

PGMcpp: PRIMED Grid Modelling (in C++)

Generated by Doxygen 1.9.1

1 Hierarchical Index	1
1.1 Class Hierarchy	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation	7
4.1 Combustion Class Reference	7
4.1.1 Detailed Description	9
4.1.2 Constructor & Destructor Documentation	9
4.1.2.1 Combustion() [1/2]	10
4.1.2.2 Combustion() [2/2]	10
4.1.2.3 ~Combustion()	10
4.1.3 Member Function Documentation	11
4.1.3.1 __checkInputs()	11
4.1.3.2 commit()	11
4.1.3.3 computeEconomics()	12
4.1.3.4 computeFuelAndEmissions()	13
4.1.3.5 getEmissionskg()	13
4.1.3.6 getFuelConsumptionL()	13
4.1.3.7 requestProductionkW()	14
4.1.4 Member Data Documentation	14
4.1.4.1 CH4_emissions_intensity_kgL	14
4.1.4.2 CH4_emissions_vec_kg	14
4.1.4.3 CO2_emissions_intensity_kgL	15
4.1.4.4 CO2_emissions_vec_kg	15
4.1.4.5 CO_emissions_intensity_kgL	15
4.1.4.6 CO_emissions_vec_kg	15
4.1.4.7 fuel_consumption_vec_L	15
4.1.4.8 fuel_cost_L	15
4.1.4.9 fuel_cost_vec	16
4.1.4.10 linear_fuel_intercept_LkWh	16
4.1.4.11 linear_fuel_slope_LkWh	16
4.1.4.12 NOx_emissions_intensity_kgL	16
4.1.4.13 NOx_emissions_vec_kg	16
4.1.4.14 PM_emissions_intensity_kgL	16
4.1.4.15 PM_emissions_vec_kg	17
4.1.4.16 SOx_emissions_intensity_kgL	17
4.1.4.17 SOx_emissions_vec_kg	17
4.1.4.18 total_emissions	17

4.1.4.19 total_fuel_consumed_L	17
4.1.4.20 type	17
4.2 CombustionInputs Struct Reference	18
4.2.1 Detailed Description	18
4.2.2 Member Data Documentation	18
4.2.2.1 production_inputs	18
4.3 Controller Class Reference	19
4.3.1 Detailed Description	20
4.3.2 Constructor & Destructor Documentation	20
4.3.2.1 Controller()	20
4.3.2.2 ~Controller()	20
4.3.3 Member Function Documentation	20
4.3.3.1 __applyCycleChargingControl_CHARGING()	21
4.3.3.2 __applyCycleChargingControl_DISCHARGING()	21
4.3.3.3 __applyLoadFollowingControl_CHARGING()	22
4.3.3.4 __applyLoadFollowingControl_DISCHARGING()	23
4.3.3.5 __computeNetLoad()	24
4.3.3.6 __constructCombustionMap()	25
4.3.3.7 __getRenewableProduction()	26
4.3.3.8 __handleCombustionDispatch()	27
4.3.3.9 __handleStorageCharging() [1/2]	28
4.3.3.10 __handleStorageCharging() [2/2]	29
4.3.3.11 __handleStorageDischarging()	29
4.3.3.12 applyDispatchControl()	30
4.3.3.13 clear()	31
4.3.3.14 init()	31
4.3.4 Member Data Documentation	32
4.3.4.1 combustion_map	32
4.3.4.2 control_mode	32
4.3.4.3 missed_load_vec_kW	32
4.3.4.4 net_load_vec_kW	32
4.4 Diesel Class Reference	33
4.4.1 Detailed Description	34
4.4.2 Constructor & Destructor Documentation	34
4.4.2.1 Diesel() [1/2]	34
4.4.2.2 Diesel() [2/2]	35
4.4.2.3 ~Diesel()	36
4.4.3 Member Function Documentation	36
4.4.3.1 __checkInputs()	36
4.4.3.2 __getGenericCapitalCost()	38
4.4.3.3 __getGenericFuelIntercept()	38
4.4.3.4 __getGenericFuelSlope()	38

4.4.3.5 __getGenericOpMaintCost()	39
4.4.3.6 __handleStartStop()	39
4.4.3.7 commit()	40
4.4.3.8 requestProductionkW()	40
4.4.4 Member Data Documentation	41
4.4.4.1 minimum_load_ratio	41
4.4.4.2 minimum_runtime_hrs	41
4.4.4.3 time_since_last_start_hrs	42
4.5 DieselInputs Struct Reference	42
4.5.1 Detailed Description	43
4.5.2 Member Data Documentation	43
4.5.2.1 capital_cost	44
4.5.2.2 CH4_emissions_intensity_kgL	44
4.5.2.3 CO2_emissions_intensity_kgL	44
4.5.2.4 CO_emissions_intensity_kgL	44
4.5.2.5 combustion_inputs	44
4.5.2.6 fuel_cost_L	44
4.5.2.7 linear_fuel_intercept_LkWh	45
4.5.2.8 linear_fuel_slope_LkWh	45
4.5.2.9 minimum_load_ratio	45
4.5.2.10 minimum_runtime_hrs	45
4.5.2.11 NOx_emissions_intensity_kgL	45
4.5.2.12 operation_maintenance_cost_kWh	46
4.5.2.13 PM_emissions_intensity_kgL	46
4.5.2.14 replace_running_hrs	46
4.5.2.15 SOx_emissions_intensity_kgL	46
4.6 ElectricalLoad Class Reference	46
4.6.1 Detailed Description	47
4.6.2 Constructor & Destructor Documentation	47
4.6.2.1 ElectricalLoad() [1/2]	48
4.6.2.2 ElectricalLoad() [2/2]	48
4.6.2.3 ~ElectricalLoad()	48
4.6.3 Member Function Documentation	48
4.6.3.1 clear()	48
4.6.3.2 readLoadData()	49
4.6.4 Member Data Documentation	50
4.6.4.1 dt_vec_hrs	50
4.6.4.2 load_vec_kW	50
4.6.4.3 max_load_kW	50
4.6.4.4 mean_load_kW	50
4.6.4.5 min_load_kW	51
4.6.4.6 n_points	51

4.6.4.7 n_years	51
4.6.4.8 path_2_electrical_load_time_series	51
4.6.4.9 time_vec_hrs	51
4.7 Emissions Struct Reference	51
4.7.1 Detailed Description	52
4.7.2 Member Data Documentation	52
4.7.2.1 CH4_kg	52
4.7.2.2 CO2_kg	52
4.7.2.3 CO_kg	52
4.7.2.4 NOx_kg	53
4.7.2.5 PM_kg	53
4.7.2.6 SOx_kg	53
4.8 Lilon Class Reference	53
4.8.1 Detailed Description	54
4.8.2 Constructor & Destructor Documentation	54
4.8.2.1 Lilon()	54
4.8.2.2 ~Lilon()	55
4.9 Model Class Reference	55
4.9.1 Detailed Description	57
4.9.2 Constructor & Destructor Documentation	57
4.9.2.1 Model() [1/2]	57
4.9.2.2 Model() [2/2]	57
4.9.2.3 ~Model()	57
4.9.3 Member Function Documentation	58
4.9.3.1 __checkInputs()	58
4.9.3.2 __computeEconomics()	58
4.9.3.3 __computeFuelAndEmissions()	58
4.9.3.4 __computeLevellizedCostOfEnergy()	59
4.9.3.5 __computeNetPresentCost()	59
4.9.3.6 addDiesel()	60
4.9.3.7 addResource()	60
4.9.3.8 addSolar()	61
4.9.3.9 addTidal()	61
4.9.3.10 addWave()	62
4.9.3.11 addWind()	62
4.9.3.12 clear()	62
4.9.3.13 reset()	63
4.9.3.14 run()	63
4.9.4 Member Data Documentation	64
4.9.4.1 combustion_ptr_vec	64
4.9.4.2 controller	64
4.9.4.3 electrical_load	64

4.9.4.4	levellized_cost_of_energy_kWh	64
4.9.4.5	net_present_cost	65
4.9.4.6	renewable_ptr_vec	65
4.9.4.7	resources	65
4.9.4.8	storage_ptr_vec	65
4.9.4.9	total_dispatch_discharge_kWh	65
4.9.4.10	total_emissions	65
4.9.4.11	total_fuel_consumed_L	66
4.10	ModelInputs Struct Reference	66
4.10.1	Detailed Description	66
4.10.2	Member Data Documentation	66
4.10.2.1	control_mode	66
4.10.2.2	path_2_electrical_load_time_series	67
4.11	Production Class Reference	67
4.11.1	Detailed Description	69
4.11.2	Constructor & Destructor Documentation	69
4.11.2.1	Production() [1/2]	69
4.11.2.2	Production() [2/2]	69
4.11.2.3	~Production()	70
4.11.3	Member Function Documentation	70
4.11.3.1	__checkInputs()	70
4.11.3.2	__computeRealDiscountAnnual()	71
4.11.3.3	__handleReplacement()	72
4.11.3.4	commit()	72
4.11.3.5	computeEconomics()	73
4.11.4	Member Data Documentation	74
4.11.4.1	capacity_kW	74
4.11.4.2	capital_cost	74
4.11.4.3	capital_cost_vec	74
4.11.4.4	curtailment_vec_kW	75
4.11.4.5	dispatch_vec_kW	75
4.11.4.6	is_running	75
4.11.4.7	is_running_vec	75
4.11.4.8	is_sunk	75
4.11.4.9	levellized_cost_of_energy_kWh	75
4.11.4.10	n_points	76
4.11.4.11	n_replacements	76
4.11.4.12	n_starts	76
4.11.4.13	net_present_cost	76
4.11.4.14	operation_maintenance_cost_kWh	76
4.11.4.15	operation_maintenance_cost_vec	76
4.11.4.16	print_flag	77

4.11.4.17 production_vec_kW	77
4.11.4.18 real_discount_annual	77
4.11.4.19 replace_running_hrs	77
4.11.4.20 running_hours	77
4.11.4.21 storage_vec_kW	77
4.11.4.22 total_dispatch_kWh	78
4.11.4.23 type_str	78
4.12 ProductionInputs Struct Reference	78
4.12.1 Detailed Description	78
4.12.2 Member Data Documentation	79
4.12.2.1 capacity_kW	79
4.12.2.2 is_sunk	79
4.12.2.3 nominal_discount_annual	79
4.12.2.4 nominal_inflation_annual	79
4.12.2.5 print_flag	79
4.12.2.6 replace_running_hrs	80
4.13 Renewable Class Reference	80
4.13.1 Detailed Description	81
4.13.2 Constructor & Destructor Documentation	81
4.13.2.1 Renewable() [1/2]	81
4.13.2.2 Renewable() [2/2]	82
4.13.2.3 ~Renewable()	82
4.13.3 Member Function Documentation	82
4.13.3.1 __checkInputs()	83
4.13.3.2 __handleStartStop()	83
4.13.3.3 commit()	83
4.13.3.4 computeEconomics()	84
4.13.3.5 computeProductionkW() [1/2]	84
4.13.3.6 computeProductionkW() [2/2]	85
4.13.4 Member Data Documentation	85
4.13.4.1 resource_key	85
4.13.4.2 type	85
4.14 RenewableInputs Struct Reference	85
4.14.1 Detailed Description	86
4.14.2 Member Data Documentation	86
4.14.2.1 production_inputs	86
4.15 Resources Class Reference	86
4.15.1 Detailed Description	87
4.15.2 Constructor & Destructor Documentation	87
4.15.2.1 Resources()	87
4.15.2.2 ~Resources()	88
4.15.3 Member Function Documentation	88

4.15.3.1 __checkResourceKey1D()	88
4.15.3.2 __checkResourceKey2D()	89
4.15.3.3 __checkTimePoint()	89
4.15.3.4 __readSolarResource()	90
4.15.3.5 __readTidalResource()	91
4.15.3.6 __readWaveResource()	92
4.15.3.7 __readWindResource()	93
4.15.3.8 __throwLengthError()	94
4.15.3.9 addResource()	94
4.15.3.10 clear()	95
4.15.4 Member Data Documentation	96
4.15.4.1 path_map_1D	96
4.15.4.2 path_map_2D	96
4.15.4.3 resource_map_1D	96
4.15.4.4 resource_map_2D	96
4.16 Solar Class Reference	97
4.16.1 Detailed Description	98
4.16.2 Constructor & Destructor Documentation	98
4.16.2.1 Solar() [1/2]	98
4.16.2.2 Solar() [2/2]	99
4.16.2.3 ~Solar()	99
4.16.3 Member Function Documentation	99
4.16.3.1 __checkInputs()	100
4.16.3.2 __getGenericCapitalCost()	100
4.16.3.3 __getGenericOpMaintCost()	100
4.16.3.4 commit()	101
4.16.3.5 computeProductionkW()	101
4.16.4 Member Data Documentation	102
4.16.4.1 derating	102
4.17 SolarInputs Struct Reference	102
4.17.1 Detailed Description	103
4.17.2 Member Data Documentation	103
4.17.2.1 capital_cost	104
4.17.2.2 derating	104
4.17.2.3 operation_maintenance_cost_kWh	104
4.17.2.4 renewable_inputs	104
4.17.2.5 resource_key	104
4.18 Storage Class Reference	105
4.18.1 Detailed Description	105
4.18.2 Constructor & Destructor Documentation	105
4.18.2.1 Storage()	105
4.18.2.2 ~Storage()	106

4.19 Tidal Class Reference	106
4.19.1 Detailed Description	108
4.19.2 Constructor & Destructor Documentation	108
4.19.2.1 Tidal() [1/2]	108
4.19.2.2 Tidal() [2/2]	108
4.19.2.3 ~Tidal()	109
4.19.3 Member Function Documentation	109
4.19.3.1 __checkInputs()	109
4.19.3.2 __computeCubicProductionkW()	110
4.19.3.3 __computeExponentialProductionkW()	110
4.19.3.4 __computeLookupProductionkW()	111
4.19.3.5 __getGenericCapitalCost()	112
4.19.3.6 __getGenericOpMaintCost()	112
4.19.3.7 commit()	112
4.19.3.8 computeProductionkW()	113
4.19.4 Member Data Documentation	114
4.19.4.1 design_speed_ms	114
4.19.4.2 power_model	114
4.20 TidalInputs Struct Reference	115
4.20.1 Detailed Description	115
4.20.2 Member Data Documentation	116
4.20.2.1 capital_cost	116
4.20.2.2 design_speed_ms	116
4.20.2.3 operation_maintenance_cost_kWh	116
4.20.2.4 power_model	116
4.20.2.5 renewable_inputs	116
4.20.2.6 resource_key	117
4.21 Wave Class Reference	117
4.21.1 Detailed Description	118
4.21.2 Constructor & Destructor Documentation	118
4.21.2.1 Wave() [1/2]	118
4.21.2.2 Wave() [2/2]	119
4.21.2.3 ~Wave()	119
4.21.3 Member Function Documentation	120
4.21.3.1 __checkInputs()	120
4.21.3.2 __computeGaussianProductionkW()	120
4.21.3.3 __computeLookupProductionkW()	121
4.21.3.4 __computeParaboloidProductionkW()	121
4.21.3.5 __getGenericCapitalCost()	123
4.21.3.6 __getGenericOpMaintCost()	124
4.21.3.7 commit()	124
4.21.3.8 computeProductionkW()	125

4.21.4 Member Data Documentation	126
4.21.4.1 design_energy_period_s	126
4.21.4.2 design_significant_wave_height_m	126
4.21.4.3 power_model	126
4.22 WaveInputs Struct Reference	127
4.22.1 Detailed Description	128
4.22.2 Member Data Documentation	128
4.22.2.1 capital_cost	128
4.22.2.2 design_energy_period_s	128
4.22.2.3 design_significant_wave_height_m	128
4.22.2.4 operation_maintenance_cost_kWh	128
4.22.2.5 power_model	129
4.22.2.6 renewable_inputs	129
4.22.2.7 resource_key	129
4.23 Wind Class Reference	129
4.23.1 Detailed Description	131
4.23.2 Constructor & Destructor Documentation	131
4.23.2.1 Wind() [1/2]	131
4.23.2.2 Wind() [2/2]	131
4.23.2.3 ~Wind()	132
4.23.3 Member Function Documentation	132
4.23.3.1 __checkInputs()	132
4.23.3.2 __computeExponentialProductionkW()	132
4.23.3.3 __computeLookupProductionkW()	133
4.23.3.4 __getGenericCapitalCost()	134
4.23.3.5 __getGenericOpMaintCost()	134
4.23.3.6 commit()	134
4.23.3.7 computeProductionkW()	135
4.23.4 Member Data Documentation	136
4.23.4.1 design_speed_ms	136
4.23.4.2 power_model	136
4.24 WindInputs Struct Reference	137
4.24.1 Detailed Description	137
4.24.2 Member Data Documentation	138
4.24.2.1 capital_cost	138
4.24.2.2 design_speed_ms	138
4.24.2.3 operation_maintenance_cost_kWh	138
4.24.2.4 power_model	138
4.24.2.5 renewable_inputs	138
4.24.2.6 resource_key	138

5.1 header/Controller.h File Reference	139
5.1.1 Detailed Description	140
5.1.2 Enumeration Type Documentation	140
5.1.2.1 ControlMode	140
5.2 header/ElectricalLoad.h File Reference	140
5.2.1 Detailed Description	141
5.3 header/Model.h File Reference	141
5.3.1 Detailed Description	142
5.4 header/Production/Combustion/Combustion.h File Reference	142
5.4.1 Detailed Description	143
5.4.2 Enumeration Type Documentation	143
5.4.2.1 CombustionType	143
5.5 header/Production/Combustion/Diesel.h File Reference	144
5.5.1 Detailed Description	145
5.6 header/Production/Production.h File Reference	145
5.6.1 Detailed Description	145
5.7 header/Production/Renewable/Renewable.h File Reference	146
5.7.1 Detailed Description	146
5.7.2 Enumeration Type Documentation	146
5.7.2.1 RenewableType	146
5.8 header/Production/Renewable/Solar.h File Reference	147
5.8.1 Detailed Description	148
5.9 header/Production/Renewable/Tidal.h File Reference	148
5.9.1 Detailed Description	149
5.9.2 Enumeration Type Documentation	149
5.9.2.1 TidalPowerProductionModel	149
5.10 header/Production/Renewable/Wave.h File Reference	149
5.10.1 Detailed Description	150
5.10.2 Enumeration Type Documentation	150
5.10.2.1 WavePowerProductionModel	150
5.11 header/Production/Renewable/Wind.h File Reference	151
5.11.1 Detailed Description	152
5.11.2 Enumeration Type Documentation	152
5.11.2.1 WindPowerProductionModel	152
5.12 header/Resources.h File Reference	152
5.12.1 Detailed Description	153
5.13 header/std_includes.h File Reference	153
5.13.1 Detailed Description	154
5.14 header/Storage/Lilon.h File Reference	154
5.14.1 Detailed Description	154
5.15 header/Storage/Storage.h File Reference	155
5.15.1 Detailed Description	155

