CO2_emissions_vec_kg
  Combustion, 20
CO2_kg
  Emissions, 68
CO_emissions_intensity_kgL
  Combustion, 20
  DieselInputs, 61
CO_emissions_vec_kg
  Combustion, 20
CO_kg
  Emissions, 68
Combustion, 9
  __checkInputs, 14
  __writeSummary, 14
  __writeTimeSeries, 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, 280
DIESEL, 280
FUEL_MODE_LINEAR, 282
FUEL_MODE_LOOKUP, 282
FuelMode, 280
N_COMBUSTION_TYPES, 280
N_FUEL_MODES, 282
combustion_inputs
DieselInputs, 61
combustion_map
Controller, 44
combustion_ptr_vec
Model, 150
CombustionInputs, 24
fuel_mode, 24
nominal_fuel_escalation_annual, 24
path_2_fuel_interp_data, 25
production_inputs, 25
CombustionType
Combustion.h, 280
commit
Combustion, 15
Diesel, 56
Hydro, 84
Noncombustion, 157, 158
Production, 169
Renewable, 184
Solar, 209
Tidal, 238
Wave, 254
Wind, 269
commitCharge
LiIon, 121
Storage, 219
commitDischarge
LiIon, 122
Storage, 219
computeEconomics
Combustion, 16
Noncombustion, 158
Production, 170
Renewable, 185
Storage, 220
computeFuelAndEmissions
Combustion, 16
computeProductionkW
Renewable, 185, 186
Solar, 210
Tidal, 239
Wave, 254
Wind, 269
computeRealDiscountAnnual
Production, 171
control_mode
Controller, 44
ModelInputs, 152
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, 39
__handleStorageDischarging, 40
∼Controller, 27
applyDispatchControl, 41
clear, 42
combustion_map, 44
control_mode, 44
control_string, 44
Controller, 27
init, 42
missed_load_vec_kW, 44
net_load_vec_kW, 44
setControlMode, 43
controller
Model, 150
Controller.h
ControlMode, 276
CYCLE_CHARGING, 276
LOAD_FOLLOWING, 276
N_CONTROL_MODES, 276
ControlMode
Controller.h, 276
curtailment_vec_kW
Production, 173
CYCLE_CHARGING
Controller.h, 276

def
PYBIND11_Controller.cpp, 328
PYBIND11_Diesel.cpp, 307
PYBIND11_Hydro.cpp, 309
PYBIND11_Interpolator.cpp, 331
PYBIND11_LiIon.cpp, 337
PYBIND11_Noncombustion.cpp, 312
PYBIND11_Production.cpp, 314
PYBIND11_Renewable.cpp, 319
PYBIND11_Solar.cpp, 320
def_readwrite
PYBIND11_Combustion.cpp, 304, 305
PYBIND11_Controller.cpp, 328
PYBIND11_Diesel.cpp, 307, 308
PYBIND11_ElectricalLoad.cpp, 330
PYBIND11_Hydro.cpp, 309–311
PYBIND11_Interpolator.cpp, 332, 333
PYBIND11_LiIon.cpp, 337–340
PYBIND11_Model.cpp, 334
PYBIND11_Production.cpp, 315–318

PYBIND11_Resources.cpp, 335
PYBIND11_Solar.cpp, 320, 321
PYBIND11_Storage.cpp, 341, 342
PYBIND11_Tidal.cpp, 322, 323
PYBIND11_Wave.cpp, 324, 325
PYBIND11_Wind.cpp, 326, 327
degradation_a_cal
LiIon, 125
LiIonInputs, 130
degradation_alpha
LiIon, 125
LiIonInputs, 130
degradation_B_hat_cal_0
LiIon, 125
LiIonInputs, 130
degradation_beta
LiIon, 125
LiIonInputs, 130
degradation_Ea_cal_0
LiIon, 126
LiIonInputs, 131
degradation_r_cal
LiIon, 126
LiIonInputs, 131
degradation_s_cal
LiIon, 126
LiIonInputs, 131
derating
Solar, 211
SolarInputs, 213
design_energy_period_s
Wave, 256
WaveInputs, 258
design_significant_wave_height_m
Wave, 256
WaveInputs, 258
design_speed_ms
Tidal, 240
TidalInputs, 242
Wind, 271
WindInputs, 273
DIESEL
Combustion.h, 280
Diesel, 45
__checkInputs, 48
__getGenericCapitalCost, 50
__getGenericFuelIntercept, 50
__getGenericFuelSlope, 51
__getGenericOpMaintCost, 51
__handleStartStop, 52
__writeSummary, 53
__writeTimeSeries, 55
~Diesel, 48
commit, 56
Diesel, 47
handleReplacement, 57
minimum_load_ratio, 58
minimum_runtime_hrs, 58
requestProductionkW, 57
time_since_last_start_hrs, 58
DieselInputs, 59
capital_cost, 60
CH4_emissions_intensity_kgL, 60
CO2_emissions_intensity_kgL, 60
CO_emissions_intensity_kgL, 61
combustion_inputs, 61
fuel_cost_L, 61
linear_fuel_intercept_LkWh, 61
linear_fuel_slope_LkWh, 61
minimum_load_ratio, 61
minimum_runtime_hrs, 62
NOx_emissions_intensity_kgL, 62
operation_maintenance_cost_kWh, 62
PM_emissions_intensity_kgL, 62
replace_running_hrs, 62
SOx_emissions_intensity_kgL, 62
discharging_efficiency
LiIon, 126
LiIonInputs, 131
discharging_power_vec_kW
Storage, 223
dispatch_vec_kW
Production, 173
dt_vec_hrs
ElectricalLoad, 66
dynamic_energy_capacity_kWh
LiIon, 126