5.16 pybindings/PYBIND11_PGM.cpp File Reference	155
5.16.1 Detailed Description	156
5.16.2 Function Documentation	156
5.16.2.1 PYBIND11_MODULE()	156
5.17 source/Controller.cpp File Reference	157
5.17.1 Detailed Description	157
5.18 source/ElectricalLoad.cpp File Reference	158
5.18.1 Detailed Description	158
5.19 source/Model.cpp File Reference	158
5.19.1 Detailed Description	158
5.20 source/Production/Combustion/Combustion.cpp File Reference	159
5.20.1 Detailed Description	159
5.21 source/Production/Combustion/Diesel.cpp File Reference	159
5.21.1 Detailed Description	159
5.22 source/Production/Production.cpp File Reference	160
5.22.1 Detailed Description	160
5.23 source/Production/Renewable/Renewable.cpp File Reference	160
5.23.1 Detailed Description	160
5.24 source/Production/Renewable/Solar.cpp File Reference	161
5.24.1 Detailed Description	161
5.25 source/Production/Renewable/Tidal.cpp File Reference	161
5.25.1 Detailed Description	161
5.26 source/Production/Renewable/Wave.cpp File Reference	162
5.26.1 Detailed Description	162
5.27 source/Production/Renewable/Wind.cpp File Reference	162
5.27.1 Detailed Description	162
5.28 source/Resources.cpp File Reference	163
5.28.1 Detailed Description	163
5.29 source/Storage/Lilon.cpp File Reference	163
5.29.1 Detailed Description	163
5.30 source/Storage/Storage.cpp File Reference	164
5.30.1 Detailed Description	164
5.31 test/source/Production/Combustion/test_Combustion.cpp File Reference	164
5.31.1 Detailed Description	165
5.31.2 Function Documentation	165
5.31.2.1 main()	165
5.32 test/source/Production/Combustion/test_Diesel.cpp File Reference	166
5.32.1 Detailed Description	167
5.32.2 Function Documentation	167
5.32.2.1 main()	167
5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference	172
5.33.1 Detailed Description	172

5.33.2 Function Documentation	172
5.33.2.1 main()	172
5.34 test/source/Production/Renewable/test_Solar.cpp File Reference	173
5.34.1 Detailed Description	174
5.34.2 Function Documentation	174
5.34.2.1 main()	174
5.35 test/source/Production/Renewable/test_Tidal.cpp File Reference	177
5.35.1 Detailed Description	177
5.35.2 Function Documentation	177
5.35.2.1 main()	178
5.36 test/source/Production/Renewable/test_Wave.cpp File Reference	180
5.36.1 Detailed Description	181
5.36.2 Function Documentation	181
5.36.2.1 main()	181
5.37 test/source/Production/Renewable/test_Wind.cpp File Reference	184
5.37.1 Detailed Description	185
5.37.2 Function Documentation	185
5.37.2.1 main()	185
5.38 test/source/Production/test_Production.cpp File Reference	188
5.38.1 Detailed Description	188
5.38.2 Function Documentation	188
5.38.2.1 main()	189
5.39 test/source/Storage/test_Lilon.cpp File Reference	190
5.39.1 Detailed Description	191
5.39.2 Function Documentation	191
5.39.2.1 main()	191
5.40 test/source/Storage/test_Storage.cpp File Reference	192
5.40.1 Detailed Description	192
5.40.2 Function Documentation	192
5.40.2.1 main()	192
5.41 test/source/test_Controller.cpp File Reference	193
5.41.1 Detailed Description	193
5.41.2 Function Documentation	193
5.41.2.1 main()	193
5.42 test/source/test_ElectricalLoad.cpp File Reference	194
5.42.1 Detailed Description	194
5.42.2 Function Documentation	195
5.42.2.1 main()	195
5.43 test/source/test_Model.cpp File Reference	197
5.43.1 Detailed Description	197
5.43.2 Function Documentation	197
5.43.2.1 main()	198

5.44 test/source/test_Resources.cpp File Reference	204
5.44.1 Detailed Description	204
5.44.2 Function Documentation	204
5.44.2.1 main()	205
5.45 test/utls/testing_utls.cpp File Reference	210
5.45.1 Detailed Description	211
5.45.2 Function Documentation	211
5.45.2.1 expectedErrorNotDetected()	211
5.45.2.2 printGold()	212
5.45.2.3 printGreen()	212
5.45.2.4 printRed()	212
5.45.2.5 testFloatEquals()	213
5.45.2.6 testGreaterThan()	213
5.45.2.7 testGreaterThanOrEqualTo()	214
5.45.2.8 testLessThan()	215
5.45.2.9 testLessThanOrEqualTo()	215
5.45.2.10 testTruth()	216
5.46 test/utls/testing_utls.h File Reference	216
5.46.1 Detailed Description	217
5.46.2 Macro Definition Documentation	218
5.46.2.1 FLOAT_TOLERANCE	218
5.46.3 Function Documentation	218
5.46.3.1 expectedErrorNotDetected()	218
5.46.3.2 printGold()	218
5.46.3.3 printGreen()	219
5.46.3.4 printRed()	219
5.46.3.5 testFloatEquals()	219
5.46.3.6 testGreaterThan()	220
5.46.3.7 testGreaterThanOrEqualTo()	221
5.46.3.8 testLessThan()	221
5.46.3.9 testLessThanOrEqualTo()	222
5.46.3.10 testTruth()	222
Bibliography	225
Index	227

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

CombustionInputs	18
Controller	19
DieselInputs	42
ElectricalLoad	46
Emissions	51
Model	55
ModelInputs	66
Production	67
Combustion	7
Diesel	33
Renewable	80
Solar	97
Tidal	106
Wave	117
Wind	129
ProductionInputs	78
RenewableInputs	85
Resources	86
SolarInputs	102
Storage	105
Lilon	53
TidalInputs	115
WaveInputs	127
WindInputs	137

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Combustion	The root of the Combustion branch of the Production hierarchy. This branch contains derived classes which model the production of energy by way of combustibles	7
CombustionInputs	A structure which bundles the necessary inputs for the Combustion constructor. Provides default values for every necessary input. Note that this structure encapsulates ProductionInputs . . .	18
Controller	A class which contains a various dispatch control logic. Intended to serve as a component class of Model	19
Diesel	A derived class of the Combustion branch of Production which models production using a diesel generator	33
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 . . .	42
ElectricalLoad	A class which contains time and electrical load data. Intended to serve as a component class of Model	46
Emissions	A structure which bundles the emitted masses of various emissions chemistries	51
Lilon	A derived class of Storage which models energy storage by way of lithium-ion batteries	53
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	55
ModelInputs	A structure which bundles the necessary inputs for the Model constructor. Provides default values for every necessary input (except <code>path_2_electrical_load_time_series</code> , for which a valid input must be provided)	66
Production	The base class of the Production hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise	67
ProductionInputs	A structure which bundles the necessary inputs for the Production constructor. Provides default values for every necessary input	78

Renewable	The root of the Renewable branch of the Production hierarchy. This branch contains derived classes which model the renewable production of energy	80
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 . . .	85
Resources	A class which contains renewable resource data. Intended to serve as a component class of Model	86
Solar	A derived class of the Renewable branch of Production which models solar production	97
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	102
Storage	The base class of the Storage hierarchy. This hierarchy contains derived classes which model the storage of energy	105
Tidal	A derived class of the Renewable branch of Production which models tidal production	106
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	115
Wave	A derived class of the Renewable branch of Production which models wave production	117
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	127
Wind	A derived class of the Renewable branch of Production which models wind production	129
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	137

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

header/ Controller.h	
Header file the Controller class	139
header/ ElectricalLoad.h	
Header file the ElectricalLoad class	140
header/ Model.h	
Header file the Model class	141
header/ Resources.h	
Header file the Resources class	152
header/ std_includes.h	
Header file which simply batches together the usual, standard includes	153
header/Production/ Production.h	
Header file the Production class	145
header/Production/Combustion/ Combustion.h	
Header file the Combustion class	142
header/Production/Combustion/ Diesel.h	
Header file the Diesel class	144
header/Production/Renewable/ Renewable.h	
Header file the Renewable class	146
header/Production/Renewable/ Solar.h	
Header file the Solar class	147
header/Production/Renewable/ Tidal.h	
Header file the Tidal class	148
header/Production/Renewable/ Wave.h	
Header file the Wave class	149
header/Production/Renewable/ Wind.h	
Header file the Wind class	151
header/Storage/ Lilon.h	
Header file the Lilon class	154
header/Storage/ Storage.h	
Header file the Storage class	155
pybindings/ PYBIND11_PGM.cpp	
Python 3 bindings file for PGMcpp	155
source/ Controller.cpp	
Implementation file for the Controller class	157
source/ ElectricalLoad.cpp	
Implementation file for the ElectricalLoad class	158

source/ Model.cpp	Implementation file for the Model class	158
source/ Resources.cpp	Implementation file for the Resources class	163
source/Production/ Production.cpp	Implementation file for the Production class	160
source/Production/Combustion/ Combustion.cpp	Implementation file for the Combustion class	159
source/Production/Combustion/ Diesel.cpp	Implementation file for the Diesel class	159
source/Production/Renewable/ Renewable.cpp	Implementation file for the Renewable class	160
source/Production/Renewable/ Solar.cpp	Implementation file for the Solar class	161
source/Production/Renewable/ Tidal.cpp	Implementation file for the Tidal class	161
source/Production/Renewable/ Wave.cpp	Implementation file for the Wave class	162
source/Production/Renewable/ Wind.cpp	Implementation file for the Wind class	162
source/Storage/ Lilon.cpp	Implementation file for the Lilon class	163
source/Storage/ Storage.cpp	Implementation file for the Storage class	164
test/source/ test_Controller.cpp	Testing suite for Controller class	193
test/source/ test_ElectricalLoad.cpp	Testing suite for ElectricalLoad class	194
test/source/ test_Model.cpp	Testing suite for Model class	197
test/source/ test_Resources.cpp	Testing suite for Resources class	204
test/source/Production/ test_Production.cpp	Testing suite for Production class	188
test/source/Production/Combustion/ test_Combustion.cpp	Testing suite for Combustion class	164
test/source/Production/Combustion/ test_Diesel.cpp	Testing suite for Diesel class	166
test/source/Production/Renewable/ test_Renewable.cpp	Testing suite for Renewable class	172
test/source/Production/Renewable/ test_Solar.cpp	Testing suite for Solar class	173
test/source/Production/Renewable/ test_Tidal.cpp	Testing suite for Tidal class	177
test/source/Production/Renewable/ test_Wave.cpp	Testing suite for Wave class	180
test/source/Production/Renewable/ test_Wind.cpp	Testing suite for Wind class	184
test/source/Storage/ test_Lilon.cpp	Testing suite for Lilon class	190
test/source/Storage/ test_Storage.cpp	Testing suite for Storage class	192
test/utills/ testing_utils.cpp	Header file for various PGMcpp testing utilities	210
test/utills/ testing_utils.h	Header file for various PGMcpp testing utilities	216

Chapter 4

Class Documentation

4.1 Combustion Class Reference

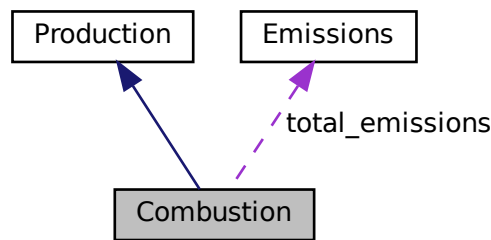
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

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

Inheritance diagram for Combustion:



Collaboration diagram for Combustion:



Public Member Functions

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

Public Attributes

- [CombustionType](#) type
The type ([CombustionType](#)) of the asset.
- double [fuel_cost_L](#)
The cost of fuel [1/L] (undefined currency).
- double [linear_fuel_slope_LkWh](#)
The slope [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double [linear_fuel_intercept_LkWh](#)
The intercept [L/kWh] to use in computing linearized fuel consumption. This is fuel consumption per unit energy produced.
- double [CO2_emissions_intensity_kgL](#)
Carbon dioxide (CO2) emissions intensity [kg/L].

- double [CO_emissions_intensity_kgL](#)
Carbon monoxide (CO) emissions intensity [kg/L].
- double [NOx_emissions_intensity_kgL](#)
Nitrogen oxide (NOx) emissions intensity [kg/L].
- double [SOx_emissions_intensity_kgL](#)
Sulfur oxide (SOx) emissions intensity [kg/L].
- double [CH4_emissions_intensity_kgL](#)
Methane (CH4) emissions intensity [kg/L].
- double [PM_emissions_intensity_kgL](#)
Particulate Matter (PM) emissions intensity [kg/L].
- double [total_fuel_consumed_L](#)
The total fuel consumed [L] over a model run.
- [Emissions total_emissions](#)
An [Emissions](#) structure for holding total emissions [kg].
- `std::vector< double >` [fuel_consumption_vec_L](#)
A vector of fuel consumed [L] over each modelling time step.
- `std::vector< double >` [fuel_cost_vec](#)
A vector of fuel costs (undefined currency) incurred over each modelling time step. These costs are not discounted (i.e., these are nominal costs).
- `std::vector< double >` [CO2_emissions_vec_kg](#)
A vector of carbon dioxide (CO2) emitted [kg] over each modelling time step.
- `std::vector< double >` [CO_emissions_vec_kg](#)
A vector of carbon monoxide (CO) emitted [kg] over each modelling time step.
- `std::vector< double >` [NOx_emissions_vec_kg](#)
A vector of nitrogen oxide (NOx) emitted [kg] over each modelling time step.
- `std::vector< double >` [SOx_emissions_vec_kg](#)
A vector of sulfur oxide (SOx) emitted [kg] over each modelling time step.
- `std::vector< double >` [CH4_emissions_vec_kg](#)
A vector of methane (CH4) emitted [kg] over each modelling time step.
- `std::vector< double >` [PM_emissions_vec_kg](#)
A vector of particulate matter (PM) emitted [kg] over each modelling time step.

Private Member Functions

- void [__checkInputs](#) ([CombustionInputs](#))
Helper method to check inputs to the [Combustion](#) constructor.

4.1.1 Detailed Description

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

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Combustion() [1/2]

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

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

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

4.1.2.2 Combustion() [2/2]

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

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

Parameters

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

```
83
84 Production(n_points, combustion_inputs.production_inputs)
85 {
86     // 1. check inputs
87     this->__checkInputs(combustion_inputs);
88
89     // 2. set attributes
90     this->fuel_cost_L = 0;
91
92     this->linear_fuel_slope_LkWh = 0;
93     this->linear_fuel_intercept_LkWh = 0;
94
95     this->CO2_emissions_intensity_kgL = 0;
96     this->CO_emissions_intensity_kgL = 0;
97     this->NOx_emissions_intensity_kgL = 0;
98     this->SOx_emissions_intensity_kgL = 0;
99     this->CH4_emissions_intensity_kgL = 0;
100     this->PM_emissions_intensity_kgL = 0;
101
102     this->total_fuel_consumed_L = 0;
103
104     this->fuel_consumption_vec_L.resize(this->n_points, 0);
105     this->fuel_cost_vec.resize(this->n_points, 0);
106
107     this->CO2_emissions_vec_kg.resize(this->n_points, 0);
108     this->CO_emissions_vec_kg.resize(this->n_points, 0);
109     this->NOx_emissions_vec_kg.resize(this->n_points, 0);
110     this->SOx_emissions_vec_kg.resize(this->n_points, 0);
111     this->CH4_emissions_vec_kg.resize(this->n_points, 0);
112     this->PM_emissions_vec_kg.resize(this->n_points, 0);
113
114     // 3. construction print
115     if (this->print_flag) {
116         std::cout << "Combustion object constructed at " << this << std::endl;
117     }
118
119     return;
120 } /* Combustion() */
```

4.1.2.3 ~Combustion()

```
Combustion::~Combustion (
```

```
void ) [virtual]
```

Destructor for the [Combustion](#) class.

```
325 {
326     // 1. destruction print
327     if (this->print_flag) {
328         std::cout << "Combustion object at " << this << " destroyed" << std::endl;
329     }
330
331     return;
332 } /* ~Combustion() */
```

4.1.3 Member Function Documentation

4.1.3.1 __checkInputs()

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

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

Parameters

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

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

4.1.3.2 commit()

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

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

Parameters

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

Returns

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

Reimplemented from [Production](#).

Reimplemented in [Diesel](#).

```

222 {
223     // 1. invoke base class method
224     load_kW = Production::commit(
225         timestep,
226         dt_hrs,
227         production_kW,
228         load_kW
229     );
230
231
232     if (this->is_running) {
233         // 2. compute and record fuel consumption
234         double fuel_consumed_L = this->getFuelConsumptionL(dt_hrs, production_kW);
235         this->fuel_consumption_vec_L[timestep] = fuel_consumed_L;
236
237         // 3. compute and record emissions
238         Emissions emissions = this->getEmissionskg(fuel_consumed_L);
239         this->CO2_emissions_vec_kg[timestep] = emissions.CO2_kg;
240         this->CO_emissions_vec_kg[timestep] = emissions.CO_kg;
241         this->NOx_emissions_vec_kg[timestep] = emissions.NOx_kg;
242         this->SOx_emissions_vec_kg[timestep] = emissions.SOx_kg;
243         this->CH4_emissions_vec_kg[timestep] = emissions.CH4_kg;
244         this->PM_emissions_vec_kg[timestep] = emissions.PM_kg;
245
246         // 4. incur fuel costs
247         this->fuel_cost_vec[timestep] = fuel_consumed_L * this->fuel_cost_L;
248     }
249
250     return load_kW;
251 } /* commit() */

```

4.1.3.3 computeEconomics()

```

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

```

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

Parameters

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

Reimplemented from [Production](#).

```

166 {
167     // 1. account for fuel costs in net present cost
168     double t_hrs = 0;
169     double real_discount_scalar = 0;
170
171     for (int i = 0; i < this->n_points; i++) {
172         t_hrs = time_vec_hrs_ptr->at(i);
173
174         real_discount_scalar = 1.0 / pow(
175             1 + this->real_discount_annual,
176             t_hrs / 8760
177         );
178
179         this->net_present_cost += real_discount_scalar * this->fuel_cost_vec[i];
180     }
181
182     // 2. invoke base class method
183     Production::computeEconomics(time_vec_hrs_ptr);
184 }

```

```

185     return;
186 } /* computeEconomics() */

```

4.1.3.4 computeFuelAndEmissions()

```

void Combustion::computeFuelAndEmissions (
    void )

```

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

```

136 {
137     for (int i = 0; i < n_points; i++) {
138         this->total_fuel_consumed_L += this->fuel_consumption_vec_L[i];
139
140         this->total_emissions.CO2_kg += this->CO2_emissions_vec_kg[i];
141         this->total_emissions.CO_kg += this->CO_emissions_vec_kg[i];
142         this->total_emissions.NOx_kg += this->NOx_emissions_vec_kg[i];
143         this->total_emissions.SOx_kg += this->SOx_emissions_vec_kg[i];
144         this->total_emissions.CH4_kg += this->CH4_emissions_vec_kg[i];
145         this->total_emissions.PM_kg += this->PM_emissions_vec_kg[i];
146     }
147
148     return;
149 } /* computeFuelAndEmissions() */

```

4.1.3.5 getEmissionskg()

```

Emissions Combustion::getEmissionskg (
    double fuel_consumed_L )

```

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

Parameters

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

Returns

A structure containing the mass spectrum of resulting emissions.

```

299     {
300     Emissions emissions;
301
302     emissions.CO2_kg = this->CO2_emissions_intensity_kgL * fuel_consumed_L;
303     emissions.CO_kg = this->CO_emissions_intensity_kgL * fuel_consumed_L;
304     emissions.NOx_kg = this->NOx_emissions_intensity_kgL * fuel_consumed_L;
305     emissions.SOx_kg = this->SOx_emissions_intensity_kgL * fuel_consumed_L;
306     emissions.CH4_kg = this->CH4_emissions_intensity_kgL * fuel_consumed_L;
307     emissions.PM_kg = this->PM_emissions_intensity_kgL * fuel_consumed_L;
308
309     return emissions;
310 } /* getEmissionskg() */

```

4.1.3.6 getFuelConsumptionL()

```

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

```

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

Parameters

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

Returns

The volume of fuel consumed [L].

```

273 {
274     double fuel_consumed_L = (
275         this->linear_fuel_slope_LkWh * production_kW +
276         this->linear_fuel_intercept_LkWh * this->capacity_kW
277     ) * dt_hrs;
278
279     return fuel_consumed_L;
280 } /* getFuelConsumptionL() */

```

4.1.3.7 requestProductionkW()

```

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

```

Reimplemented in [Diesel](#).

```

123 {return 0;}

```

4.1.4 Member Data Documentation

4.1.4.1 CH4_emissions_intensity_kgL

```
double Combustion::CH4_emissions_intensity_kgL
```

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

4.1.4.2 CH4_emissions_vec_kg

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

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

4.1.4.3 CO2_emissions_intensity_kgL

```
double Combustion::CO2_emissions_intensity_kgL
```

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

4.1.4.4 CO2_emissions_vec_kg

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

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

4.1.4.5 CO_emissions_intensity_kgL

```
double Combustion::CO_emissions_intensity_kgL
```

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

4.1.4.6 CO_emissions_vec_kg

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

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

4.1.4.7 fuel_consumption_vec_L

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

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

4.1.4.8 fuel_cost_L

```
double Combustion::fuel_cost_L
```

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

4.1.4.9 fuel_cost_vec

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

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

4.1.4.10 linear_fuel_intercept_LkWh

```
double Combustion::linear_fuel_intercept_LkWh
```

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

4.1.4.11 linear_fuel_slope_LkWh

```
double Combustion::linear_fuel_slope_LkWh
```

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

4.1.4.12 NOx_emissions_intensity_kgL

```
double Combustion::NOx_emissions_intensity_kgL
```

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

4.1.4.13 NOx_emissions_vec_kg

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

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

4.1.4.14 PM_emissions_intensity_kgL

```
double Combustion::PM_emissions_intensity_kgL
```

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

4.1.4.15 PM_emissions_vec_kg

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

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

4.1.4.16 SOx_emissions_intensity_kgL

```
double Combustion::SOx_emissions_intensity_kgL
```

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

4.1.4.17 SOx_emissions_vec_kg

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

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

4.1.4.18 total_emissions

```
Emissions Combustion::total_emissions
```

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

4.1.4.19 total_fuel_consumed_L

```
double Combustion::total_fuel_consumed_L
```

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

4.1.4.20 type

```
CombustionType Combustion::type
```

The type (CombustionType) of the asset.

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

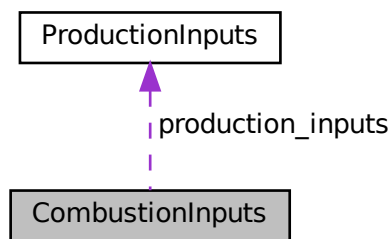
- header/Production/Combustion/[Combustion.h](#)
- source/Production/Combustion/[Combustion.cpp](#)

4.2 CombustionInputs Struct Reference

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

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

Collaboration diagram for CombustionInputs:



Public Attributes

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

4.2.1 Detailed Description

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

4.2.2 Member Data Documentation

4.2.2.1 `production_inputs`

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

An encapsulated [ProductionInputs](#) instance.

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

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

4.3 Controller Class Reference

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

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

Public Member Functions

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

Public Attributes

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

Private Member Functions

- void [__computeNetLoad](#) ([ElectricalLoad](#) *, std::vector< [Renewable](#) * > *, [Resources](#) *)
Helper method to compute and populate the net load vector.
- void [__constructCombustionMap](#) (std::vector< [Combustion](#) * > *)
Helper method to construct a [Combustion](#) map, for use in determining.
- void [__applyLoadFollowingControl_CHARGING](#) (int, [ElectricalLoad](#) *, std::vector< [Combustion](#) * > *, std::vector< [Renewable](#) * > *, std::vector< [Storage](#) * > *)
Helper method to apply load following control action for given timestep of the [Model](#) run when net load <= 0.
- void [__applyLoadFollowingControl_DISCHARGING](#) (int, [ElectricalLoad](#) *, std::vector< [Combustion](#) * > *, std::vector< [Renewable](#) * > *, std::vector< [Storage](#) * > *)
Helper method to apply load following control action for given timestep of the [Model](#) run when net load > 0.
- void [__applyCycleChargingControl_CHARGING](#) (int, [ElectricalLoad](#) *, std::vector< [Combustion](#) * > *, std::vector< [Renewable](#) * > *, std::vector< [Storage](#) * > *)
Helper method to apply cycle charging control action for given timestep of the [Model](#) run when net load <= 0. Simply defaults to load following control.

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

4.3.1 Detailed Description

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

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Controller()

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

Constructor for the [Controller](#) class.

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

4.3.2.2 ~Controller()

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

Destructor for the [Controller](#) class.

```
993 {
994     this->clear();
995
996     return;
997 } /* ~Controller() */
```

4.3.3 Member Function Documentation

4.3.3.1 __applyCycleChargingControl_CHARGING()

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

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

Parameters

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

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

4.3.3.2 __applyCycleChargingControl_DISCHARGING()

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

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

Parameters

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

curtailment

```

434 {
435     // 1. get dt_hrs, net load
436     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
437     double net_load_kW = this->net_load_vec_kW[timestep];
438
439     // 2. partition Storage assets into depleted and non-depleted
440     std::list<Storage*> depleted_storage_ptr_list;
441     std::list<Storage*> nondepleted_storage_ptr_list;
442
443     Storage* storage_ptr;
444     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
445         storage_ptr = storage_ptr_vec_ptr->at(i);
446
447         //...
448     }
449
450     // 3. discharge non-depleted storage assets
451     net_load_kW = this->__handleStorageDischarging(
452         timestep,
453         dt_hrs,
454         net_load_kW,
455         nondepleted_storage_ptr_list
456     );
457
458     // 4. request optimal production from all Combustion assets
459     // default to load following if no depleted storage
460     if (depleted_storage_ptr_list.empty()) {
461         net_load_kW = this->__handleCombustionDispatch(
462             timestep,
463             dt_hrs,
464             net_load_kW,
465             combustion_ptr_vec_ptr,
466             false // is_cycle_charging
467         );
468     }
469
470     else {
471         net_load_kW = this->__handleCombustionDispatch(
472             timestep,
473             dt_hrs,
474             net_load_kW,
475             combustion_ptr_vec_ptr,
476             true // is_cycle_charging
477         );
478     }
479
480     // 5. attempt to charge depleted Storage assets using any and all available
481     // charge priority is Combustion, then Renewable
482     this->__handleStorageCharging(timestep, dt_hrs, depleted_storage_ptr_list);
483
484     // 6. record any missed load
485     if (net_load_kW > 0) {
486         this->missed_load_vec_kW[timestep] = net_load_kW;
487     }
488
489     return;
490 }
491 } /* __applyCycleChargingControl_DISCHARGING() */

```

4.3.3.3 __applyLoadFollowingControl_CHARGING()

```

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

```

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

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>electrical_load_ptr</i>	A pointer to the ElectricalLoad component of the Model .

Parameters

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

```

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

```

4.3.3.4 __applyLoadFollowingControl_DISCHARGING()

```

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

```

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

Parameters

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

curtailment

```

302 {
303     // 1. get dt_hrs, net load
304     double dt_hrs = electrical_load_ptr->dt_vec_hrs[timestep];
305     double net_load_kW = this->net_load_vec_kW[timestep];
306
307     // 2. partition Storage assets into depleted and non-depleted
308     std::list<Storage*> depleted_storage_ptr_list;
309     std::list<Storage*> nondepleted_storage_ptr_list;
310
311     Storage* storage_ptr;
312     for (size_t i = 0; i < storage_ptr_vec_ptr->size(); i++) {
313         storage_ptr = storage_ptr_vec_ptr->at(i);
314
315         //...
316     }
317
318     // 3. discharge non-depleted storage assets

```

```

319     net_load_kW = this->__handleStorageDischarging(
320         timestep,
321         dt_hrs,
322         net_load_kW,
323         nondepleted_storage_ptr_list
324     );
325
326     // 4. request optimal production from all Combustion assets
327     net_load_kW = this->__handleCombustionDispatch(
328         timestep,
329         dt_hrs,
330         net_load_kW,
331         combustion_ptr_vec_ptr,
332         false // is_cycle_charging
333     );
334
335     // 5. attempt to charge depleted Storage assets using any and all available
336     // charge priority is Combustion, then Renewable
337     this->__handleStorageCharging(timestep, dt_hrs, depleted_storage_ptr_list);
338
339     // 6. record any missed load
340     if (net_load_kW > 0) {
341         this->missed_load_vec_kW[timestep] = net_load_kW;
342     }
343
344     return;
345 } /* __applyLoadFollowingControl_DISCHARGING() */

```

4.3.3.5 __computeNetLoad()

```

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

```

Helper method to compute and populate the net load vector.

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

Parameters

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

```

57 {
58     // 1. init
59     this->net_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
60     this->missed_load_vec_kW.resize(electrical_load_ptr->n_points, 0);
61
62     // 2. populate net load vector
63     double dt_hrs = 0;
64     double load_kW = 0;
65     double net_load_kW = 0;
66     double production_kW = 0;
67
68     Renewable* renewable_ptr;
69
70     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
71         dt_hrs = electrical_load_ptr->dt_vec_hrs[i];
72         load_kW = electrical_load_ptr->load_vec_kW[i];
73         net_load_kW = load_kW;
74
75         for (size_t j = 0; j < renewable_ptr_vec_ptr->size(); j++) {
76             renewable_ptr = renewable_ptr_vec_ptr->at(j);
77
78             production_kW = this->__getRenewableProduction(
79                 i,

```



```

80         dt_hrs,
81         renewable_ptr,
82         resources_ptr
83     );
84
85     load_kW = renewable_ptr->commit(
86         i,
87         dt_hrs,
88         production_kW,
89         load_kW
90     );
91
92     net_load_kW -= production_kW;
93 }
94
95     this->net_load_vec_kW[i] = net_load_kW;
96 }
97
98     return;
99 } /* __computeNetLoad() */

```

4.3.3.6 __constructCombustionMap()

```

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

```

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

Parameters

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

```

121 {
122     // 1. get state table dimensions
123     int n_cols = combustion_ptr_vec_ptr->size();
124     int n_rows = pow(2, n_cols);
125
126     // 2. init state table (all possible on/off combinations)
127     std::vector<std::vector<bool>> state_table;
128     state_table.resize(n_rows, {});
129
130     int x = 0;
131     for (int i = 0; i < n_rows; i++) {
132         state_table[i].resize(n_cols, false);
133
134         x = i;
135         for (int j = 0; j < n_cols; j++) {
136             if (x % 2 == 0) {
137                 state_table[i][j] = true;
138             }
139             x /= 2;
140         }
141     }
142
143     // 3. construct combustion map (handle duplicates by keeping rows with minimum
144     //     trues)
145     double total_capacity_kW = 0;
146     int truth_count = 0;
147     int current_truth_count = 0;
148
149     for (int i = 0; i < n_rows; i++) {
150         total_capacity_kW = 0;
151         truth_count = 0;
152         current_truth_count = 0;
153
154         for (int j = 0; j < n_cols; j++) {
155             if (state_table[i][j]) {
156                 total_capacity_kW += combustion_ptr_vec_ptr->at(j)->capacity_kW;
157                 truth_count++;
158             }
159         }
160
161         if (this->combustion_map.count(total_capacity_kW) > 0) {
162             for (int j = 0; j < n_cols; j++) {
163                 if (this->combustion_map[total_capacity_kW][j]) {

```

```

164             current_truth_count++;
165         }
166     }
167
168     if (truth_count < current_truth_count) {
169         this->combustion_map.erase(total_capacity_kW);
170     }
171 }
172
173 this->combustion_map.insert(
174     std::pair<double, std::vector<bool>> (
175         total_capacity_kW,
176         state_table[i]
177     )
178 );
179 }
180
181 // 4. test print
182 /*
183 std::cout << std::endl;
184
185 std::cout << "\t\t";
186 for (size_t i = 0; i < combustion_ptr_vec_ptr->size(); i++) {
187     std::cout << combustion_ptr_vec_ptr->at(i)->capacity_kW << "\t";
188 }
189 std::cout << std::endl;
190
191 std::map<double, std::vector<bool>>::iterator iter;
192 for (
193     iter = this->combustion_map.begin();
194     iter != this->combustion_map.end();
195     iter++
196 ) {
197     std::cout << iter->first << ":\t\t";
198
199     for (size_t i = 0; i < iter->second.size(); i++) {
200         std::cout << iter->second[i] << "\t";
201     }
202     std::cout << "]" << std::endl;
203 }
204 */
205
206 return;
207 } /* __constructCombustionTable() */

```

4.3.3.7 __getRenewableProduction()

```

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

```

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

Parameters

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

Returns

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

```

595 {
596     double production_kW = 0;

```

```

597
598     switch (renewable_ptr->type) {
599         case (RenewableType :: SOLAR): {
600             production_kW = renewable_ptr->computeProductionkW(
601                 timestep,
602                 dt_hrs,
603                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
604             );
605             break;
606         }
607
608         case (RenewableType :: TIDAL): {
609             production_kW = renewable_ptr->computeProductionkW(
610                 timestep,
611                 dt_hrs,
612                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
613             );
614             break;
615         }
616
617         case (RenewableType :: WAVE): {
618             production_kW = renewable_ptr->computeProductionkW(
619                 timestep,
620                 dt_hrs,
621                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][0],
622                 resources_ptr->resource_map_2D[renewable_ptr->resource_key][timestep][1]
623             );
624             break;
625         }
626
627         case (RenewableType :: WIND): {
628             production_kW = renewable_ptr->computeProductionkW(
629                 timestep,
630                 dt_hrs,
631                 resources_ptr->resource_map_1D[renewable_ptr->resource_key][timestep]
632             );
633             break;
634         }
635
636         default: {
637             std::string error_str = "ERROR: Controller::__getRenewableProduction(): ";
638             error_str += "renewable type ";
639             error_str += std::to_string(renewable_ptr->type);
640             error_str += " not recognized";
641
642             #ifdef _WIN32
643                 std::cout << error_str << std::endl;
644             #endif
645
646             throw std::runtime_error(error_str);
647
648             break;
649         }
650     }
651
652     return production_kW;
653 } /* __getRenewableProduction() */

```

4.3.3.8 __handleCombustionDispatch()

```

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

```

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

bool is_cycle_charging)

Parameters

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

Returns

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

```

699 {
700     // 1. get minimal Combustion dispatch
701     double target_production_kW = 1.2 * net_load_kW;
702     double total_capacity_kW = 0;
703
704     std::map<double, std::vector<bool>::iterator> iter = this->combustion_map.begin();
705     while (iter != std::prev(this->combustion_map.end(), 1)) {
706         if (target_production_kW <= total_capacity_kW) {
707             break;
708         }
709         iter++;
710         total_capacity_kW = iter->first;
711     }
712
713     // 2. share load proportionally (by rated capacity) over active diesels
714     Combustion* combustion_ptr;
715     double production_kW = 0;
716     double request_kW = 0;
717     double _net_load_kW = net_load_kW;
718
719     for (size_t i = 0; i < this->combustion_map[total_capacity_kW].size(); i++) {
720         combustion_ptr = combustion_ptr_vec_ptr->at(i);
721
722         if (total_capacity_kW > 0) {
723             request_kW =
724                 int(this->combustion_map[total_capacity_kW][i]) *
725                 net_load_kW *
726                 (combustion_ptr->capacity_kW / total_capacity_kW);
727         }
728         else {
729             request_kW = 0;
730         }
731
732         if (is_cycle_charging and request_kW > 0) {
733             if (request_kW < 0.85 * combustion_ptr->capacity_kW) {
734                 request_kW = 0.85 * combustion_ptr->capacity_kW;
735             }
736         }
737
738         production_kW = combustion_ptr->requestProductionkW(
739             timestep,
740             dt_hrs,
741             request_kW
742         );
743
744         _net_load_kW = combustion_ptr->commit(
745             timestep,
746             dt_hrs,
747             production_kW,
748             _net_load_kW
749         );
750     }
751     return _net_load_kW;
752 }
753
754 /* __handleCombustionDispatch() */
755 }

```

4.3.3.9 __handleStorageCharging() [1/2]

```

void Controller::__handleStorageCharging (
    int timestep,

```

```
double dt_hrs,
std::list< Storage * > storage_ptr_list ) [private]
```

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

Parameters

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

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

4.3.3.10 __handleStorageCharging() [2/2]