electrical_load
Model, 150
ElectricalLoad, 63
~ElectricalLoad, 64
clear, 64
dt_vec_hrs, 66
ElectricalLoad, 64
load_vec_kW, 66
max_load_kW, 66
mean_load_kW, 66
min_load_kW, 67
n_points, 67
n_years, 67
path_2_electrical_load_time_series, 67
readLoadData, 65
time_vec_hrs, 67
Emissions, 68
CH4_kg, 68
CO2_kg, 68
CO_kg, 68
NOx_kg, 69
PM_kg, 69
SOx_kg, 69
energy_capacity_kWh
Storage, 223
StorageInputs, 228
example.cpp
main, 298
expectedErrorNotDetected

testing_utils.cpp, 459
testing_utils.h, 466

FLOAT_TOLERANCE
testing_utils.h, 466
FLOW_TO_POWER_INTERP_KEY
Hydro.h, 284
fluid_density_kgm3
Hydro, 86
HydroInputs, 90
fuel_consumption_vec_L
Combustion, 20
fuel_cost_L
Combustion, 20
DieselInputs, 61
fuel_cost_vec
Combustion, 21
fuel_mode
Combustion, 21
CombustionInputs, 24
FUEL_MODE_LINEAR
Combustion.h, 282
FUEL_MODE_LOOKUP
Combustion.h, 282
fuel_mode_str
Combustion, 21
FuelMode
Combustion.h, 280

gas_constant_JmolK
LiIon, 126
LiIonInputs, 131
GENERATOR_EFFICIENCY_INTERP_KEY
Hydro.h, 284
getAcceptablekW
LiIon, 123
Storage, 221
getAvailablekW
LiIon, 124
Storage, 221
getEmissionskg
Combustion, 16
getFuelConsumptionL
Combustion, 17
getProductionkW
Production, 172

handleReplacement
Combustion, 18
Diesel, 57
Hydro, 85
LiIon, 124
Noncombustion, 159
Production, 172
Renewable, 186
Solar, 211
Storage, 221
Tidal, 240
Wave, 256
Wind, 271
header/Controller.h, 275
header/doxygen_cite.h, 276
header/ElectricalLoad.h, 277
header/Interpolator.h, 277
header/Model.h, 278
header/Production/Combustion/Combustion.h, 279
header/Production/Combustion/Diesel.h, 282
header/Production/Noncombustion/Hydro.h, 283
header/Production/Noncombustion/Noncombustion.h, 285
header/Production/Production.h, 286
header/Production/Renewable/Renewable.h, 287
header/Production/Renewable/Solar.h, 288
header/Production/Renewable/Tidal.h, 289
header/Production/Renewable/Wave.h, 291
header/Production/Renewable/Wind.h, 292
header/Resources.h, 294
header/std_includes.h, 295
header/Storage/LiIon.h, 295
header/Storage/Storage.h, 296
HYDRO
Noncombustion.h, 286
Hydro, 70
__checkInputs, 74
__flowToPower, 75
__getAcceptableFlow, 75
__getAvailableFlow, 76
__getEfficiencyFactor, 76
__getGenericCapitalCost, 77
__getGenericOpMaintCost, 77
__getMaximumFlowm3hr, 78
__getMinimumFlowm3hr, 78
__initInterpolator, 78
__powerToFlow, 80
__updateState, 81
__writeSummary, 82
__writeTimeSeries, 83
∼Hydro, 74
commit, 84
fluid_density_kgm3, 86
handleReplacement, 85
Hydro, 72
init_reservoir_state, 87
maximum_flow_m3hr, 87
minimum_flow_m3hr, 87
minimum_power_kW, 87
net_head_m, 87
requestProductionkW, 85
reservoir_capacity_m3, 87
spill_rate_vec_m3hr, 88
stored_volume_m3, 88
stored_volume_vec_m3, 88
turbine_flow_vec_m3hr, 88
turbine_type, 88
Hydro.h
FLOW_TO_POWER_INTERP_KEY, 284
GENERATOR_EFFICIENCY_INTERP_KEY, 284

HYDRO_TURBINE_FRANCIS, 284
HYDRO_TURBINE_KAPLAN, 284
HYDRO_TURBINE_PELTON, 284
HydroInterpKeys, 284
HydroTurbineType, 284
N_HYDRO_INTERP_KEYS, 284
N_HYDRO_TURBINES, 284
TURBINE_EFFICIENCY_INTERP_KEY, 284
HYDRO_TURBINE_FRANCIS
Hydro.h, 284
HYDRO_TURBINE_KAPLAN
Hydro.h, 284
HYDRO_TURBINE_PELTON
Hydro.h, 284
HydroInputs, 89
capital_cost, 90
fluid_density_kgm3, 90
init_reservoir_state, 90
net_head_m, 90
noncombustion_inputs, 90
operation_maintenance_cost_kWh, 90
reservoir_capacity_m3, 91
resource_key, 91
turbine_type, 91
HydroInterpKeys
Hydro.h, 284
HydroTurbineType
Hydro.h, 284
hysteresis_SOC
LiIon, 127
LiIonInputs, 131

init
Controller, 42
init_reservoir_state
Hydro, 87
HydroInputs, 90
init_SOC
LiIon, 127
LiIonInputs, 132
interp1D
Interpolator, 102
interp2D
Interpolator, 103
interp_map_1D
Interpolator, 104
interp_map_2D
Interpolator, 104
Interpolator, 91
__checkBounds1D, 93
__checkBounds2D, 94
__checkDataKey1D, 95
__checkDataKey2D, 95
__getDataStringMatrix, 96
__getInterpolationIndex, 96
__isNonNumeric, 97
__readData1D, 97
__readData2D, 98
__splitCommaSeparatedString, 100
__throwReadError, 101
∼Interpolator, 93
addData1D, 101
addData2D, 102
interp1D, 102
interp2D, 103
interp_map_1D, 104
interp_map_2D, 104
Interpolator, 93
path_map_1D, 104
path_map_2D, 104
interpolator
Production, 174
Storage, 224
InterpolatorStruct1D, 105
max_x, 105
min_x, 105
n_points, 105
x_vec, 106
y_vec, 106
InterpolatorStruct2D, 106
max_x, 107
max_y, 107
min_x, 107
min_y, 107
n_cols, 107
n_rows, 107
x_vec, 108
y_vec, 108
z_matrix, 108
is_depleted
Storage, 224
is_running
Production, 174
is_running_vec
Production, 174
is_sunk
Production, 174
ProductionInputs, 179
Storage, 224
StorageInputs, 228

levellized_cost_of_energy_kWh
Model, 150
Production, 174
Storage, 224
LIION
Storage.h, 297
LiIon, 109
__checkInputs, 113
__getBcal, 115
__getEacal, 116
__getGenericCapitalCost, 116
__getGenericOpMaintCost, 116
__handleDegradation, 117
__modelDegradation, 117
__toggleDepleted, 119
__writeSummary, 119
__writeTimeSeries, 121