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

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

Parameters

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

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

4.3.3.11 __handleStorageDischarging()

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

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

Parameters

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

Returns

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

```

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

```

4.3.3.12 applyDispatchControl()

```

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

```

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

Parameters

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

```

888 {
889     for (int i = 0; i < electrical_load_ptr->n_points; i++) {
890         switch (this->control_mode) {
891             case (ControlMode :: LOAD_FOLLOWING): {
892                 if (this->net_load_vec_kW[i] <= 0) {
893                     this->__applyLoadFollowingControl_CHARGING(
894                         i,
895                         electrical_load_ptr,
896                         combustion_ptr_vec_ptr,
897                         renewable_ptr_vec_ptr,
898                         storage_ptr_vec_ptr
899                     );
900                 }
901             }
902             else {
903                 this->__applyLoadFollowingControl_DISCHARGING(
904                     i,
905                     electrical_load_ptr,
906                     combustion_ptr_vec_ptr,
907                     renewable_ptr_vec_ptr,
908                     storage_ptr_vec_ptr
909                 );
910             }
911             break;
912         }
913     }
914
915     case (ControlMode :: CYCLE_CHARGING): {
916         if (this->net_load_vec_kW[i] <= 0) {
917             this->__applyCycleChargingControl_CHARGING(
918                 i,
919                 electrical_load_ptr,
920                 combustion_ptr_vec_ptr,
921                 renewable_ptr_vec_ptr,
922                 storage_ptr_vec_ptr
923             );
924         }
925     }
926     else {
927         this->__applyCycleChargingControl_DISCHARGING(
928             i,
929             electrical_load_ptr,
930             combustion_ptr_vec_ptr,
931             renewable_ptr_vec_ptr,

```

```

932         storage_ptr_vec_ptr
933     );
934 }
935
936     break;
937 }
938
939     default: {
940         std::string error_str = "ERROR: Controller :: applyDispatchControl(): ";
941         error_str += "control mode ";
942         error_str += std::to_string(this->control_mode);
943         error_str += " not recognized";
944
945         #ifdef _WIN32
946             std::cout << error_str << std::endl;
947         #endif
948
949         throw std::runtime_error(error_str);
950
951         break;
952     }
953 }
954 }
955
956 return;
957 } /* applyDispatchControl() */

```

4.3.3.13 clear()

```

void Controller::clear (
    void )

```

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

```

972 {
973     this->net_load_vec_kW.clear();
974     this->missed_load_vec_kW.clear();
975     this->combustion_map.clear();
976
977     return;
978 } /* clear() */

```

4.3.3.14 init()

```

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

```

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

Parameters

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

```

847 {
848     // 1. compute net load
849     this->__computeNetLoad(electrical_load_ptr, renewable_ptr_vec_ptr, resources_ptr);

```

```
850
851 // 2. construct Combustion table
852 this->__constructCombustionMap(combustion_ptr_vec_ptr);
853
854 return;
855 } /* init() */
```

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 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.

4.3.4.4 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:

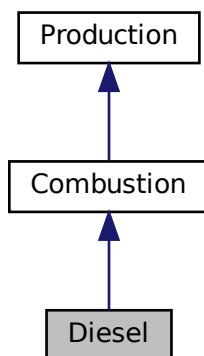
- header/[Controller.h](#)
- source/[Controller.cpp](#)

4.4 Diesel Class Reference

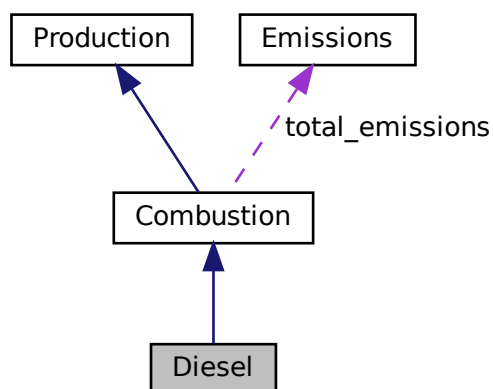
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

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

Inheritance diagram for Diesel:



Collaboration diagram for Diesel:



Public Member Functions

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

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

Public Attributes

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

Private Member Functions

- void [__checkInputs](#) ([DieselInputs](#))
Helper method to check inputs to the [Diesel](#) constructor.
- void [__handleStartStop](#) (int, double, double)
Helper method (private) to handle the starting/stopping of the diesel generator. The minimum runtime constraint is enforced in this method.
- double [__getGenericFuelSlope](#) (void)
Helper method to generate a generic, linearized fuel consumption slope for a diesel generator.
- double [__getGenericFuelIntercept](#) (void)
Helper method to generate a generic, linearized fuel consumption intercept for a diesel generator.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic diesel generator capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method (private) to generate a generic diesel generator operation and maintenance cost. This is a cost incurred per unit energy produced.

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.

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

Parameters

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

```

337 {
338     return;
339 } /* Diesel() */

```

4.4.2.2 Diesel() [2/2]

```

Diesel::Diesel (
    int n_points,
    DieselInputs diesel_inputs )
    :
357 Combustion(n_points, diesel_inputs.combustion_inputs)
358 {
359     // 1. check inputs
360     this->__checkInputs(diesel_inputs);
361
362     // 2. set attributes
363     this->type = CombustionType :: DIESEL;
364     this->type_str = "DIESEL";
365
366     this->replace_running_hrs = diesel_inputs.replace_running_hrs;
367
368     this->fuel_cost_L = diesel_inputs.fuel_cost_L;
369
370     this->minimum_load_ratio = diesel_inputs.minimum_load_ratio;
371     this->minimum_runtime_hrs = diesel_inputs.minimum_runtime_hrs;
372     this->time_since_last_start_hrs = 0;
373
374     this->CO2_emissions_intensity_kgL = diesel_inputs.CO2_emissions_intensity_kgL;
375     this->CO_emissions_intensity_kgL = diesel_inputs.CO_emissions_intensity_kgL;
376     this->NOx_emissions_intensity_kgL = diesel_inputs.NOx_emissions_intensity_kgL;
377     this->SOx_emissions_intensity_kgL = diesel_inputs.SOx_emissions_intensity_kgL;
378     this->CH4_emissions_intensity_kgL = diesel_inputs.CH4_emissions_intensity_kgL;
379     this->PM_emissions_intensity_kgL = diesel_inputs.PM_emissions_intensity_kgL;
380
381     if (diesel_inputs.linear_fuel_slope_LkWh < 0) {
382         this->linear_fuel_slope_LkWh = this->__getGenericFuelSlope();
383     }
384
385     if (diesel_inputs.linear_fuel_intercept_LkWh < 0) {
386         this->linear_fuel_intercept_LkWh = this->__getGenericFuelIntercept();
387     }
388
389     if (diesel_inputs.capital_cost < 0) {
390         this->capital_cost = this->__getGenericCapitalCost();
391     }
392
393     if (diesel_inputs.operation_maintenance_cost_kWh < 0) {
394         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
395     }
396
397     if (this->is_sunk) {
398         this->capital_cost_vec[0] = this->capital_cost;
399     }
400
401     // 3. construction print
402     if (this->print_flag) {
403         std::cout << "Diesel object constructed at " << this << std::endl;
404     }
405
406     return;
407 } /* Diesel() */

```

4.4.2.3 ~Diesel()

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

Destructor for the [Diesel](#) class.

```
537 {
538     // 1. destruction print
539     if (this->print_flag) {
540         std::cout << "Diesel object at " << this << " destroyed" << std::endl;
541     }
542
543     return;
544 } /* ~Diesel() */
```

4.4.3 Member Function Documentation

4.4.3.1 __checkInputs()

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

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

Parameters

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

```
39 {
40     // 1. check fuel_cost_L
41     if (diesel_inputs.fuel_cost_L < 0) {
42         std::string error_str = "ERROR: Diesel(): ";
43         error_str += "DieselInputs::fuel_cost_L must be >= 0";
44
45         #ifdef _WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check CO2_emissions_intensity_kgL
53     if (diesel_inputs.CO2_emissions_intensity_kgL < 0) {
54         std::string error_str = "ERROR: Diesel(): ";
55         error_str += "DieselInputs::CO2_emissions_intensity_kgL must be >= 0";
56
57         #ifdef _WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     // 3. check CO_emissions_intensity_kgL
65     if (diesel_inputs.CO_emissions_intensity_kgL < 0) {
66         std::string error_str = "ERROR: Diesel(): ";
67         error_str += "DieselInputs::CO_emissions_intensity_kgL must be >= 0";
68
69         #ifdef _WIN32
70             std::cout << error_str << std::endl;
71         #endif
72
73         throw std::invalid_argument(error_str);
74     }
75
76     // 4. check NOx_emissions_intensity_kgL
77     if (diesel_inputs.NOx_emissions_intensity_kgL < 0) {
```

```

78     std::string error_str = "ERROR: Diesel(): ";
79     error_str += "DieselInputs::NOx_emissions_intensity_kgL must be >= 0";
80
81     #ifdef _WIN32
82         std::cout << error_str << std::endl;
83     #endif
84
85     throw std::invalid_argument(error_str);
86 }
87
88 // 5. check SOx_emissions_intensity_kgL
89 if (diesel_inputs.SOx_emissions_intensity_kgL < 0) {
90     std::string error_str = "ERROR: Diesel(): ";
91     error_str += "DieselInputs::SOx_emissions_intensity_kgL must be >= 0";
92
93     #ifdef _WIN32
94         std::cout << error_str << std::endl;
95     #endif
96
97     throw std::invalid_argument(error_str);
98 }
99
100 // 6. check CH4_emissions_intensity_kgL
101 if (diesel_inputs.CH4_emissions_intensity_kgL < 0) {
102     std::string error_str = "ERROR: Diesel(): ";
103     error_str += "DieselInputs::CH4_emissions_intensity_kgL must be >= 0";
104
105     #ifdef _WIN32
106         std::cout << error_str << std::endl;
107     #endif
108
109     throw std::invalid_argument(error_str);
110 }
111
112 // 7. check PM_emissions_intensity_kgL
113 if (diesel_inputs.PM_emissions_intensity_kgL < 0) {
114     std::string error_str = "ERROR: Diesel(): ";
115     error_str += "DieselInputs::PM_emissions_intensity_kgL must be >= 0";
116
117     #ifdef _WIN32
118         std::cout << error_str << std::endl;
119     #endif
120
121     throw std::invalid_argument(error_str);
122 }
123
124 // 8. check minimum_load_ratio
125 if (diesel_inputs.minimum_load_ratio < 0) {
126     std::string error_str = "ERROR: Diesel(): ";
127     error_str += "DieselInputs::minimum_load_ratio must be >= 0";
128
129     #ifdef _WIN32
130         std::cout << error_str << std::endl;
131     #endif
132
133     throw std::invalid_argument(error_str);
134 }
135
136 // 9. check minimum_runtime_hrs
137 if (diesel_inputs.minimum_runtime_hrs < 0) {
138     std::string error_str = "ERROR: Diesel(): ";
139     error_str += "DieselInputs::minimum_runtime_hrs must be >= 0";
140
141     #ifdef _WIN32
142         std::cout << error_str << std::endl;
143     #endif
144
145     throw std::invalid_argument(error_str);
146 }
147
148 // 10. check replace_running_hrs
149 if (diesel_inputs.replace_running_hrs <= 0) {
150     std::string error_str = "ERROR: Diesel(): ";
151     error_str += "DieselInputs::replace_running_hrs must be > 0";
152
153     #ifdef _WIN32
154         std::cout << error_str << std::endl;
155     #endif
156
157     throw std::invalid_argument(error_str);
158 }
159
160 return;
161 } /* __checkInputs() */

```

4.4.3.2 `__getGenericCapitalCost()`

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

Helper method to generate a generic diesel generator capital cost.

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

Returns

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

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

4.4.3.3 `__getGenericFuelIntercept()`

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

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

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

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

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

Returns

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

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

4.4.3.4 `__getGenericFuelSlope()`

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

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

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

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

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

Returns

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

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

4.4.3.5 `__getGenericOpMaintCost()`

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

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

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

Returns

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

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

4.4.3.6 `__handleStartStop()`

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

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

Parameters

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

```
292 {
293     /*
294     * Helper method (private) to handle the starting/stopping of the diesel
295     * generator. The minimum runtime constraint is enforced in this method.
296     */
297
298     if (this->is_running) {
299         // handle stopping
300         if (
301             production_kW <= 0 and
302             this->time_since_last_start_hrs >= this->minimum_runtime_hrs
303         ) {
304             this->is_running = false;
305         }
306     }
307
308     else {
309         // handle starting
310         if (production_kW > 0) {
311             this->is_running = true;
312             this->n_starts++;
313             this->time_since_last_start_hrs = 0;
314         }
315     }
316 }
```

```

317     return;
318 } /* __handleStartStop() */

```

4.4.3.7 commit()

```

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

```

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

Parameters

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

Returns

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

Reimplemented from [Combustion](#).

```

495 {
496     // 1. handle start/stop, enforce minimum runtime constraint
497     this->__handleStartStop(timestep, dt_hrs, production_kW);
498
499     // 2. invoke base class method
500     load_kW = Combustion::commit(
501         timestep,
502         dt_hrs,
503         production_kW,
504         load_kW
505     );
506
507     if (this->is_running) {
508         // 3. log time since last start
509         this->time_since_last_start_hrs += dt_hrs;
510
511         // 4. correct operation and maintenance costs (should be non-zero if idling)
512         if (production_kW <= 0) {
513             double produced_kWh = 0.01 * this->capacity_kW * dt_hrs;
514
515             double operation_maintenance_cost =
516                 this->operation_maintenance_cost_kWh * produced_kWh;
517             this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
518         }
519     }
520
521     return load_kW;
522 } /* commit() */

```

4.4.3.8 requestProductionkW()

```

double Diesel::requestProductionkW (
    int timestep,

```



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

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

Parameters

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

Returns

The production [kW] delivered by the diesel generator.

Reimplemented from [Combustion](#).

```
440 {
441     // 1. return on request of zero
442     if (request_kW <= 0) {
443         return 0;
444     }
445
446     double deliver_kW = request_kW;
447
448     // 2. enforce capacity constraint
449     if (deliver_kW > this->capacity_kW) {
450         deliver_kW = this->capacity_kW;
451     }
452
453     // 3. enforce minimum load ratio
454     if (deliver_kW < this->minimum_load_ratio * this->capacity_kW) {
455         deliver_kW = this->minimum_load_ratio * this->capacity_kW;
456     }
457
458     return deliver_kW;
459 } /* requestProductionkW() */
```

4.4.4 Member Data Documentation

4.4.4.1 minimum_load_ratio

```
double Diesel::minimum_load_ratio
```

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

4.4.4.2 minimum_runtime_hrs

```
double Diesel::minimum_runtime_hrs
```

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

4.4.4.3 time_since_last_start_hrs

```
double Diesel::time_since_last_start_hrs
```

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

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

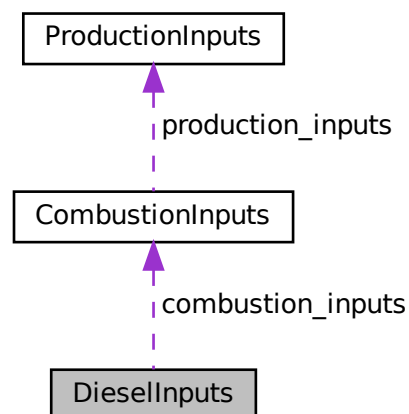
- [header/Production/Combustion/Diesel.h](#)
- [source/Production/Combustion/Diesel.cpp](#)

4.5 DieselInputs Struct Reference

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

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

Collaboration diagram for DieselInputs:



Public Attributes

- [CombustionInputs combustion_inputs](#)
An encapsulated [CombustionInputs](#) instance.
- double [replace_running_hrs](#) = 30000
The number of running hours after which the asset must be replaced. Overwrites the [ProductionInputs](#) attribute.
- double [capital_cost](#) = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double [operation_maintenance_cost_kWh](#) = -1

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

- double `fuel_cost_L` = 1.70

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

- double `minimum_load_ratio` = 0.2

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

- double `minimum_runtime_hrs` = 4

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

- double `linear_fuel_slope_LkWh` = -1

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

- double `linear_fuel_intercept_LkWh` = -1

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

- double `CO2_emissions_intensity_kgL` = 2.7

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

- double `CO_emissions_intensity_kgL` = 0.0178

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

- double `NOx_emissions_intensity_kgL` = 0.0014

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

- double `SOx_emissions_intensity_kgL` = 0.0042

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

- double `CH4_emissions_intensity_kgL` = 0.0007

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

- double `PM_emissions_intensity_kgL` = 0.0001

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

4.5.1 Detailed Description

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

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

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

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

Ref: docs/refs/diesel_emissions_ref_1.pdf

Ref: docs/refs/diesel_emissions_ref_2.pdf

4.5.2 Member Data Documentation

4.5.2.1 capital_cost

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

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

4.5.2.2 CH4_emissions_intensity_kgL

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

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

4.5.2.3 CO2_emissions_intensity_kgL

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

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

4.5.2.4 CO_emissions_intensity_kgL

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

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

4.5.2.5 combustion_inputs

```
CombustionInputs DieselInputs::combustion_inputs
```

An encapsulated [CombustionInputs](#) instance.

4.5.2.6 fuel_cost_L

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

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

4.5.2.7 linear_fuel_intercept_LkWh

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

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

4.5.2.8 linear_fuel_slope_LkWh

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

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

4.5.2.9 minimum_load_ratio

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

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

4.5.2.10 minimum_runtime_hrs

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

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

4.5.2.11 NOx_emissions_intensity_kgL

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

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

4.5.2.12 operation_maintenance_cost_kWh

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

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

4.5.2.13 PM_emissions_intensity_kgL

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

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

4.5.2.14 replace_running_hrs

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

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

4.5.2.15 SOx_emissions_intensity_kgL

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

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

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

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

4.6 ElectricalLoad Class Reference

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

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

Public Member Functions

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

Public Attributes

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

4.6.1 Detailed Description

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

4.6.2 Constructor & Destructor Documentation

4.6.2.1 ElectricalLoad() [1/2]

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

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

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

4.6.2.2 ElectricalLoad() [2/2]

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

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

Parameters

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

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

4.6.2.3 ~ElectricalLoad()

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

Destructor for the [ElectricalLoad](#) class.