~LiIon, 112
charging_efficiency, 125
commitCharge, 121
commitDischarge, 122
degradation_a_cal, 125
degradation_alpha, 125
degradation_B_hat_cal_0, 125
degradation_beta, 125
degradation_Ea_cal_0, 126
degradation_r_cal, 126
degradation_s_cal, 126
discharging_efficiency, 126
dynamic_energy_capacity_kWh, 126
gas_constant_JmolK, 126
getAcceptablekW, 123
getAvailablekW, 124
handleReplacement, 124
hysteresis_SOC, 127
init_SOC, 127
LiIon, 111
max_SOC, 127
min_SOC, 127
replace_SOH, 127
SOH, 127
SOH_vec, 128
temperature_K, 128
LiIonInputs, 128
capital_cost, 130
charging_efficiency, 130
degradation_a_cal, 130
degradation_alpha, 130
degradation_B_hat_cal_0, 130
degradation_beta, 130
degradation_Ea_cal_0, 131
degradation_r_cal, 131
degradation_s_cal, 131
discharging_efficiency, 131
gas_constant_JmolK, 131
hysteresis_SOC, 131
init_SOC, 132
max_SOC, 132
min_SOC, 132
operation_maintenance_cost_kWh, 132
replace_SOH, 132
storage_inputs, 132
temperature_K, 133
linear_fuel_intercept_LkWh
Combustion, 21
DieselInputs, 61
linear_fuel_slope_LkWh
Combustion, 21
DieselInputs, 61
LOAD_FOLLOWING
Controller.h, 276
load_vec_kW
ElectricalLoad, 66
main
example.cpp, 298
test_Combustion.cpp, 352
test_Controller.cpp, 411
test_Diesel.cpp, 356
test_ElectricalLoad.cpp, 413
test_Hydro.cpp, 366
test_Interpolator.cpp, 416
test_LiIon.cpp, 403
test_Model.cpp, 427
test_Noncombustion.cpp, 371
test_Production.cpp, 400
test_Renewable.cpp, 373
test_Resources.cpp, 448
test_Solar.cpp, 375
test_Storage.cpp, 408
test_Tidal.cpp, 382
test_Wave.cpp, 387
test_Wind.cpp, 395
max_load_kW
ElectricalLoad, 66
max_SOC
LiIon, 127
LiIonInputs, 132
max_x
InterpolatorStruct1D, 105
InterpolatorStruct2D, 107
max_y
InterpolatorStruct2D, 107
maximum_flow_m3hr
Hydro, 87
mean_load_kW
ElectricalLoad, 66
min_load_kW
ElectricalLoad, 67
min_SOC
LiIon, 127
LiIonInputs, 132
min_x
InterpolatorStruct1D, 105
InterpolatorStruct2D, 107
min_y
InterpolatorStruct2D, 107
minimum_flow_m3hr
Hydro, 87
minimum_load_ratio
Diesel, 58
DieselInputs, 61
minimum_power_kW
Hydro, 87
minimum_runtime_hrs
Diesel, 58
DieselInputs, 62
missed_load_vec_kW
Controller, 44
Model, 133
__checkInputs, 136
__computeEconomics, 137
__computeFuelAndEmissions, 137
__computeLevellizedCostOfEnergy, 138

__computeNetPresentCost, 138
__writeSummary, 139
__writeTimeSeries, 142
~Model, 136
addDiesel, 143
addHydro, 143
addLiIon, 144
addResource, 144, 145
addSolar, 145
addTidal, 146
addWave, 146
addWind, 146
clear, 147
combustion_ptr_vec, 150
controller, 150
electrical_load, 150
levellized_cost_of_energy_kWh, 150
Model, 135, 136
net_present_cost, 150
noncombustion_ptr_vec, 150
renewable_ptr_vec, 151
reset, 147
resources, 151
run, 148
storage_ptr_vec, 151
total_dispatch_discharge_kWh, 151
total_emissions, 151
total_fuel_consumed_L, 151
total_renewable_dispatch_kWh, 152
writeResults, 148
ModelInputs, 152
control_mode, 152
path_2_electrical_load_time_series, 153

n_cols
InterpolatorStruct2D, 107
N_COMBUSTION_TYPES
Combustion.h, 280
N_CONTROL_MODES
Controller.h, 276
N_FUEL_MODES
Combustion.h, 282
N_HYDRO_INTERP_KEYS
Hydro.h, 284
N_HYDRO_TURBINES
Hydro.h, 284
N_NONCOMBUSTION_TYPES
Noncombustion.h, 286
n_points
ElectricalLoad, 67
InterpolatorStruct1D, 105
Production, 174
Storage, 224
N_RENEWABLE_TYPES
Renewable.h, 288
n_replacements
Production, 175
Storage, 224
n_rows
InterpolatorStruct2D, 107
n_starts
Production, 175
N_STORAGE_TYPES
Storage.h, 297
N_TIDAL_POWER_PRODUCTION_MODELS
Tidal.h, 291
N_WAVE_POWER_PRODUCTION_MODELS
Wave.h, 292
N_WIND_POWER_PRODUCTION_MODELS
Wind.h, 293
n_years
ElectricalLoad, 67
Production, 175
Storage, 225
net_head_m
Hydro, 87
HydroInputs, 90
net_load_vec_kW
Controller, 44
net_present_cost
Model, 150
Production, 175
Storage, 225
nominal_discount_annual
Production, 175
ProductionInputs, 179
Storage, 225
StorageInputs, 228
nominal_fuel_escalation_annual
Combustion, 21
CombustionInputs, 24
nominal_inflation_annual
Production, 175
ProductionInputs, 179
Storage, 225
StorageInputs, 228
Noncombustion, 153
__checkInputs, 156
__handleStartStop, 156
__writeSummary, 157
__writeTimeSeries, 157
~Noncombustion, 156
commit, 157, 158
computeEconomics, 158
handleReplacement, 159
Noncombustion, 155
requestProductionkW, 159
resource_key, 160
type, 160
writeResults, 159
Noncombustion.h
HYDRO, 286
N_NONCOMBUSTION_TYPES, 286
NoncombustionType, 286
noncombustion_inputs
HydroInputs, 90
noncombustion_ptr_vec

Model, 150
NoncombustionInputs, 161
production_inputs, 161
NoncombustionType
Noncombustion.h, 286
normalized_production_series_given
Production, 176
normalized_production_vec
Production, 176
NOx_emissions_intensity_kgL
Combustion, 22
DieselInputs, 62
NOx_emissions_vec_kg
Combustion, 22
NOx_kg
Emissions, 69

operation_maintenance_cost_kWh
DieselInputs, 62
HydroInputs, 90
LiIonInputs, 132
Production, 176
SolarInputs, 213
Storage, 225
TidalInputs, 242
WaveInputs, 259
WindInputs, 273
operation_maintenance_cost_vec
Production, 176
Storage, 225

path_2_electrical_load_time_series
ElectricalLoad, 67
ModelInputs, 153
path_2_fuel_interp_data
CombustionInputs, 25
path_2_normalized_performance_matrix
WaveInputs, 259
path_2_normalized_production_time_series
Production, 176
ProductionInputs, 179
path_map_1D
Interpolator, 104
Resources, 201
path_map_2D
Interpolator, 104
Resources, 201
PM_emissions_intensity_kgL
Combustion, 22
DieselInputs, 62
PM_emissions_vec_kg
Combustion, 22
PM_kg
Emissions, 69
power_capacity_kW
Storage, 226
StorageInputs, 228
power_kW
Storage, 226
power_model
Tidal, 241
TidalInputs, 243
Wave, 257
WaveInputs, 259
Wind, 271
WindInputs, 273
power_model_string
Tidal, 241
Wave, 257
Wind, 271
print_flag
Production, 176
ProductionInputs, 179
Storage, 226
StorageInputs, 228
printGold
testing_utils.cpp, 460
testing_utils.h, 466
printGreen
testing_utils.cpp, 460
testing_utils.h, 467
printRed
testing_utils.cpp, 460
testing_utils.h, 467
Production, 162
__checkInputs, 166
__checkNormalizedProduction, 167
__checkTimePoint, 168
__readNormalizedProductionData, 168
__throwLengthError, 169
∼Production, 166
capacity_kW, 173
capital_cost, 173
capital_cost_vec, 173
commit, 169
computeEconomics, 170
computeRealDiscountAnnual, 171
curtailment_vec_kW, 173
dispatch_vec_kW, 173
getProductionkW, 172
handleReplacement, 172
interpolator, 174
is_running, 174
is_running_vec, 174
is_sunk, 174
levellized_cost_of_energy_kWh, 174
n_points, 174
n_replacements, 175
n_starts, 175
n_years, 175
net_present_cost, 175
nominal_discount_annual, 175
nominal_inflation_annual, 175
normalized_production_series_given, 176
normalized_production_vec, 176
operation_maintenance_cost_kWh, 176
operation_maintenance_cost_vec, 176

path_2_normalized_production_time_series, 176
print_flag, 176
Production, 165
production_vec_kW, 177
real_discount_annual, 177
replace_running_hrs, 177
running_hours, 177
storage_vec_kW, 177
total_dispatch_kWh, 177
type_str, 178
production_inputs
CombustionInputs, 25
NoncombustionInputs, 161
RenewableInputs, 188
production_vec_kW
Production, 177
ProductionInputs, 178
capacity_kW, 179
is_sunk, 179
nominal_discount_annual, 179
nominal_inflation_annual, 179
path_2_normalized_production_time_series, 179
print_flag, 179
replace_running_hrs, 180
projects/example.cpp, 298
PYBIND11_Combustion.cpp
def_readwrite, 304, 305
value, 305
PYBIND11_Controller.cpp
def, 328
def_readwrite, 328
value, 329
PYBIND11_Diesel.cpp
def, 307
def_readwrite, 307, 308
PYBIND11_ElectricalLoad.cpp
def_readwrite, 330
PYBIND11_Hydro.cpp
def, 309
def_readwrite, 309–311
value, 311
PYBIND11_Interpolator.cpp
def, 331
def_readwrite, 332, 333
PYBIND11_LiIon.cpp
def, 337
def_readwrite, 337–340
PYBIND11_Model.cpp
def_readwrite, 334
PYBIND11_MODULE
PYBIND11_PGM.cpp, 303
PYBIND11_Noncombustion.cpp
def, 312
value, 312
PYBIND11_PGM.cpp
PYBIND11_MODULE, 303
PYBIND11_Production.cpp
def, 314
def_readwrite, 315–318
PYBIND11_Renewable.cpp
def, 319
value, 319
PYBIND11_Resources.cpp
def_readwrite, 335
PYBIND11_Solar.cpp
def, 320
def_readwrite, 320, 321
PYBIND11_Storage.cpp
def_readwrite, 341, 342
value, 341
PYBIND11_Tidal.cpp
def_readwrite, 322, 323
value, 322
PYBIND11_Wave.cpp
def_readwrite, 324, 325
value, 324, 325
PYBIND11_Wind.cpp
def_readwrite, 326, 327
value, 326
pybindings/PYBIND11_PGM.cpp, 302
pybindings/snippets/Production/Combustion/PYBIND11_Combustion.cpp, 304
pybindings/snippets/Production/Combustion/PYBIND11_Diesel.cpp, 305
pybindings/snippets/Production/Noncombustion/PYBIND11_Hydro.cpp, 308
pybindings/snippets/Production/Noncombustion/PYBIND11_Noncombustio 312
pybindings/snippets/Production/PYBIND11_Production.cpp, 313
pybindings/snippets/Production/Renewable/PYBIND11_Renewable.cpp, 318
pybindings/snippets/Production/Renewable/PYBIND11_Solar.cpp, 320
pybindings/snippets/Production/Renewable/PYBIND11_Tidal.cpp, 321
pybindings/snippets/Production/Renewable/PYBIND11_Wave.cpp, 323
pybindings/snippets/Production/Renewable/PYBIND11_Wind.cpp, 325
pybindings/snippets/PYBIND11_Controller.cpp, 327
pybindings/snippets/PYBIND11_ElectricalLoad.cpp, 329
pybindings/snippets/PYBIND11_Interpolator.cpp, 331
pybindings/snippets/PYBIND11_Model.cpp, 333
pybindings/snippets/PYBIND11_Resources.cpp, 334
pybindings/snippets/Storage/PYBIND11_LiIon.cpp, 335
pybindings/snippets/Storage/PYBIND11_Storage.cpp, 340