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

4.6.3 Member Function Documentation

4.6.3.1 clear()

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

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

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



```

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

```

4.6.3.2 readLoadData()

```

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

```

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

Parameters

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

```

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

```

```

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

```

4.6.4 Member Data Documentation

4.6.4.1 dt_vec_hrs

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

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

4.6.4.2 load_vec_kW

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

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

4.6.4.3 max_load_kW

`double ElectricalLoad::max_load_kW`

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

4.6.4.4 mean_load_kW

`double ElectricalLoad::mean_load_kW`

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

4.6.4.5 min_load_kW

```
double ElectricalLoad::min_load_kW
```

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

4.6.4.6 n_points

```
int ElectricalLoad::n_points
```

The number of points in the modelling time series.

4.6.4.7 n_years

```
double ElectricalLoad::n_years
```

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

4.6.4.8 path_2_electrical_load_time_series

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

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

4.6.4.9 time_vec_hrs

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

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

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

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

4.7 Emissions Struct Reference

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

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

Public Attributes

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

4.7.1 Detailed Description

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

4.7.2 Member Data Documentation

4.7.2.1 CH4_kg

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

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

4.7.2.2 CO2_kg

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

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

4.7.2.3 CO_kg

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

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

4.7.2.4 NOx_kg

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

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

4.7.2.5 PM_kg

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

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

4.7.2.6 SOx_kg

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

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

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

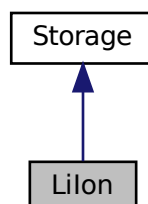
- [header/Production/Combustion/Combustion.h](#)

4.8 Lilon Class Reference

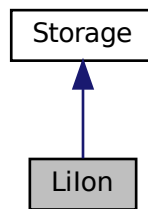
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

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

Inheritance diagram for Lilon:



Collaboration diagram for Lilon:



Public Member Functions

- `Lilon` (void)
Constructor for the `Lilon` class.
- `~Lilon` (void)
Destructor for the `Lilon` class.

4.8.1 Detailed Description

A derived class of `Storage` which models energy storage by way of lithium-ion batteries.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Lilon()

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

Constructor for the `Lilon` class.

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

4.8.2.2 ~Lilon()

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

Destructor for the [LiIon](#) class.

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

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

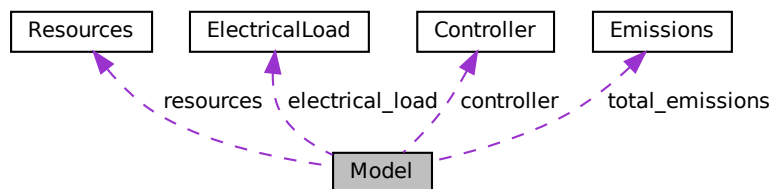
- header/Storage/[Lilon.h](#)
- source/Storage/[Lilon.cpp](#)

4.9 Model Class Reference

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

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

Collaboration diagram for Model:



Public Member Functions

- [Model](#) (void)
Constructor (dummy) for the [Model](#) class.
- [Model](#) (ModelInputs)
Constructor (intended) for the [Model](#) class.
- void [addDiesel](#) (DieselInputs)
Method to add a [Diesel](#) asset to the [Model](#).
- void [addResource](#) (RenewableType, std::string, int)
A method to add a renewable resource time series to the [Model](#).
- void [addSolar](#) (SolarInputs)
Method to add a [Solar](#) asset to the [Model](#).
- void [addTidal](#) (TidalInputs)
Method to add a [Tidal](#) asset to the [Model](#).

- void `addWave` (`WaveInputs`)
Method to add a `Wave` asset to the `Model`.
- void `addWind` (`WindInputs`)
Method to add a `Wind` asset to the `Model`.
- void `run` (void)
A method to run the `Model`.
- void `reset` (void)
Method which resets the model for use in assessing a new candidate microgrid design. This method only clears the asset pointer vectors; it leaves the `Controller`, `ElectricalLoad`, and `Resources` objects of the `Model` alone.
- void `clear` (void)
Method to clear all attributes of the `Model` object.
- `~Model` (void)
Destructor for the `Model` class.

Public Attributes

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

Private Member Functions

- void `__checkInputs` (`ModelInputs`)
Helper method (private) to check inputs to the `Model` constructor.
- void `__computeFuelAndEmissions` (void)
Helper method to compute the total fuel consumption and emissions over the `Model` run.
- void `__computeNetPresentCost` (void)
Helper method to compute the overall net present cost, for the `Model` run, from the asset-wise net present costs.
- void `__computeLevellizedCostOfEnergy` (void)
Helper method to compute the overall levellized cost of energy, for the `Model` run, from the asset-wise levellized costs of energy.
- void `__computeEconomics` (void)
Helper method to compute key economic metrics for the `Model` run.

4.9.1 Detailed Description

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

4.9.2 Constructor & Destructor Documentation

4.9.2.1 Model() [1/2]

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

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

```
232 {
233     return;
234 } /* Model() */
```

4.9.2.2 Model() [2/2]

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

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

Parameters

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

```
251 {
252     // 1. check inputs
253     this->__checkInputs(model_inputs);
254
255     // 2. read in electrical load data
256     this->electrical_load.readLoadData(model_inputs.path_2_electrical_load_time_series);
257
258     // 3. set control mode
259     this->controller.control_mode = model_inputs.control_mode;
260
261     // 4. set public attributes
262     this->total_fuel_consumed_L = 0;
263     this->net_present_cost = 0;
264     this->total_dispatch_discharge_kWh = 0;
265     this->levellized_cost_of_energy_kWh = 0;
266
267     return;
268 } /* Model() */
```

4.9.2.3 ~Model()

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

Destructor for the [Model](#) class.

```
550 {
551     this->clear();
552     return;
553 } /* ~Model() */
```

4.9.3 Member Function Documentation

4.9.3.1 __checkInputs()

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

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

Parameters

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

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

4.9.3.2 __computeEconomics()

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

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

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

4.9.3.3 __computeFuelAndEmissions()

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

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

```
60 {
61     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
62         this->combustion_ptr_vec[i]->computeFuelAndEmissions();
63
64         this->total_fuel_consumed_L +=
65             this->combustion_ptr_vec[i]->total_fuel_consumed_L;
66
67         this->total_emissions.CO2_kg +=
```

```

68         this->combustion_ptr_vec[i]->total_emissions.CO2_kg;
69
70         this->total_emissions.CO_kg +=
71             this->combustion_ptr_vec[i]->total_emissions.CO_kg;
72
73         this->total_emissions.NOx_kg +=
74             this->combustion_ptr_vec[i]->total_emissions.NOx_kg;
75
76         this->total_emissions.SOx_kg +=
77             this->combustion_ptr_vec[i]->total_emissions.SOx_kg;
78
79         this->total_emissions.CH4_kg +=
80             this->combustion_ptr_vec[i]->total_emissions.CH4_kg;
81
82         this->total_emissions.PM_kg +=
83             this->combustion_ptr_vec[i]->total_emissions.PM_kg;
84     }
85
86     return;
87 } /* __computeFuelAndEmissions() */

```

4.9.3.4 __computeLevellizedCostOfEnergy()

```

void Model::__computeLevellizedCostOfEnergy (
    void ) [private]

```

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

```

162 {
163     // 1. account for Combustion economics in levellized cost of energy
164     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
165         this->levellized_cost_of_energy_kWh +=
166             (
167                 this->combustion_ptr_vec[i]->levellized_cost_of_energy_kWh *
168                 this->combustion_ptr_vec[i]->total_dispatch_kWh
169             ) / this->total_dispatch_discharge_kWh;
170     }
171
172     // 2. account for Renewable economics in levellized cost of energy
173     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
174         this->levellized_cost_of_energy_kWh +=
175             (
176                 this->renewable_ptr_vec[i]->levellized_cost_of_energy_kWh *
177                 this->renewable_ptr_vec[i]->total_dispatch_kWh
178             ) / this->total_dispatch_discharge_kWh;
179     }
180
181     // 3. account for Storage economics in levellized cost of energy
182     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
183         /*
184         this->levellized_cost_of_energy_kWh +=
185             (
186                 this->storage_ptr_vec[i]->levellized_cost_of_energy_kWh *
187                 this->storage_ptr_vec[i]->total_dispatch_kWh
188             ) / this->total_dispatch_discharge_kWh;
189         */
190     }
191
192     return;
193 } /* __computeLevellizedCostOfEnergy() */

```

4.9.3.5 __computeNetPresentCost()

```

void Model::__computeNetPresentCost (
    void ) [private]

```

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

```

103 {
104     // 1. account for Combustion economics in net present cost

```

```

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

```

4.9.3.6 addDiesel()

```

void Model::addDiesel (
    DieselInputs diesel_inputs )

```

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

Parameters

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

```

285 {
286     Combustion* diesel_ptr = new Diesel(this->electrical_load.n_points, diesel_inputs);
287     this->combustion_ptr_vec.push_back(diesel_ptr);
288
289     return;
290 } /* addDiesel() */

```

4.9.3.7 addResource()

```

void Model::addResource (
    RenewableType renewable_type,

```

```
std::string path_2_resource_data,
int resource_key )
```

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

Parameters

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

```
320 {
321     resources.addResource(
322         renewable_type,
323         path_2_resource_data,
324         resource_key,
325         &(this->electrical_load)
326     );
327
328     return;
329 } /* addResource() */
```

4.9.3.8 addSolar()

```
void Model::addSolar (
    SolarInputs solar_inputs )
```

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

Parameters

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

```
346 {
347     Renewable* solar_ptr = new Solar(this->electrical_load.n_points, solar_inputs);
348
349     this->renewable_ptr_vec.push_back(solar_ptr);
350
351     return;
352 } /* addSolar() */
```

4.9.3.9 addTidal()

```
void Model::addTidal (
    TidalInputs tidal_inputs )
```

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

Parameters

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

```
369 {
370     Renewable* tidal_ptr = new Tidal(this->electrical_load.n_points, tidal_inputs);
```

```

371
372     this->renewable_ptr_vec.push_back(tidal_ptr);
373
374     return;
375 } /* addTidal() */

```

4.9.3.10 addWave()

```

void Model::addWave (
    WaveInputs wave_inputs )

```

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

Parameters

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

```

392 {
393     Renewable* wave_ptr = new Wave(this->electrical_load.n_points, wave_inputs);
394
395     this->renewable_ptr_vec.push_back(wave_ptr);
396
397     return;
398 } /* addWave() */

```

4.9.3.11 addWind()

```

void Model::addWind (
    WindInputs wind_inputs )

```

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

Parameters

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

```

415 {
416     Renewable* wind_ptr = new Wind(this->electrical_load.n_points, wind_inputs);
417
418     this->renewable_ptr_vec.push_back(wind_ptr);
419
420     return;
421 } /* addWind() */

```

4.9.3.12 clear()

```

void Model::clear (
    void )

```

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

```

525 {
526     // 1. reset
527     this->reset();

```

```

528
529     // 2. clear components
530     controller.clear();
531     electrical_load.clear();
532     resources.clear();
533
534     return;
535 } /* clear() */

```

4.9.3.13 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; it leaves the [Controller](#), [ElectricalLoad](#), and [Resources](#) objects of the [Model](#) alone.

```

476 {
477     // 1. clear combustion_ptr_vec
478     for (size_t i = 0; i < this->combustion_ptr_vec.size(); i++) {
479         delete this->combustion_ptr_vec[i];
480     }
481     this->combustion_ptr_vec.clear();
482
483     // 2. clear renewable_ptr_vec
484     for (size_t i = 0; i < this->renewable_ptr_vec.size(); i++) {
485         delete this->renewable_ptr_vec[i];
486     }
487     this->renewable_ptr_vec.clear();
488
489     // 3. clear storage_ptr_vec
490     for (size_t i = 0; i < this->storage_ptr_vec.size(); i++) {
491         delete this->storage_ptr_vec[i];
492     }
493     this->storage_ptr_vec.clear();
494
495     // 4. reset attributes
496     this->total_fuel_consumed_L = 0;
497
498     this->total_emissions.CO2_kg = 0;
499     this->total_emissions.CO_kg = 0;
500     this->total_emissions.NOx_kg = 0;
501     this->total_emissions.SOx_kg = 0;
502     this->total_emissions.CH4_kg = 0;
503     this->total_emissions.PM_kg = 0;
504
505     this->net_present_cost = 0;
506     this->total_dispatch_discharge_kWh = 0;
507     this->levellized_cost_of_energy_kWh = 0;
508
509     return;
510 } /* reset() */

```

4.9.3.14 run()

```

void Model::run (
    void )

```

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

```

436 {
437     // 1. init Controller
438     this->controller.init(
439         &(this->electrical_load),
440         &(this->renewable_ptr_vec),
441         &(this->resources),
442         &(this->combustion_ptr_vec)
443     );
444
445     // 2. apply dispatch control
446     this->controller.applyDispatchControl(

```

```

447         &(this->electrical_load),
448         &(this->combustion_ptr_vec),
449         &(this->renewable_ptr_vec),
450         &(this->storage_ptr_vec)
451     );
452
453     // 3. compute total fuel consumption and emissions
454     this->__computeFuelAndEmissions();
455
456     // 4. compute key economic metrics
457     this->__computeEconomics();
458
459     return;
460 } /* run() */

```

4.9.4 Member Data Documentation

4.9.4.1 combustion_ptr_vec

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

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

4.9.4.2 controller

[Controller](#) `Model::controller`

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

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

4.9.4.3 electrical_load

[ElectricalLoad](#) `Model::electrical_load`

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

4.9.4.4 levellized_cost_of_energy_kWh

`double Model::levellized_cost_of_energy_kWh`

4.9.4.5 net_present_cost

```
double Model::net_present_cost
```

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

4.9.4.6 renewable_ptr_vec

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

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

4.9.4.7 resources

```
Resources Model::resources
```

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

4.9.4.8 storage_ptr_vec

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

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

4.9.4.9 total_dispatch_discharge_kWh

```
double Model::total_dispatch_discharge_kWh
```

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

4.9.4.10 total_emissions

```
Emissions Model::total_emissions
```

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

4.9.4.11 total_fuel_consumed_L

```
double Model::total_fuel_consumed_L
```

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

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

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

4.10 ModelInputs Struct Reference

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

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

Public Attributes

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

4.10.1 Detailed Description

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

4.10.2 Member Data Documentation

4.10.2.1 control_mode

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

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

4.10.2.2 path_2_electrical_load_time_series

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

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

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

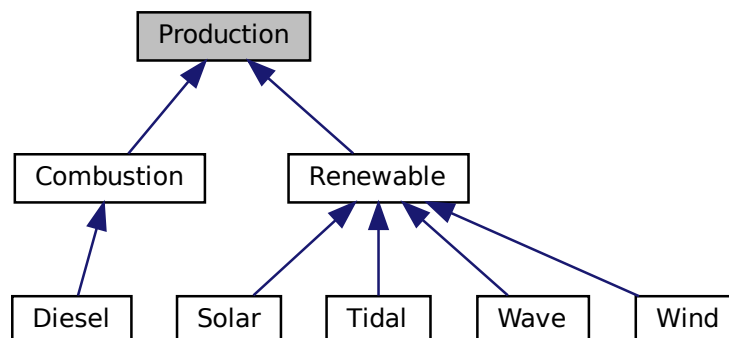
- header/[Model.h](#)

4.11 Production Class Reference

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

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

Inheritance diagram for Production:



Public Member Functions

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

Public Attributes

- bool [print_flag](#)
A flag which indicates whether or not object construct/destruction should be verbose.
- bool [is_running](#)
A boolean which indicates whether or not the asset is running.
- bool [is_sunk](#)
A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- int [n_points](#)
The number of points in the modelling time series.
- int [n_starts](#)
The number of times the asset has been started.
- int [n_replacements](#)
The number of times the asset has been replaced.
- double [running_hours](#)
The number of hours for which the asset has been operating.
- double [replace_running_hrs](#)
The number of running hours after which the asset must be replaced.
- double [capacity_kW](#)
The rated production capacity [kW] of the asset.
- double [real_discount_annual](#)
The real, annual discount rate used in computing model economics. Is computed from the given nominal inflation and discount rates.
- double [capital_cost](#)
The capital cost of the asset (undefined currency).
- double [operation_maintenance_cost_kWh](#)
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced.
- double [net_present_cost](#)
The net present cost of this asset.
- double [total_dispatch_kWh](#)
The total energy dispatched [kWh] over the [Model](#) run.
- double [levellized_cost_of_energy_kWh](#)
The levellized cost of energy [1/kWh] (undefined currency) of this asset. This metric considers only dispatched and stored energy.
- std::string [type_str](#)
A string describing the type of the asset.
- std::vector< bool > [is_running_vec](#)
A boolean vector for tracking if the asset is running at a particular point in time.
- std::vector< double > [production_vec_kW](#)
A vector of production [kW] at each point in the modelling time series.
- std::vector< double > [dispatch_vec_kW](#)
A vector of dispatch [kW] at each point in the modelling time series. Dispatch is the amount of production that is sent to the grid to satisfy load.
- std::vector< double > [storage_vec_kW](#)
A vector of storage [kW] at each point in the modelling time series. [Storage](#) is the amount of production that is sent to storage.
- std::vector< double > [curtailment_vec_kW](#)
A vector of curtailment [kW] at each point in the modelling time series. Curtailment is the amount of production that can be neither dispatched nor stored, and is hence curtailed.
- std::vector< double > [capital_cost_vec](#)

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

- `std::vector< double > operation_maintenance_cost_vec`

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

Private Member Functions

- `void __checkInputs (int, ProductionInputs)`

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

- `void __handleReplacement (int)`

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

- `double __computeRealDiscountAnnual (double, double)`

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

4.11.1 Detailed Description

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

4.11.2 Constructor & Destructor Documentation

4.11.2.1 [Production\(\)](#) [1/2]

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

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

```
164 {
165     return;
166 } /* Production() */
```

4.11.2.2 [Production\(\)](#) [2/2]

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

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

Parameters

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

```

188 {
189     // 1. check inputs
190     this->__checkInputs(n_points, production_inputs);
191
192     // 2. set attributes
193     this->print_flag = production_inputs.print_flag;
194     this->is_running = false;
195
196     this->n_points = n_points;
197     this->n_starts = 0;
198
199     this->running_hours = 0;
200     this->replace_running_hrs = production_inputs.replace_running_hrs;
201
202     this->capacity_kW = production_inputs.capacity_kW;
203
204     this->real_discount_annual = this->__computeRealDiscountAnnual(
205         production_inputs.nominal_inflation_annual,
206         production_inputs.nominal_discount_annual
207     );
208     this->capital_cost = 0;
209     this->operation_maintenance_cost_kWh = 0;
210     this->net_present_cost = 0;
211     this->total_dispatch_kWh = 0;
212     this->levellized_cost_of_energy_kWh = 0;
213
214     this->is_running_vec.resize(this->n_points, 0);
215
216     this->production_vec_kW.resize(this->n_points, 0);
217     this->dispatch_vec_kW.resize(this->n_points, 0);
218     this->storage_vec_kW.resize(this->n_points, 0);
219     this->curtailment_vec_kW.resize(this->n_points, 0);
220
221     this->capital_cost_vec.resize(this->n_points, 0);
222     this->operation_maintenance_cost_vec.resize(this->n_points, 0);
223
224     // 3. construction print
225     if (this->print_flag) {
226         std::cout << "Production object constructed at " << this << std::endl;
227     }
228
229     return;
230 } /* Production() */