readLoadData
ElectricalLoad, 65
real_discount_annual
Production, 177
Storage, 226
real_fuel_escalation_annual
Combustion, 22

Renewable, 180
__checkInputs, 183
__handleStartStop, 183
__writeSummary, 184
__writeTimeSeries, 184
∼Renewable, 183
commit, 184
computeEconomics, 185
computeProductionkW, 185, 186
handleReplacement, 186
Renewable, 182
resource_key, 187
type, 187
writeResults, 186
Renewable.h
N_RENEWABLE_TYPES, 288
RenewableType, 288
SOLAR, 288
TIDAL, 288
WAVE, 288
WIND, 288
renewable_inputs
SolarInputs, 213
TidalInputs, 243
WaveInputs, 259
WindInputs, 274
renewable_ptr_vec
Model, 151
RenewableInputs, 188
production_inputs, 188
RenewableType
Renewable.h, 288
replace_running_hrs
DieselInputs, 62
Production, 177
ProductionInputs, 180
replace_SOH
LiIon, 127
LiIonInputs, 132
requestProductionkW
Combustion, 18
Diesel, 57
Hydro, 85
Noncombustion, 159
reservoir_capacity_m3
Hydro, 87
HydroInputs, 91
reset
Model, 147
resource_key
HydroInputs, 91
Noncombustion, 160
Renewable, 187
SolarInputs, 213
TidalInputs, 243
WaveInputs, 259
WindInputs, 274
resource_map_1D
Resources, 201
resource_map_2D
Resources, 202
Resources, 189
__checkResourceKey1D, 191
__checkResourceKey2D, 192
__checkTimePoint, 193
__readHydroResource, 193
__readSolarResource, 194
__readTidalResource, 195
__readWaveResource, 196
__readWindResource, 197
__throwLengthError, 198
∼Resources, 190
addResource, 199, 200
clear, 201
path_map_1D, 201
path_map_2D, 201
resource_map_1D, 201
resource_map_2D, 202
Resources, 190
string_map_1D, 202
string_map_2D, 202
resources
Model, 151
run
Model, 148
running_hours
Production, 177

setControlMode
Controller, 43
SOH
LiIon, 127
SOH_vec
LiIon, 128
SOLAR
Renewable.h, 288
Solar, 203
__checkInputs, 206
__getGenericCapitalCost, 206
__getGenericOpMaintCost, 207
__writeSummary, 207
__writeTimeSeries, 208
∼Solar, 206
commit, 209
computeProductionkW, 210
derating, 211
handleReplacement, 211
Solar, 204, 205
SolarInputs, 212
capital_cost, 213
derating, 213
operation_maintenance_cost_kWh, 213
renewable_inputs, 213
resource_key, 213
source/Controller.cpp, 342
source/ElectricalLoad.cpp, 343
source/Interpolator.cpp, 343

source/Model.cpp, 344
source/Production/Combustion/Combustion.cpp, 344
source/Production/Combustion/Diesel.cpp, 345
source/Production/Noncombustion/Hydro.cpp, 345
source/Production/Noncombustion/Noncombustion.cpp, 346
source/Production/Production.cpp, 347
source/Production/Renewable/Renewable.cpp, 347
source/Production/Renewable/Solar.cpp, 348
source/Production/Renewable/Tidal.cpp, 348
source/Production/Renewable/Wave.cpp, 349
source/Production/Renewable/Wind.cpp, 349
source/Resources.cpp, 350
source/Storage/LiIon.cpp, 351
source/Storage/Storage.cpp, 351
SOx_emissions_intensity_kgL
- Combustion, 22
- DieselInputs, 62

SOx_emissions_vec_kg
- Combustion, 23

SOx_kg
- Emissions, 69

spill_rate_vec_m3hr
- Hydro, 88

Storage, 214
- __checkInputs, 217
- __computeRealDiscountAnnual, 218
- __writeSummary, 219
- __writeTimeSeries, 219
- ∼Storage, 217
- capital_cost, 222
- capital_cost_vec, 223
- charge_kWh, 223
- charge_vec_kWh, 223
- charging_power_vec_kW, 223
- commitCharge, 219
- commitDischarge, 219
- computeEconomics, 220
- discharging_power_vec_kW, 223
- energy_capacity_kWh, 223
- getAcceptablekW, 221
- getAvailablekW, 221
- handleReplacement, 221
- interpolator, 224
- is_depleted, 224
- is_sunk, 224
- levellized_cost_of_energy_kWh, 224
- n_points, 224
- n_replacements, 224
- n_years, 225
- net_present_cost, 225
- nominal_discount_annual, 225
- nominal_inflation_annual, 225
- operation_maintenance_cost_kWh, 225
- operation_maintenance_cost_vec, 225
- power_capacity_kW, 226
- power_kW, 226
- print_flag, 226
- real_discount_annual, 226
- Storage, 216
- total_discharge_kWh, 226
- type, 226
- type_str, 227
- writeResults, 221