```

4.11.2.3 ~Production()

```

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

```

Destructor for the [Production](#) class.

```

380 {
381     // 1. destruction print
382     if (this->print_flag) {
383         std::cout << "Production object at " << this << " destroyed" << std::endl;
384     }
385
386     return;
387 } /* ~Production() */

```

4.11.3 Member Function Documentation

4.11.3.1 __checkInputs()

```

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

```

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

Parameters

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

```

41 {
42     // 1. check n_points
43     if (n_points <= 0) {
44         std::string error_str = "ERROR: Production(): n_points must be > 0";
45
46         #ifdef _WIN32
47             std::cout << error_str << std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     // 2. check capacity_kW
54     if (production_inputs.capacity_kW <= 0) {
55         std::string error_str = "ERROR: Production(): ";
56         error_str += "ProductionInputs::capacity_kW must be > 0";
57
58         #ifdef _WIN32
59             std::cout << error_str << std::endl;
60         #endif
61
62         throw std::invalid_argument(error_str);
63     }
64
65     // 3. check replace_running_hrs
66     if (production_inputs.replace_running_hrs <= 0) {
67         std::string error_str = "ERROR: Production(): ";
68         error_str += "ProductionInputs::replace_running_hrs must be > 0";
69
70         #ifdef _WIN32
71             std::cout << error_str << std::endl;
72         #endif
73
74         throw std::invalid_argument(error_str);
75     }
76
77     return;
78 } /* __checkInputs() */

```

4.11.3.2 __computeRealDiscountAnnual()

```

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

```

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

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

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

Parameters

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

Returns

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

```

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

```

4.11.3.3 __handleReplacement()

```

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

```

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

Parameters

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

```

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

```

4.11.3.4 commit()

```

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

```

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

Parameters

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

Returns

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

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

```

323 {
324     // 1. record production
325     this->production_vec_kW[timestep] = production_kW;
326
327     // 2. compute and record dispatch and curtailment
328     double dispatch_kW = 0;
329     double curtailment_kW = 0;
330
331     if (production_kW > load_kW) {
332         dispatch_kW = load_kW;
333         curtailment_kW = production_kW - dispatch_kW;
334     }
335
336     else {
337         dispatch_kW = production_kW;
338     }
339
340     this->dispatch_vec_kW[timestep] = dispatch_kW;
341     this->total_dispatch_kWh += dispatch_kW * dt_hrs;
342     this->curtailment_vec_kW[timestep] = curtailment_kW;
343
344     // 3. update load
345     load_kW -= dispatch_kW;
346
347     if (this->is_running) {
348         // 4. log running state, running hours
349         this->is_running_vec[timestep] = this->is_running;
350         this->running_hours += dt_hrs;
351
352         // 5. incur operation and maintenance costs
353         double produced_kWh = production_kW * dt_hrs;
354
355         double operation_maintenance_cost =
356             this->operation_maintenance_cost_kWh * produced_kWh;
357         this->operation_maintenance_cost_vec[timestep] = operation_maintenance_cost;
358
359         // 6. incur capital costs (i.e., handle replacement)
360         this->__handleReplacement(timestep);
361     }
362
363
364     return load_kW;
365 } /* commit() */

```

4.11.3.5 computeEconomics()

```

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

```

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

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

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

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

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

Parameters

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

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

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

```

252 {

```

```

253 // 1. compute net present cost
254 double t_hrs = 0;
255 double real_discount_scalar = 0;
256
257 for (int i = 0; i < this->n_points; i++) {
258     t_hrs = time_vec_hrs_ptr->at(i);
259
260     real_discount_scalar = 1.0 / pow(
261         1 + this->real_discount_annual,
262         t_hrs / 8760
263     );
264
265     this->net_present_cost += real_discount_scalar * this->capital_cost_vec[i];
266
267     this->net_present_cost +=
268         real_discount_scalar * this->operation_maintenance_cost_vec[i];
269 }
270
271 // assuming 8,760 hours per year
272 double n_years = time_vec_hrs_ptr->at(this->n_points - 1) / 8760;
273
274 double capital_recovery_factor =
275     (this->real_discount_annual * pow(1 + this->real_discount_annual, n_years)) /
276     (pow(1 + this->real_discount_annual, n_years) - 1);
277
278 double total_annualized_cost = capital_recovery_factor *
279     this->net_present_cost;
280
281 this->levelled_cost_of_energy_kWh =
282     (n_years * total_annualized_cost) /
283     total_dispatch_kWh;
284
285 return;
286 } /* computeEconomics() */

```

4.11.4 Member Data Documentation

4.11.4.1 capacity_kW

```
double Production::capacity_kW
```

The rated production capacity [kW] of the asset.

4.11.4.2 capital_cost

```
double Production::capital_cost
```

The capital cost of the asset (undefined currency).

4.11.4.3 capital_cost_vec

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

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

4.11.4.4 curtailment_vec_kW

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

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

4.11.4.5 dispatch_vec_kW

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

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

4.11.4.6 is_running

```
bool Production::is_running
```

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

4.11.4.7 is_running_vec

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

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

4.11.4.8 is_sunk

```
bool Production::is_sunk
```

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

4.11.4.9 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 dispatched and stored energy.

4.11.4.10 n_points

```
int Production::n_points
```

The number of points in the modelling time series.

4.11.4.11 n_replacements

```
int Production::n_replacements
```

The number of times the asset has been replaced.

4.11.4.12 n_starts

```
int Production::n_starts
```

The number of times the asset has been started.

4.11.4.13 net_present_cost

```
double Production::net_present_cost
```

The net present cost of this asset.

4.11.4.14 operation_maintenance_cost_kWh

```
double Production::operation_maintenance_cost_kWh
```

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

4.11.4.15 operation_maintenance_cost_vec

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

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

4.11.4.16 print_flag

```
bool Production::print_flag
```

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

4.11.4.17 production_vec_kW

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

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

4.11.4.18 real_discount_annual

```
double Production::real_discount_annual
```

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

4.11.4.19 replace_running_hrs

```
double Production::replace_running_hrs
```

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

4.11.4.20 running_hours

```
double Production::running_hours
```

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

4.11.4.21 storage_vec_kW

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

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

4.11.4.22 total_dispatch_kWh

```
double Production::total_dispatch_kWh
```

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

4.11.4.23 type_str

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

A string describing the type of the asset.

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

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

4.12 ProductionInputs Struct Reference

A structure which bundles the necessary inputs for the [Production](#) constructor. Provides default values for every necessary input.

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

Public Attributes

- bool [print_flag](#) = false
A flag which indicates whether or not object construct/destruction should be verbose.
- bool [is_sunk](#) = false
A boolean which indicates whether or not the asset should be considered a sunk cost (i.e., capital cost incurred at the start of the model, or no).
- double [capacity_kW](#) = 100
The rated production capacity [kW] of the asset.
- double [nominal_inflation_annual](#) = 0.02
The nominal, annual inflation rate to use in computing model economics.
- double [nominal_discount_annual](#) = 0.04
The nominal, annual discount rate to use in computing model economics.
- double [replace_running_hrs](#) = 90000
The number of running hours after which the asset must be replaced.

4.12.1 Detailed Description

A structure which bundles the necessary inputs for the [Production](#) constructor. Provides default values for every necessary input.

4.12.2 Member Data Documentation

4.12.2.1 capacity_kW

```
double ProductionInputs::capacity_kW = 100
```

The rated production capacity [kW] of the asset.

4.12.2.2 is_sunk

```
bool ProductionInputs::is_sunk = false
```

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

4.12.2.3 nominal_discount_annual

```
double ProductionInputs::nominal_discount_annual = 0.04
```

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

4.12.2.4 nominal_inflation_annual

```
double ProductionInputs::nominal_inflation_annual = 0.02
```

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

4.12.2.5 print_flag

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

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

4.12.2.6 replace_running_hrs

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

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

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

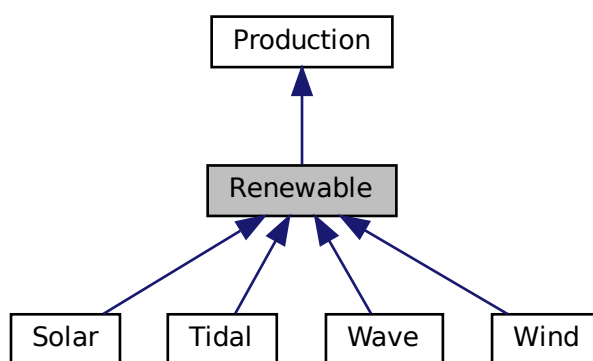
- header/Production/[Production.h](#)

4.13 Renewable Class Reference

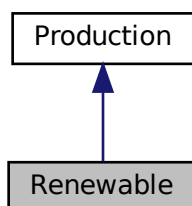
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

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

Inheritance diagram for Renewable:



Collaboration diagram for Renewable:



Public Member Functions

- [Renewable](#) (void)
Constructor (dummy) for the [Renewable](#) class.
- [Renewable](#) (int, [RenewableInputs](#))
- 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.
- 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.

4.13.1 Detailed Description

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

4.13.2 Constructor & Destructor Documentation

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

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

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

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

Parameters

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

```

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

```

4.13.2.2 Renewable() [2/2]

```

Renewable::Renewable (
    int n_points,
    RenewableInputs renewable_inputs )
114 :
115     Production(n_points, renewable_inputs.production_inputs)
116 {
117     // 1. check inputs
118     this->__checkInputs(renewable_inputs);
119
120     // 2. set attributes
121     //...
122
123     // 3. construction print
124     if (this->print_flag) {
125         std::cout << "Renewable object constructed at " << this << std::endl;
126     }
127
128     return;
129 } /* Renewable() */

```

4.13.2.3 ~Renewable()

```

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

```

Destructor for the [Renewable](#) class.

```

218 {
219     // 1. destruction print
220     if (this->print_flag) {
221         std::cout << "Renewable object at " << this << " destroyed" << std::endl;
222     }
223
224     return;
225 } /* ~Renewable() */

```

4.13.3 Member Function Documentation

4.13.3.1 __checkInputs()

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

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

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

4.13.3.2 __handleStartStop()

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

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

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

4.13.3.3 commit()

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

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

Parameters

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

Returns

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

Reimplemented from [Production](#).

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

```

187 {
188     // 1. handle start/stop
189     this->__handleStartStop(timestep, dt_hrs, production_kW);
190
191     // 2. invoke base class method
192     load_kW = Production::commit(
193         timestep,
194         dt_hrs,
195         production_kW,
196         load_kW
197     );
198
199
200     //...
201
202     return load_kW;
203 } /* commit() */

```

4.13.3.4 computeEconomics()

```

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

```

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

Parameters

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

Reimplemented from [Production](#).

```

146 {
147     // 1. invoke base class method
148     Production::computeEconomics(time_vec_hrs_ptr);
149
150     return;
151 } /* computeEconomics() */

```

4.13.3.5 computeProductionkW() [1/2]

```

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

```

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

```

86 {return 0;}

```

4.13.3.6 computeProductionkW() [2/2]

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

Reimplemented in [Wave](#).

```
87 {return 0;}
```

4.13.4 Member Data Documentation

4.13.4.1 resource_key

```
int Renewable::resource_key
```

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

4.13.4.2 type

```
RenewableType Renewable::type
```

The type (RenewableType) of the asset.

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

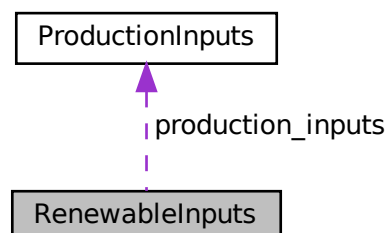
- header/Production/Renewable/[Renewable.h](#)
- source/Production/Renewable/[Renewable.cpp](#)

4.14 RenewableInputs Struct Reference

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

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

Collaboration diagram for RenewableInputs:



Public Attributes

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

4.14.1 Detailed Description

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

4.14.2 Member Data Documentation

4.14.2.1 `production_inputs`

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

An encapsulated [ProductionInputs](#) instance.

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

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

4.15 Resources Class Reference

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

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

Public Member Functions

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

Public Attributes

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

Private Member Functions

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

4.15.1 Detailed Description

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

4.15.2 Constructor & Destructor Documentation

4.15.2.1 Resources()

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

Constructor for the [Resources](#) class.

```
569 {
570     return;
571 } /* Resources() */
```

4.15.2.2 ~Resources()

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

Destructor for the [Resources](#) class.

```
711 {
712     this->clear();
713     return;
714 } /* ~Resources() */
```

4.15.3 Member Function Documentation

4.15.3.1 __checkResourceKey1D()

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

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

Parameters

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

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



```
87 } /* __checkResourceKey1D() */
```

4.15.3.2 __checkResourceKey2D()

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

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

Parameters

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

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

4.15.3.3 __checkTimePoint()

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

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

Parameters

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

```

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

```

4.15.3.4 __readSolarResource()

```

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

```

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

Parameters

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

```

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

```

```

295         );
296
297         this->resource_map_1D[resource_key][n_points] = solar_resource_kWm2;
298
299         n_points++;
300     }
301
302     // 4. check data length
303     if (n_points != electrical_load_ptr->n_points) {
304         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
305     }
306
307     return;
308 } /* __readSolarResource() */

```

4.15.3.5 __readTidalResource()

```

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

```

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

Parameters

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

```

337 {
338     // 1. init CSV reader, record path
339     io::CSVReader<2> CSV(path_2_resource_data);
340
341     CSV.read_header(
342         io::ignore_extra_column,
343         "Time (since start of data) [hrs]",
344         "Tidal Speed (hub depth) [m/s]"
345     );
346
347     this->path_map_1D.insert(
348         std::pair<int, std::string>(resource_key, path_2_resource_data)
349     );
350
351     // 2. init map element
352     this->resource_map_1D.insert(
353         std::pair<int, std::vector<double>>(resource_key, {})
354     );
355     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
356
357
358     // 3. read in resource data, check against time series (point-wise and length)
359     int n_points = 0;
360     double time_hrs = 0;
361     double time_expected_hrs = 0;
362     double tidal_resource_ms = 0;
363
364     while (CSV.read_row(time_hrs, tidal_resource_ms)) {
365         if (n_points > electrical_load_ptr->n_points) {
366             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
367         }
368
369         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
370         this->__checkTimePoint(
371             time_hrs,
372             time_expected_hrs,
373             path_2_resource_data,
374             electrical_load_ptr
375         );
376
377         this->resource_map_1D[resource_key][n_points] = tidal_resource_ms;
378     }

```

```

379         n_points++;
380     }
381
382     // 4. check data length
383     if (n_points != electrical_load_ptr->n_points) {
384         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
385     }
386
387     return;
388 } /* __readTidalResource() */

```

4.15.3.6 __readWaveResource()

```

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

```

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

Parameters

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

```

417 {
418     // 1. init CSV reader, record path
419     io::CSVReader<3> CSV(path_2_resource_data);
420
421     CSV.read_header(
422         io::ignore_extra_column,
423         "Time (since start of data) [hrs]",
424         "Significant Wave Height [m]",
425         "Energy Period [s]"
426     );
427
428     this->path_map_2D.insert(
429         std::pair<int, std::string>(resource_key, path_2_resource_data)
430     );
431
432     // 2. init map element
433     this->resource_map_2D.insert(
434         std::pair<int, std::vector<std::vector<double>>>(resource_key, {})
435     );
436     this->resource_map_2D[resource_key].resize(electrical_load_ptr->n_points, {0, 0});
437
438
439     // 3. read in resource data, check against time series (point-wise and length)
440     int n_points = 0;
441     double time_hrs = 0;
442     double time_expected_hrs = 0;
443     double significant_wave_height_m = 0;
444     double energy_period_s = 0;
445
446     while (CSV.read_row(time_hrs, significant_wave_height_m, energy_period_s)) {
447         if (n_points > electrical_load_ptr->n_points) {
448             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
449         }
450
451         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
452         this->__checkTimePoint(
453             time_hrs,
454             time_expected_hrs,
455             path_2_resource_data,
456             electrical_load_ptr
457         );
458
459         this->resource_map_2D[resource_key][n_points][0] = significant_wave_height_m;
460         this->resource_map_2D[resource_key][n_points][1] = energy_period_s;
461
462         n_points++;

```

```

463     }
464
465     // 4. check data length
466     if (n_points != electrical_load_ptr->n_points) {
467         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
468     }
469
470     return;
471 } /* __readWaveResource() */

```

4.15.3.7 __readWindResource()

```

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

```

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

Parameters

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

```

500 {
501     // 1. init CSV reader, record path
502     io::CSVReader<2> CSV(path_2_resource_data);
503
504     CSV.read_header(
505         io::ignore_extra_column,
506         "Time (since start of data) [hrs]",
507         "Wind Speed (hub height) [m/s]"
508     );
509
510     this->path_map_1D.insert(
511         std::pair<int, std::string>(resource_key, path_2_resource_data)
512     );
513
514     // 2. init map element
515     this->resource_map_1D.insert(
516         std::pair<int, std::vector<double>>(resource_key, {})
517     );
518     this->resource_map_1D[resource_key].resize(electrical_load_ptr->n_points, 0);
519
520
521     // 3. read in resource data, check against time series (point-wise and length)
522     int n_points = 0;
523     double time_hrs = 0;
524     double time_expected_hrs = 0;
525     double wind_resource_ms = 0;
526
527     while (CSV.read_row(time_hrs, wind_resource_ms)) {
528         if (n_points > electrical_load_ptr->n_points) {
529             this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
530         }
531
532         time_expected_hrs = electrical_load_ptr->time_vec_hrs[n_points];
533         this->__checkTimePoint(
534             time_hrs,
535             time_expected_hrs,
536             path_2_resource_data,
537             electrical_load_ptr
538         );
539
540         this->resource_map_1D[resource_key][n_points] = wind_resource_ms;
541         n_points++;
542     }
543
544     // 4. check data length
545     if (n_points != electrical_load_ptr->n_points) {

```

```

547         this->__throwLengthError(path_2_resource_data, electrical_load_ptr);
548     }
549
550     return;
551 } /* __readWindResource() */

```

4.15.3.8 __throwLengthError()

```

void Resources::__throwLengthError (
    std::string path_2_resource_data,
    ElectricalLoad * electrical_load_ptr ) [private]

```

Helper method to throw data length error.