Storage.h
- LIION, 297
- N_STORAGE_TYPES, 297
- StorageType, 297

storage_inputs
- LiIonInputs, 132

storage_ptr_vec
- Model, 151

storage_vec_kW
- Production, 177

StorageInputs, 227
- energy_capacity_kWh, 228
- is_sunk, 228
- nominal_discount_annual, 228
- nominal_inflation_annual, 228
- power_capacity_kW, 228
- print_flag, 228

StorageType
- Storage.h, 297

stored_volume_m3
- Hydro, 88

stored_volume_vec_m3
- Hydro, 88

string_map_1D
- Resources, 202

string_map_2D
- Resources, 202

temperature_K
- LiIon, 128
- LiIonInputs, 133

test/source/Production/Combustion/test_Combustion.cpp, 352
test/source/Production/Combustion/test_Diesel.cpp, 354
test/source/Production/Noncombustion/test_Hydro.cpp, 365
test/source/Production/Noncombustion/test_Noncombustion.cpp, 370
test/source/Production/Renewable/test_Renewable.cpp, 372
test/source/Production/Renewable/test_Solar.cpp, 374
test/source/Production/Renewable/test_Tidal.cpp, 381
test/source/Production/Renewable/test_Wave.cpp, 386
test/source/Production/Renewable/test_Wind.cpp, 394
test/source/Production/test_Production.cpp, 399
test/source/Storage/test_LiIon.cpp, 402
test/source/Storage/test_Storage.cpp, 407
test/source/test_Controller.cpp, 410
test/source/test_ElectricalLoad.cpp, 412
test/source/test_Interpolator.cpp, 416
test/source/test_Model.cpp, 426
test/source/test_Resources.cpp, 446

test/utils/testing_utils.cpp, 458
test/utils/testing_utils.h, 464
test_Combustion.cpp
    main, 352
    testConstruct_Combustion, 353
test_Controller.cpp
    main, 411
    testConstruct_Controller, 411
test_Diesel.cpp
    main, 356
    testBadConstruct_Diesel, 356
    testCapacityConstraint_Diesel, 357
    testCommit_Diesel, 357
    testConstruct_Diesel, 359
    testConstructLookup_Diesel, 360
    testEconomics_Diesel, 360
    testFuelConsumptionEmissions_Diesel, 361
    testFuelLookup_Diesel, 363
    testMinimumLoadRatioConstraint_Diesel, 364
    testMinimumRuntimeConstraint_Diesel, 364
test_ElectricalLoad.cpp
    main, 413
    testConstruct_ElectricalLoad, 413
    testDataRead_ElectricalLoad, 413
    testPostConstructionAttributes_ElectricalLoad, 415
test_Hydro.cpp
    main, 366
    testCommit_Hydro, 367
    testConstruct_Hydro, 368
    testEfficiencyInterpolation_Hydro, 369
test_Interpolator.cpp
    main, 416
    testBadIndexing1D_Interpolator, 417
    testConstruct_Interpolator, 418
    testDataRead1D_Interpolator, 418
    testDataRead2D_Interpolator, 419
    testInterpolation1D_Interpolator, 422
    testInterpolation2D_Interpolator, 423
    testInvalidInterpolation1D_Interpolator, 424
    testInvalidInterpolation2D_Interpolator, 425
test_LiIon.cpp
    main, 403
    testBadConstruct_LiIon, 404
    testCommitCharge_LiIon, 404
    testCommitDischarge_LiIon, 405
    testConstruct_LiIon, 406
test_Model.cpp
    main, 427
    testAddDiesel_Model, 429
    testAddHydro_Model, 430
    testAddHydroResource_Model, 431
    testAddLiIon_Model, 432
    testAddSolar_Model, 432
    testAddSolar_productionOverride_Model, 433
    testAddSolarResource_Model, 434
    testAddTidal_Model, 435
    testAddTidalResource_Model, 436
    testAddWave_Model, 437
    testAddWaveResource_Model, 437
    testAddWind_Model, 439
    testAddWindResource_Model, 440
    testBadConstruct_Model, 441
    testConstruct_Model, 441
    testEconomics_Model, 442
    testElectricalLoadData_Model, 442
    testFuelConsumptionEmissions_Model, 443
    testLoadBalance_Model, 444
    testPostConstructionAttributes_Model, 446
test_Noncombustion.cpp
    main, 371
    testConstruct_Noncombustion, 371
test_Production.cpp
    main, 400
    testBadConstruct_Production, 400
    testConstruct_Production, 401
test_Renewable.cpp
    main, 373
    testConstruct_Renewable, 373
test_Resources.cpp
    main, 448
    testAddHydroResource_Resources, 449
    testAddSolarResource_Resources, 450
    testAddTidalResource_Resources, 451
    testAddWaveResource_Resources, 453
    testAddWindResource_Resources, 455
    testBadAdd_Resources, 456
    testConstruct_Resources, 458
test_Solar.cpp
    main, 375
    testBadConstruct_Solar, 376
    testCommit_Solar, 376
    testConstruct_Solar, 378
    testEconomics_Solar, 379
    testProductionConstraint_Solar, 379
    testProductionOverride_Solar, 380
test_Storage.cpp
    main, 408
    testBadConstruct_Storage, 409
    testConstruct_Storage, 409
test_Tidal.cpp
    main, 382
    testBadConstruct_Tidal, 382
    testCommit_Tidal, 383
    testConstruct_Tidal, 384
    testEconomics_Tidal, 385
    testProductionConstraint_Tidal, 386
test_Wave.cpp
    main, 387
    testBadConstruct_Wave, 388
    testCommit_Wave, 389
    testConstruct_Wave, 390
    testConstructLookup_Wave, 391
    testEconomics_Wave, 391
    testProductionConstraint_Wave, 392
    testProductionLookup_Wave, 392
test_Wind.cpp