Parameters

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

```

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

```

4.15.3.9 addResource()

```

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

```

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

Parameters

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

```

608 {
609     switch (renewable_type) {
610         case (RenewableType :: SOLAR): {
611             this->__checkResourceKey1D(resource_key, renewable_type);
612
613             this->__readSolarResource(
614                 path_2_resource_data,
615                 resource_key,
616                 electrical_load_ptr
617             );
618
619             break;
620         }
621
622         case (RenewableType :: TIDAL): {
623             this->__checkResourceKey1D(resource_key, renewable_type);
624
625             this->__readTidalResource(
626                 path_2_resource_data,
627                 resource_key,
628                 electrical_load_ptr
629             );
630
631             break;
632         }
633
634         case (RenewableType :: WAVE): {
635             this->__checkResourceKey2D(resource_key, renewable_type);
636
637             this->__readWaveResource(
638                 path_2_resource_data,
639                 resource_key,
640                 electrical_load_ptr
641             );
642
643             break;
644         }
645
646         case (RenewableType :: WIND): {
647             this->__checkResourceKey1D(resource_key, renewable_type);
648
649             this->__readWindResource(
650                 path_2_resource_data,
651                 resource_key,
652                 electrical_load_ptr
653             );
654
655             break;
656         }
657
658         default: {
659             std::string error_str = "ERROR: Resources :: addResource(: ";
660             error_str += "renewable type ";
661             error_str += std::to_string(renewable_type);
662             error_str += " not recognized";
663
664             #ifdef _WIN32
665                 std::cout << error_str << std::endl;
666             #endif
667
668             throw std::runtime_error(error_str);
669
670             break;
671         }
672     }
673
674     return;
675 } /* addResource() */

```

4.15.3.10 clear()

```

void Resources::clear (
    void )

```

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

```

689 {
690     this->resource_map_1D.clear();
691     this->path_map_1D.clear();

```

```
692
693     this->resource_map_2D.clear();
694     this->path_map_2D.clear();
695
696     return;
697 } /* clear() */
```

4.15.4 Member Data Documentation

4.15.4.1 path_map_1D

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

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

4.15.4.2 path_map_2D

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

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

4.15.4.3 resource_map_1D

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

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

4.15.4.4 resource_map_2D

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

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

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

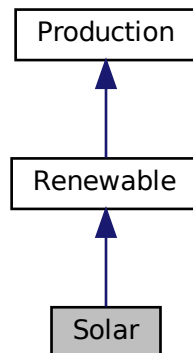
- header/[Resources.h](#)
- source/[Resources.cpp](#)

4.16 Solar Class Reference

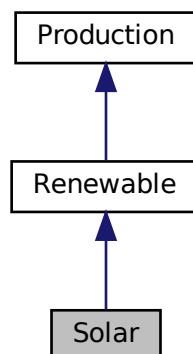
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

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

Inheritance diagram for Solar:



Collaboration diagram for Solar:



Public Member Functions

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

- 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.

4.16.1 Detailed Description

A derived class of the `Renewable` branch of `Production` which models solar production.

4.16.2 Constructor & Destructor Documentation

4.16.2.1 `Solar()` [1/2]

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

Constructor (dummy) for the `Solar` class.

Constructor (intended) for the `Solar` class.

Parameters

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

124 {

```

125     //...
126
127     return;
128 } /* Solar() */

```

4.16.2.2 Solar() [2/2]

```

Solar::Solar (
    int n_points,
    SolarInputs solar_inputs )
146 :
147 Renewable(n_points, solar_inputs.renewable_inputs)
148 {
149     // 1. check inputs
150     this->__checkInputs(solar_inputs);
151
152     // 2. set attributes
153     this->type = RenewableType :: SOLAR;
154     this->type_str = "SOLAR";
155
156     this->resource_key = solar_inputs.resource_key;
157
158     this->derating = solar_inputs.derating;
159
160     if (solar_inputs.capital_cost < 0) {
161         this->capital_cost = this->__getGenericCapitalCost();
162     }
163
164     if (solar_inputs.operation_maintenance_cost_kWh < 0) {
165         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
166     }
167
168     if (this->is_sunk) {
169         this->capital_cost_vec[0] = this->capital_cost;
170     }
171
172     // 3. construction print
173     if (this->print_flag) {
174         std::cout << "Solar object constructed at " << this << std::endl;
175     }
176
177     return;
178 } /* Renewable() */

```

4.16.2.3 ~Solar()

```

Solar::~~Solar (
    void )

```

Destructor for the `Solar` class.

```

291 {
292     // 1. destruction print
293     if (this->print_flag) {
294         std::cout << "Solar object at " << this << " destroyed" << std::endl;
295     }
296
297     return;
298 } /* ~Solar() */

```

4.16.3 Member Function Documentation

4.16.3.1 __checkInputs()

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

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

```
37 {
38     // 1. check derating
39     if (
40         solar_inputs.derating < 0 or
41         solar_inputs.derating > 1
42     ) {
43         std::string error_str = "ERROR: Solar(): ";
44         error_str += "SolarInputs::derating must be in the closed interval [0, 1]";
45
46         #ifdef WIN32
47             std::cout << error_str << std::endl;
48         #endif
49
50         throw std::invalid_argument(error_str);
51     }
52
53     return;
54 } /* __checkInputs() */
```

4.16.3.2 __getGenericCapitalCost()

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

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

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

Returns

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

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

4.16.3.3 __getGenericOpMaintCost()

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

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

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

Returns

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

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

4.16.3.4 commit()

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

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```
263 {
264     // 1. invoke base class method
265     load_kW = Renewable::commit(
266         timestep,
267         dt_hrs,
268         production_kW,
269         load_kW
270     );
271
272
273     //...
274
275     return load_kW;
276 } /* commit() */
```

4.16.3.5 computeProductionkW()

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

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

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

Parameters

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

Returns

The production [kW] of the solar PV array.

Reimplemented from [Renewable](#).

```
212 {
213     // check if no resource
214     if (solar_resource_kWm2 <= 0) {
215         return 0;
216     }
217
218     // compute production
219     double production_kW = this->derating * solar_resource_kWm2 * this->capacity_kW;
220
221     // cap production at capacity
222     if (production_kW > this->capacity_kW) {
223         production_kW = this->capacity_kW;
224     }
225
226     return production_kW;
227 } /* computeProductionkW() */
```

4.16.4 Member Data Documentation

4.16.4.1 derating

```
double Solar::derating
```

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

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

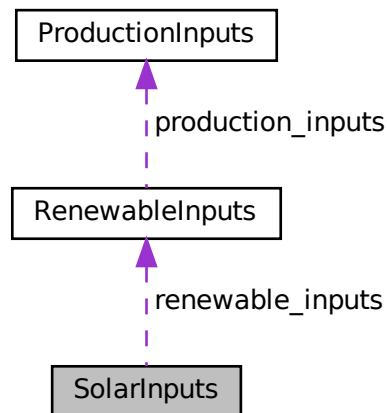
- header/Production/Renewable/[Solar.h](#)
- source/Production/Renewable/[Solar.cpp](#)

4.17 SolarInputs Struct Reference

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

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

Collaboration diagram for SolarInputs:



Public Attributes

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

4.17.1 Detailed Description

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

4.17.2 Member Data Documentation

4.17.2.1 capital_cost

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

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

4.17.2.2 derating

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

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

4.17.2.3 operation_maintenance_cost_kWh

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

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

4.17.2.4 renewable_inputs

```
RenewableInputs SolarInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.17.2.5 resource_key

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

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

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

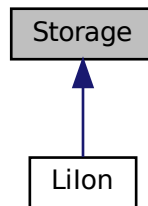
- [header/Production/Renewable/Solar.h](#)

4.18 Storage Class Reference

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

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

Inheritance diagram for Storage:



Public Member Functions

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

4.18.1 Detailed Description

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

4.18.2 Constructor & Destructor Documentation

4.18.2.1 Storage()

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

Constructor for the [Storage](#) class.

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

4.18.2.2 ~Storage()

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

Destructor for the [Storage](#) class.

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

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

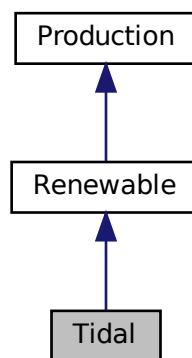
- [header/Storage/Storage.h](#)
- [source/Storage/Storage.cpp](#)

4.19 Tidal Class Reference

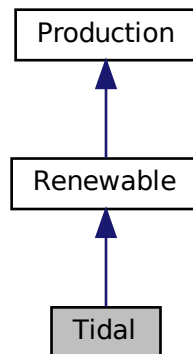
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

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

Inheritance diagram for Tidal:



Collaboration diagram for Tidal:



Public Member Functions

- [Tidal](#) (void)
Constructor (dummy) for the [Tidal](#) class.
- [Tidal](#) (int, [TidalInputs](#))
- 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.

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.

4.19.1 Detailed Description

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

4.19.2 Constructor & Destructor Documentation

4.19.2.1 Tidal() [1/2]

```
Tidal::Tidal (
    void )
```

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

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

Parameters

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

```
269 {
270     return;
271 } /* Tidal() */
```

4.19.2.2 Tidal() [2/2]

```
Tidal::Tidal (
    int n_points,
    TidalInputs tidal_inputs )
289 :
290 Renewable(n_points, tidal_inputs.renewable_inputs)
291 {
292     // 1. check inputs
293     this->__checkInputs(tidal_inputs);
294
295     // 2. set attributes
296     this->type = RenewableType :: TIDAL;
297     this->type_str = "TIDAL";
298
299     this->resource_key = tidal_inputs.resource_key;
300
301     this->design_speed_ms = tidal_inputs.design_speed_ms;
302
303     this->power_model = tidal_inputs.power_model;
```

```

304
305     if (tidal_inputs.capital_cost < 0) {
306         this->capital_cost = this->__getGenericCapitalCost();
307     }
308
309     if (tidal_inputs.operation_maintenance_cost_kWh < 0) {
310         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
311     }
312
313     if (this->is_sunk) {
314         this->capital_cost_vec[0] = this->capital_cost;
315     }
316
317     // 3. construction print
318     if (this->print_flag) {
319         std::cout << "Tidal object constructed at " << this << std::endl;
320     }
321
322     return;
323 } /* Renewable() */

```

4.19.2.3 ~Tidal()

```

Tidal::~Tidal (
    void )

```

Destructor for the [Tidal](#) class.

```

477 {
478     // 1. destruction print
479     if (this->print_flag) {
480         std::cout << "Tidal object at " << this << " destroyed" << std::endl;
481     }
482
483     return;
484 } /* ~Tidal() */

```

4.19.3 Member Function Documentation

4.19.3.1 __checkInputs()

```

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

```

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

```

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

```

4.19.3.2 `__computeCubicProductionkW()`

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

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

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

Parameters

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

Returns

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

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

4.19.3.3 `__computeExponentialProductionkW()`

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

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

Ref: [docs/refs/wind_tidal_wave.pdf](#)

Parameters

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

Returns

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

```

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

```

4.19.3.4 __computeLookupProductionkW()

```

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

```

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

Parameters

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

Returns

The interpolated production [kW] of the tidal tubrine.

```

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

```

4.19.3.5 `__getGenericCapitalCost()`

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

Helper method to generate a generic tidal turbine capital cost.

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

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

Returns

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

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

4.19.3.6 `__getGenericOpMaintCost()`

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

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

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

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

Returns

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

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

4.19.3.7 `commit()`

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

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```

449 {
450     // 1. invoke base class method
451     load_kW = Renewable::commit(
452         timestep,
453         dt_hrs,
454         production_kW,
455         load_kW
456     );
457
458
459     //...
460
461     return load_kW;
462 } /* commit() */

```

4.19.3.8 computeProductionkW()

```

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

```

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

Parameters

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

Returns

The production [kW] of the tidal turbine.

Reimplemented from [Renewable](#).

```

355 {
356     // check if no resource
357     if (tidal_resource_ms <= 0) {
358         return 0;
359     }
360
361     // compute production
362     double production_kW = 0;

```

```

363
364     switch (this->power_model) {
365         case (TidalPowerProductionModel :: TIDAL_POWER_CUBIC): {
366             production_kW = this->__computeCubicProductionkW(
367                 timestep,
368                 dt_hrs,
369                 tidal_resource_ms
370             );
371
372             break;
373         }
374
375         case (TidalPowerProductionModel :: TIDAL_POWER_EXPONENTIAL): {
376             production_kW = this->__computeExponentialProductionkW(
377                 timestep,
378                 dt_hrs,
379                 tidal_resource_ms
380             );
381
382             break;
383         }
384
385         case (TidalPowerProductionModel :: TIDAL_POWER_LOOKUP): {
386             production_kW = this->__computeLookupProductionkW(
387                 timestep,
388                 dt_hrs,
389                 tidal_resource_ms
390             );
391
392             break;
393         }
394
395         default: {
396             std::string error_str = "ERROR: Tidal::computeProductionkW(): ";
397             error_str += "power model ";
398             error_str += std::to_string(this->power_model);
399             error_str += " not recognized";
400
401             #ifdef _WIN32
402                 std::cout << error_str << std::endl;
403             #endif
404
405             throw std::runtime_error(error_str);
406
407             break;
408         }
409     }
410 }
411
412 return production_kW;
413 } /* computeProductionkW() */

```

4.19.4 Member Data Documentation

4.19.4.1 design_speed_ms

double Tidal::design_speed_ms

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

4.19.4.2 power_model

TidalPowerProductionModel Tidal::power_model

The tidal power production model to be applied.

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

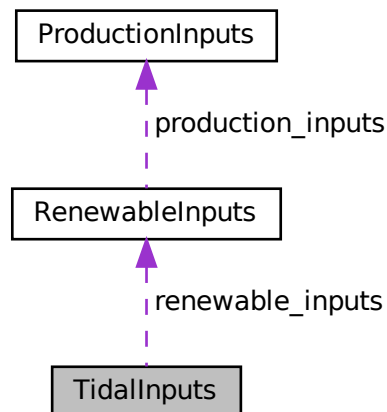
- header/Production/Renewable/Tidal.h
- source/Production/Renewable/Tidal.cpp

4.20 TidalInputs Struct Reference

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

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

Collaboration diagram for TidalInputs:



Public Attributes

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

4.20.1 Detailed Description

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

4.20.2 Member Data Documentation

4.20.2.1 capital_cost

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

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

4.20.2.2 design_speed_ms

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

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

4.20.2.3 operation_maintenance_cost_kWh

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

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

4.20.2.4 power_model

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

The tidal power production model to be applied.

4.20.2.5 renewable_inputs

```
RenewableInputs TidalInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.20.2.6 resource_key

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

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

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

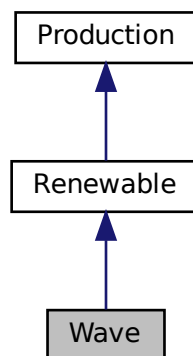
- [header/Production/Renewable/Tidal.h](#)

4.21 Wave Class Reference

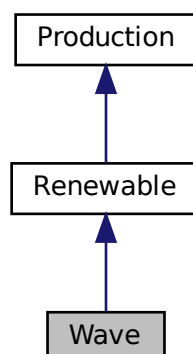
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

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

Inheritance diagram for Wave:



Collaboration diagram for Wave:



Public Member Functions

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

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.

4.21.1 Detailed Description

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

4.21.2 Constructor & Destructor Documentation

4.21.2.1 [Wave](#)() [1/2]

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

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

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

Parameters

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

```

299 {
300     return;
301 } /* Wave() */

```

4.21.2.2 Wave() [2/2]

```

Wave::Wave (
    int n_points,
    WaveInputs wave_inputs )
319     :
320     Renewable(n_points, wave_inputs.renewable_inputs)
321 {
322     // 1. check inputs
323     this->__checkInputs(wave_inputs);
324
325     // 2. set attributes
326     this->type = RenewableType :: WAVE;
327     this->type_str = "WAVE";
328
329     this->resource_key = wave_inputs.resource_key;
330
331     this->design_significant_wave_height_m =
332         wave_inputs.design_significant_wave_height_m;
333     this->design_energy_period_s = wave_inputs.design_energy_period_s;
334
335     this->power_model = wave_inputs.power_model;
336
337     if (wave_inputs.capital_cost < 0) {
338         this->capital_cost = this->__getGenericCapitalCost();
339     }
340
341     if (wave_inputs.operation_maintenance_cost_kWh < 0) {
342         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
343     }
344
345     if (this->is_sunk) {
346         this->capital_cost_vec[0] = this->capital_cost;
347     }
348
349     // 3. construction print
350     if (this->print_flag) {
351         std::cout << "Wave object constructed at " << this << std::endl;
352     }
353
354     return;
355 } /* Renewable() */

```

4.21.2.3 ~Wave()

```

Wave::~Wave (
    void )

```

Destructor for the [Wave](#) class.

```

515 {
516     // 1. destruction print
517     if (this->print_flag) {
518         std::cout << "Wave object at " << this << " destroyed" << std::endl;
519     }
520
521     return;
522 } /* ~Wave() */

```

4.21.3 Member Function Documentation

4.21.3.1 `__checkInputs()`

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

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

Parameters

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

```
39 {
40     // 1. check design_significant_wave_height_m
41     if (wave_inputs.design_significant_wave_height_m <= 0) {
42         std::string error_str = "ERROR: Wave(): ";
43         error_str += "WaveInputs::design_significant_wave_height_m must be > 0";
44
45         #ifdef WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     // 2. check design_energy_period_s
53     if (wave_inputs.design_energy_period_s <= 0) {
54         std::string error_str = "ERROR: Wave(): ";
55         error_str += "WaveInputs::design_energy_period_s must be > 0";
56
57         #ifdef WIN32
58             std::cout << error_str << std::endl;
59         #endif
60
61         throw std::invalid_argument(error_str);
62     }
63
64     return;
65 } /* __checkInputs() */
```

4.21.3.2 `__computeGaussianProductionkW()`

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

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

Ref: docs/refs/wind_tidal_wave.pdf

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under an exponential model.

```

160 {
161     double H_s_nondim =
162         (significant_wave_height_m - this->design_significant_wave_height_m) /
163         this->design_significant_wave_height_m;
164
165     double T_e_nondim =
166         (energy_period_s - this->design_energy_period_s) /
167         this->design_energy_period_s;
168
169     double production = exp(
170         -2.25119 * pow(T_e_nondim, 2) +
171         3.44570 * T_e_nondim * H_s_nondim -
172         4.01508 * pow(H_s_nondim, 2)
173     );
174
175     return production * this->capacity_kW;
176 } /* __computeGaussianProductionkW() */

```

4.21.3.3 __computeLookupProductionkW()

```

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

```

Helper method to compute wave energy converter production by way of looking up using given performance matrix.