main, 395
testBadConstruct_Wind, 395
testCommit_Wind, 396
testConstruct_Wind, 397
testEconomics_Wind, 398
testProductionConstraint_Wind, 398
testAddDiesel_Model
test_Model.cpp, 429
testAddHydro_Model
test_Model.cpp, 430
testAddHydroResource_Model
test_Model.cpp, 431
testAddHydroResource_Resources
test_Resources.cpp, 449
testAddLiIon_Model
test_Model.cpp, 432
testAddSolar_Model
test_Model.cpp, 432
testAddSolar_productionOverride_Model
test_Model.cpp, 433
testAddSolarResource_Model
test_Model.cpp, 434
testAddSolarResource_Resources
test_Resources.cpp, 450
testAddTidal_Model
test_Model.cpp, 435
testAddTidalResource_Model
test_Model.cpp, 436
testAddTidalResource_Resources
test_Resources.cpp, 451
testAddWave_Model
test_Model.cpp, 437
testAddWaveResource_Model
test_Model.cpp, 437
testAddWaveResource_Resources
test_Resources.cpp, 453
testAddWind_Model
test_Model.cpp, 439
testAddWindResource_Model
test_Model.cpp, 440
testAddWindResource_Resources
test_Resources.cpp, 455
testBadAdd_Resources
test_Resources.cpp, 456
testBadConstruct_Diesel
test_Diesel.cpp, 356
testBadConstruct_LiIon
test_LiIon.cpp, 404
testBadConstruct_Model
test_Model.cpp, 441
testBadConstruct_Production
test_Production.cpp, 400
testBadConstruct_Solar
test_Solar.cpp, 376
testBadConstruct_Storage
test_Storage.cpp, 409
testBadConstruct_Tidal
test_Tidal.cpp, 382
testBadConstruct_Wave
test_Wave.cpp, 388
testBadConstruct_Wind
test_Wind.cpp, 395
testBadIndexing1D_Interpolator
test_Interpolator.cpp, 417
testCapacityConstraint_Diesel
test_Diesel.cpp, 357
testCommit_Diesel
test_Diesel.cpp, 357
testCommit_Hydro
test_Hydro.cpp, 367
testCommit_Solar
test_Solar.cpp, 376
testCommit_Tidal
test_Tidal.cpp, 383
testCommit_Wave
test_Wave.cpp, 389
testCommit_Wind
test_Wind.cpp, 396
testCommitCharge_LiIon
test_LiIon.cpp, 404
testCommitDischarge_LiIon
test_LiIon.cpp, 405
testConstruct_Combustion
test_Combustion.cpp, 353
testConstruct_Controller
test_Controller.cpp, 411
testConstruct_Diesel
test_Diesel.cpp, 359
testConstruct_ElectricalLoad
test_ElectricalLoad.cpp, 413
testConstruct_Hydro
test_Hydro.cpp, 368
testConstruct_Interpolator
test_Interpolator.cpp, 418
testConstruct_LiIon
test_LiIon.cpp, 406
testConstruct_Model
test_Model.cpp, 441
testConstruct_Noncombustion
test_Noncombustion.cpp, 371
testConstruct_Production
test_Production.cpp, 401
testConstruct_Renewable
test_Renewable.cpp, 373
testConstruct_Resources
test_Resources.cpp, 458
testConstruct_Solar
test_Solar.cpp, 378
testConstruct_Storage
test_Storage.cpp, 409
testConstruct_Tidal
test_Tidal.cpp, 384
testConstruct_Wave
test_Wave.cpp, 390
testConstruct_Wind
test_Wind.cpp, 397

testConstructLookup_Diesel
test_Diesel.cpp, 360
testConstructLookup_Wave
test_Wave.cpp, 391
testDataRead1D_Interpolator
test_Interpolator.cpp, 418
testDataRead2D_Interpolator
test_Interpolator.cpp, 419
testDataRead_ElectricalLoad
test_ElectricalLoad.cpp, 413
testEconomics_Diesel
test_Diesel.cpp, 360
testEconomics_Model
test_Model.cpp, 442
testEconomics_Solar
test_Solar.cpp, 379
testEconomics_Tidal
test_Tidal.cpp, 385
testEconomics_Wave
test_Wave.cpp, 391
testEconomics_Wind
test_Wind.cpp, 398
testEfficiencyInterpolation_Hydro
test_Hydro.cpp, 369
testElectricalLoadData_Model
test_Model.cpp, 442
testFloatEquals
testing_utils.cpp, 461
testing_utils.h, 467
testFuelConsumptionEmissions_Diesel
test_Diesel.cpp, 361
testFuelConsumptionEmissions_Model
test_Model.cpp, 443
testFuelLookup_Diesel
test_Diesel.cpp, 363
testGreaterThan
testing_utils.cpp, 461
testing_utils.h, 468
testGreaterThanOrEqualTo
testing_utils.cpp, 462
testing_utils.h, 469
testing_utils.cpp
expectedErrorNotDetected, 459
printGold, 460
printGreen, 460
printRed, 460
testFloatEquals, 461
testGreaterThan, 461
testGreaterThanOrEqualTo, 462
testLessThan, 463
testLessThanOrEqualTo, 463
testTruth, 464
testing_utils.h
expectedErrorNotDetected, 466
FLOAT_TOLERANCE, 466
printGold, 466
printGreen, 467
printRed, 467
testFloatEquals, 467
testGreaterThan, 468
testGreaterThanOrEqualTo, 469
testLessThan, 469
testLessThanOrEqualTo, 470
testTruth, 470
testInterpolation1D_Interpolator
test_Interpolator.cpp, 422
testInterpolation2D_Interpolator
test_Interpolator.cpp, 423
testInvalidInterpolation1D_Interpolator
test_Interpolator.cpp, 424
testInvalidInterpolation2D_Interpolator
test_Interpolator.cpp, 425
testLessThan
testing_utils.cpp, 463
testing_utils.h, 469
testLessThanOrEqualTo
testing_utils.cpp, 463
testing_utils.h, 470
testLoadBalance_Model
test_Model.cpp, 444
testMinimumLoadRatioConstraint_Diesel
test_Diesel.cpp, 364
testMinimumRuntimeConstraint_Diesel
test_Diesel.cpp, 364
testPostConstructionAttributes_ElectricalLoad
test_ElectricalLoad.cpp, 415
testPostConstructionAttributes_Model
test_Model.cpp, 446
testProductionConstraint_Solar
test_Solar.cpp, 379
testProductionConstraint_Tidal
test_Tidal.cpp, 386
testProductionConstraint_Wave
test_Wave.cpp, 392
testProductionConstraint_Wind
test_Wind.cpp, 398
testProductionLookup_Wave
test_Wave.cpp, 392
testProductionOverride_Solar
test_Solar.cpp, 380
testTruth
testing_utils.cpp, 464
testing_utils.h, 470
TIDAL
Renewable.h, 288
Tidal, 229
__checkInputs, 233
__computeCubicProductionkW, 233
__computeExponentialProductionkW, 234
__computeLookupProductionkW, 235
__getGenericCapitalCost, 235
__getGenericOpMaintCost, 235
__writeSummary, 236
__writeTimeSeries, 237
∼Tidal, 233
commit, 238

computeProductionkW, 239
design_speed_ms, 240
handleReplacement, 240
power_model, 241
power_model_string, 241
Tidal, 231
Tidal.h
N_TIDAL_POWER_PRODUCTION_MODELS, 291
TIDAL_POWER_CUBIC, 291
TIDAL_POWER_EXPONENTIAL, 291
TIDAL_POWER_LOOKUP, 291
TidalPowerProductionModel, 290
TIDAL_POWER_CUBIC
Tidal.h, 291
TIDAL_POWER_EXPONENTIAL
Tidal.h, 291
TIDAL_POWER_LOOKUP
Tidal.h, 291
TidalInputs, 241
capital_cost, 242
design_speed_ms, 242
operation_maintenance_cost_kWh, 242
power_model, 243
renewable_inputs, 243
resource_key, 243
TidalPowerProductionModel
Tidal.h, 290
time_since_last_start_hrs
Diesel, 58
time_vec_hrs
ElectricalLoad, 67
total_discharge_kWh
Storage, 226
total_dispatch_discharge_kWh
Model, 151
total_dispatch_kWh
Production, 177
total_emissions
Combustion, 23
Model, 151
total_fuel_consumed_L
Combustion, 23
Model, 151
total_renewable_dispatch_kWh
Model, 152
TURBINE_EFFICIENCY_INTERP_KEY
Hydro.h, 284
turbine_flow_vec_m3hr
Hydro, 88
turbine_type
Hydro, 88
HydroInputs, 91
type
Combustion, 23
Noncombustion, 160
Renewable, 187
Storage, 226
type_str
Production, 178
Storage, 227

value
PYBIND11_Combustion.cpp, 305
PYBIND11_Controller.cpp, 329
PYBIND11_Hydro.cpp, 311
PYBIND11_Noncombustion.cpp, 312
PYBIND11_Renewable.cpp, 319
PYBIND11_Storage.cpp, 341
PYBIND11_Tidal.cpp, 322
PYBIND11_Wave.cpp, 324, 325
PYBIND11_Wind.cpp, 326

WAVE
Renewable.h, 288
Wave, 244
__checkInputs, 248
__computeGaussianProductionkW, 248
__computeLookupProductionkW, 249
__computeParaboloidProductionkW, 250
__getGenericCapitalCost, 250
__getGenericOpMaintCost, 251
__writeSummary, 251
__writeTimeSeries, 253
~Wave, 247
commit, 254
computeProductionkW, 254
design_energy_period_s, 256
design_significant_wave_height_m, 256
handleReplacement, 256
power_model, 257
power_model_string, 257
Wave, 246
Wave.h
N_WAVE_POWER_PRODUCTION_MODELS, 292
WAVE_POWER_GAUSSIAN, 292
WAVE_POWER_LOOKUP, 292
WAVE_POWER_PARABOLOID, 292
WavePowerProductionModel, 292
WAVE_POWER_GAUSSIAN
Wave.h, 292
WAVE_POWER_LOOKUP
Wave.h, 292
WAVE_POWER_PARABOLOID
Wave.h, 292
WaveInputs, 257
capital_cost, 258
design_energy_period_s, 258
design_significant_wave_height_m, 258
operation_maintenance_cost_kWh, 259
path_2_normalized_performance_matrix, 259
power_model, 259
renewable_inputs, 259
resource_key, 259
WavePowerProductionModel
Wave.h, 292

WIND
    Renewable.h, 288
Wind, 260
    __checkInputs, 263
    __computeCubicProductionkW, 264
    __computeExponentialProductionkW, 265
    __computeLookupProductionkW, 265
    __getGenericCapitalCost, 266
    __getGenericOpMaintCost, 266
    __writeSummary, 266
    __writeTimeSeries, 268
    ∼Wind, 263
    commit, 269
    computeProductionkW, 269
    design_speed_ms, 271
    handleReplacement, 271
    power_model, 271
    power_model_string, 271
    Wind, 262
Wind.h
    N_WIND_POWER_PRODUCTION_MODELS, 293
    WIND_POWER_CUBIC, 293
    WIND_POWER_EXPONENTIAL, 293
    WIND_POWER_LOOKUP, 293
    WindPowerProductionModel, 293
WIND_POWER_CUBIC
    Wind.h, 293
WIND_POWER_EXPONENTIAL
    Wind.h, 293
WIND_POWER_LOOKUP
    Wind.h, 293
WindInputs, 272
    capital_cost, 273
    design_speed_ms, 273
    operation_maintenance_cost_kWh, 273
    power_model, 273
    renewable_inputs, 274
    resource_key, 274
WindPowerProductionModel
    Wind.h, 293
writeResults
    Combustion, 18
    Model, 148
    Noncombustion, 159
    Renewable, 186
    Storage, 221

x_vec
    InterpolatorStruct1D, 106
    InterpolatorStruct2D, 108

y_vec
    InterpolatorStruct1D, 106
    InterpolatorStruct2D, 108

z_matrix
    InterpolatorStruct2D, 108