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The interpolated production [kW] of the wave energy converter.

```

277 {
278     // *** WORK IN PROGRESS *** //
279
280     return 0;
281 } /* __computeLookupProductionkW() */

```

4.21.3.4 __computeParaboloidProductionkW()

```

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

```

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

Ref: [Robertson et al. \[2021\]](#)

Parameters

<i>timestep</i>	The current time step of the Model run.
<i>dt_hrs</i>	The interval of time [hrs] associated with the action.
<i>significant_wave_height_m</i>	The significant wave height [m] in the vicinity of the wave energy converter.
<i>energy_period_s</i>	The energy period [s] in the vicinity of the wave energy converter

Returns

The production [kW] of the wave energy converter, under a paraboloid model.

```

217 {
218     // first, check for idealized wave breaking (deep water)
219     if (significant_wave_height_m >= 0.2184 * pow(energy_period_s, 2)) {
220         return 0;
221     }
222
223     // otherwise, apply generic quadratic performance model
224     // (with outputs bounded to [0, 1])
225     double production =
226         0.289 * significant_wave_height_m -
227         0.00111 * pow(significant_wave_height_m, 2) * energy_period_s -
228         0.0169 * energy_period_s;
229
230     if (production < 0) {
231         production = 0;
232     }
233
234     else if (production > 1) {
235         production = 1;
236     }
237
238     return production * this->capacity_kW;
239 } /* __computeParaboloidProductionkW() */

```

4.21.3.5 __getGenericCapitalCost()

```

double Wave::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic wave energy converter capital cost.

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

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

Returns

A generic capital cost for the wave energy converter [CAD].

```

87 {
88     double capital_cost_per_kW = 7000 * pow(this->capacity_kW, -0.15) + 5000;
89
90     return capital_cost_per_kW * this->capacity_kW;
91 } /* __getGenericCapitalCost() */

```

4.21.3.6 `__getGenericOpMaintCost()`

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

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

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

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

Returns

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

```
115 {
116     double operation_maintenance_cost_kWh = 0.05 * pow(this->capacity_kW, -0.2) + 0.05;
117
118     return operation_maintenance_cost_kWh;
119 } /* __getGenericOpMaintCost() */
```

4.21.3.7 `commit()`

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

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```
487 {
488     // 1. invoke base class method
489     load_kW = Renewable::commit(
490         timestep,
491         dt_hrs,
492         production_kW,
493         load_kW
494     );
```

```

495
496
497     //...
498
499     return load_kW;
500 } /* commit() */

```

4.21.3.8 computeProductionkW()

```

double Wave::computeProductionkW (
    int timestep,
    double dt_hrs,
    double significant_wave_height_m,
    double energy_period_s ) [virtual]

```

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

Parameters

<i>timestep</i>	The timestep (i.e., time series index) for the request.
<i>dt_hrs</i>	The interval of time [hrs] associated with the timestep.
<i>signficiant_wave_height_m</i>	The significant wave height (wave statistic) [m].
<i>energy_period_s</i>	The energy period (wave statistic) [s].

Returns

The production [kW] of the wave turbine.

Reimplemented from [Renewable](#).

```

391 {
392     // check if no resource
393     if (significant_wave_height_m <= 0 or energy_period_s <= 0) {
394         return 0;
395     }
396
397     // compute production
398     double production_kW = 0;
399
400     switch (this->power_model) {
401         case (WavePowerProductionModel :: WAVE_POWER_PARABOLOID): {
402             production_kW = this->__computeParaboloidProductionkW(
403                 timestep,
404                 dt_hrs,
405                 significant_wave_height_m,
406                 energy_period_s
407             );
408
409             break;
410         }
411
412         case (WavePowerProductionModel :: WAVE_POWER_GAUSSIAN): {
413             production_kW = this->__computeGaussianProductionkW(
414                 timestep,
415                 dt_hrs,
416                 significant_wave_height_m,
417                 energy_period_s
418             );
419
420             break;
421         }
422
423         case (WavePowerProductionModel :: WAVE_POWER_LOOKUP): {
424             production_kW = this->__computeLookupProductionkW(

```

```

425         timestep,
426         dt_hrs,
427         significant_wave_height_m,
428         energy_period_s
429     );
430
431     break;
432 }
433
434 default: {
435     std::string error_str = "ERROR: Wave::computeProductionkW(): ";
436     error_str += "power model ";
437     error_str += std::to_string(this->power_model);
438     error_str += " not recognized";
439
440     #ifdef _WIN32
441         std::cout << error_str << std::endl;
442     #endif
443
444     throw std::runtime_error(error_str);
445
446     break;
447 }
448 }
449
450 return production_kW;
451 } /* computeProductionkW() */

```

4.21.4 Member Data Documentation

4.21.4.1 design_energy_period_s

```
double Wave::design_energy_period_s
```

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

4.21.4.2 design_significant_wave_height_m

```
double Wave::design_significant_wave_height_m
```

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

4.21.4.3 power_model

```
WavePowerProductionModel Wave::power_model
```

The wave power production model to be applied.

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

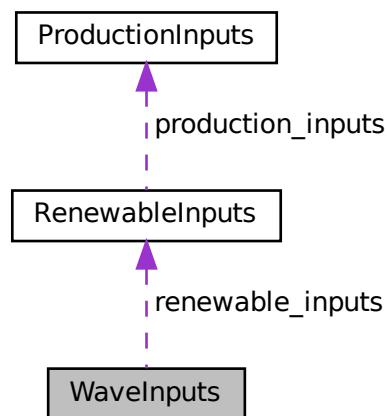
- header/Production/Renewable/[Wave.h](#)
- source/Production/Renewable/[Wave.cpp](#)

4.22 WaveInputs Struct Reference

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

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

Collaboration diagram for WaveInputs:



Public Attributes

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

4.22.1 Detailed Description

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

4.22.2 Member Data Documentation

4.22.2.1 capital_cost

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

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

4.22.2.2 design_energy_period_s

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

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

4.22.2.3 design_significant_wave_height_m

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

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

4.22.2.4 operation_maintenance_cost_kWh

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

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

4.22.2.5 power_model

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

The wave power production model to be applied.

4.22.2.6 renewable_inputs

```
RenewableInputs WaveInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.22.2.7 resource_key

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

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

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

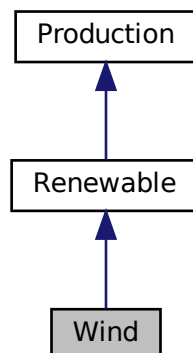
- [header/Production/Renewable/Wave.h](#)

4.23 Wind Class Reference

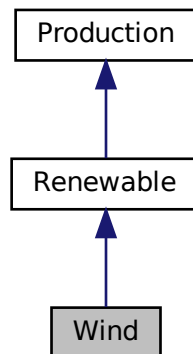
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

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

Inheritance diagram for Wind:



Collaboration diagram for Wind:



Public Member Functions

- [Wind](#) (void)
Constructor (dummy) for the [Wind](#) class.
- [Wind](#) (int, [WindInputs](#))
- 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.

Private Member Functions

- void [__checkInputs](#) ([WindInputs](#))
Helper method to check inputs to the [Wind](#) constructor.
- double [__getGenericCapitalCost](#) (void)
Helper method to generate a generic wind turbine capital cost.
- double [__getGenericOpMaintCost](#) (void)
Helper method to generate a generic wind turbine operation and maintenance cost. This is a cost incurred per unit energy produced.
- double [__computeExponentialProductionkW](#) (int, double, double)
Helper method to compute wind turbine production under an exponential production model.
- double [__computeLookupProductionkW](#) (int, double, double)
Helper method to compute wind turbine production by way of looking up using given power curve data.

4.23.1 Detailed Description

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

4.23.2 Constructor & Destructor Documentation

4.23.2.1 Wind() [1/2]

```
Wind::Wind (
    void )
```

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

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

Parameters

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

```
213 {
214     return;
215 } /* Wind() */
```

4.23.2.2 Wind() [2/2]

```
Wind::Wind (
    int n_points,
    WindInputs wind_inputs )
233 :
234 Renewable(n_points, wind_inputs.renewable_inputs)
235 {
236     // 1. check inputs
237     this->__checkInputs(wind_inputs);
238
239     // 2. set attributes
240     this->type = RenewableType :: WIND;
241     this->type_str = "WIND";
242
243     this->resource_key = wind_inputs.resource_key;
244
245     this->design_speed_ms = wind_inputs.design_speed_ms;
246
247     this->power_model = wind_inputs.power_model;
248
249     if (wind_inputs.capital_cost < 0) {
250         this->capital_cost = this->__getGenericCapitalCost();
251     }
252
253     if (wind_inputs.operation_maintenance_cost_kWh < 0) {
254         this->operation_maintenance_cost_kWh = this->__getGenericOpMaintCost();
255     }
256
257     if (this->is_sunk) {
258         this->capital_cost_vec[0] = this->capital_cost;
259     }
260
261     // 3. construction print
```

```

262     if (this->print_flag) {
263         std::cout << "Wind object constructed at " << this << std::endl;
264     }
265
266     return;
267 } /* Renewable() */

```

4.23.2.3 ~Wind()

```

Wind::~Wind (
    void )

```

Destructor for the [Wind](#) class.

```

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

```

4.23.3 Member Function Documentation

4.23.3.1 __checkInputs()

```

void Wind::__checkInputs (
    WindInputs wind_inputs ) [private]

```

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

Parameters

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

```

39 {
40     // 1. check design_speed_ms
41     if (wind_inputs.design_speed_ms <= 0) {
42         std::string error_str = "ERROR: Wind(): ";
43         error_str += "WindInputs::design_speed_ms must be > 0";
44
45         #ifdef WIN32
46             std::cout << error_str << std::endl;
47         #endif
48
49         throw std::invalid_argument(error_str);
50     }
51
52     return;
53 } /* __checkInputs() */

```

4.23.3.2 __computeExponentialProductionkW()

```

double Wind::__computeExponentialProductionkW (
    int timestep,

```

```
double dt_hrs,
double wind_resource_ms ) [private]
```

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

Ref: docs/refs/wind_tidal_wave.pdf

Parameters

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

Returns

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

```
140 {
141     double production = 0;
142
143     double turbine_speed = (wind_resource_ms - this->design_speed_ms) /
144         this->design_speed_ms;
145
146     if (turbine_speed < -0.76 or turbine_speed > 0.68) {
147         production = 0;
148     }
149
150     else if (turbine_speed >= -0.76 and turbine_speed <= 0) {
151         production = 1.03273 * exp(-5.97588 * pow(turbine_speed, 2)) - 0.03273;
152     }
153
154     else {
155         production = 0.16154 * exp(-9.30254 * pow(turbine_speed, 2)) + 0.83846;
156     }
157
158     return production * this->capacity_kW;
159 } /* __computeExponentialProductionkW() */
```

4.23.3.3 __computeLookupProductionkW()

```
double Wind::__computeLookupProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [private]
```

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

Parameters

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

Returns

The interpolated production [kW] of the wind turbine.

```

191 {
192     // *** WORK IN PROGRESS *** //
193
194     return 0;
195 } /* __computeLookupProductionkW() */

```

4.23.3.4 __getGenericCapitalCost()

```

double Wind::__getGenericCapitalCost (
    void ) [private]

```

Helper method to generate a generic wind turbine capital cost.

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

Returns

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

```

75 {
76     double capital_cost_per_kW = 3000 * pow(this->capacity_kW, -0.15) + 3000;
77
78     return capital_cost_per_kW * this->capacity_kW;
79 } /* __getGenericCapitalCost() */

```

4.23.3.5 __getGenericOpMaintCost()

```

double Wind::__getGenericOpMaintCost (
    void ) [private]

```

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

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

Returns

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

```

102 {
103     double operation_maintenance_cost_kWh = 0.025 * pow(this->capacity_kW, -0.2) + 0.025;
104
105     return operation_maintenance_cost_kWh;
106 } /* __getGenericOpMaintCost() */

```

4.23.3.6 commit()

```

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

```

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

Parameters

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

Returns

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

Reimplemented from [Renewable](#).

```

382 {
383     // 1. invoke base class method
384     load_kW = Renewable::commit(
385         timestep,
386         dt_hrs,
387         production_kW,
388         load_kW
389     );
390
391
392     //...
393
394     return load_kW;
395 } /* commit() */

```

4.23.3.7 computeProductionkW()

```

double Wind::computeProductionkW (
    int timestep,
    double dt_hrs,
    double wind_resource_ms ) [virtual]

```

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

Parameters

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

Returns

The production [kW] of the wind turbine.

Reimplemented from [Renewable](#).

```

299 {
300     // check if no resource
301     if (wind_resource_ms <= 0) {
302         return 0;
303     }
304
305     // compute production
306     double production_kW = 0;

```

```

307
308     switch (this->power_model) {
309         case (WindPowerProductionModel :: WIND_POWER_EXPONENTIAL): {
310             production_kW = this->__computeExponentialProductionkW(
311                 timestep,
312                 dt_hrs,
313                 wind_resource_ms
314             );
315
316             break;
317         }
318
319         case (WindPowerProductionModel :: WIND_POWER_LOOKUP): {
320             production_kW = this->__computeLookupProductionkW(
321                 timestep,
322                 dt_hrs,
323                 wind_resource_ms
324             );
325
326             break;
327         }
328
329         default: {
330             std::string error_str = "ERROR: Wind::computeProductionkW(): ";
331             error_str += "power model ";
332             error_str += std::to_string(this->power_model);
333             error_str += " not recognized";
334
335             #ifdef _WIN32
336                 std::cout << error_str << std::endl;
337             #endif
338
339             throw std::runtime_error(error_str);
340
341             break;
342         }
343     }
344
345     return production_kW;
346 } /* computeProductionkW() */

```

4.23.4 Member Data Documentation

4.23.4.1 design_speed_ms

double Wind::design_speed_ms

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

4.23.4.2 power_model

WindPowerProductionModel Wind::power_model

The wind power production model to be applied.

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

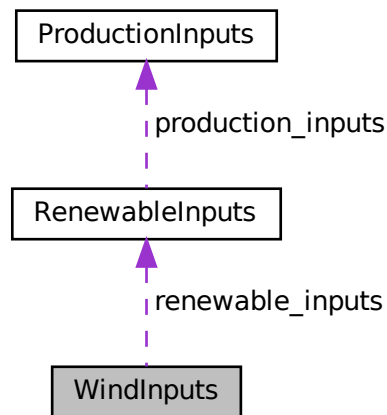
- [header/Production/Renewable/Wind.h](#)
- [source/Production/Renewable/Wind.cpp](#)

4.24 WindInputs Struct Reference

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

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

Collaboration diagram for WindInputs:



Public Attributes

- [RenewableInputs](#) `renewable_inputs`
An encapsulated [RenewableInputs](#) instance.
- int `resource_key` = 0
A key used to index into the [Resources](#) object, to associate this asset with the appropriate resource time series.
- double `capital_cost` = -1
The capital cost of the asset (undefined currency). -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD].
- double `operation_maintenance_cost_kWh` = -1
The operation and maintenance cost of the asset [1/kWh] (undefined currency). This is a cost incurred per unit of energy produced. -1 is a sentinel value, which triggers a generic cost model on construction (in fact, any negative value here will trigger). Note that the generic cost model is in terms of Canadian dollars [CAD/kWh].
- double `design_speed_ms` = 8
The wind speed [m/s] at which the wind turbine achieves its rated capacity.
- [WindPowerProductionModel](#) `power_model` = [WindPowerProductionModel](#) :: [WIND_POWER_EXPONENTIAL](#)
The wind power production model to be applied.

4.24.1 Detailed Description

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

4.24.2 Member Data Documentation

4.24.2.1 capital_cost

```
double WindInputs::capital_cost = -1
```

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

4.24.2.2 design_speed_ms

```
double WindInputs::design_speed_ms = 8
```

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

4.24.2.3 operation_maintenance_cost_kWh

```
double WindInputs::operation_maintenance_cost_kWh = -1
```

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

4.24.2.4 power_model

```
WindPowerProductionModel WindInputs::power_model = WindPowerProductionModel :: WIND_POWER_EXPONENTIAL
```

The wind power production model to be applied.

4.24.2.5 renewable_inputs

```
RenewableInputs WindInputs::renewable_inputs
```

An encapsulated [RenewableInputs](#) instance.

4.24.2.6 resource_key

```
int WindInputs::resource_key = 0
```

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

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

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

Chapter 5

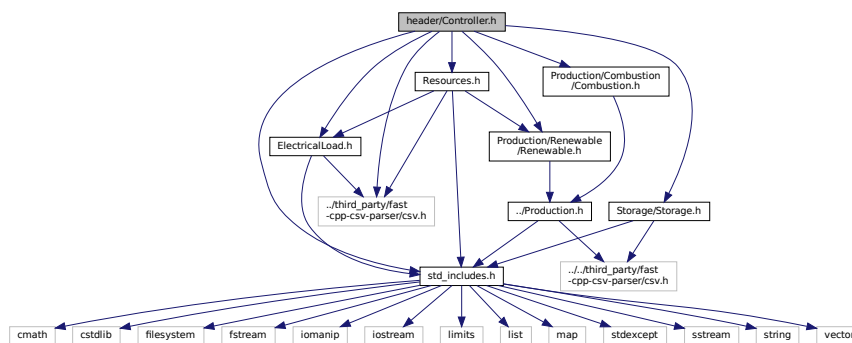
File Documentation

5.1 header/Controller.h File Reference

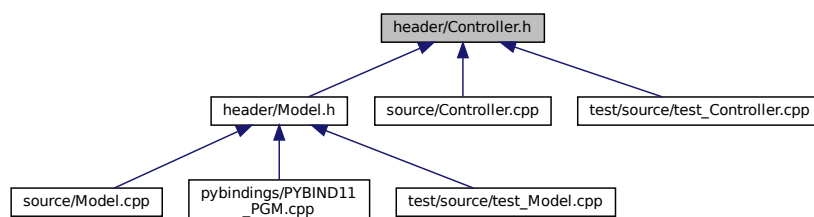
Header file the [Controller](#) class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Resources.h"
#include "Production/Combustion/Combustion.h"
#include "Production/Renewable/Renewable.h"
#include "Storage/Storage.h"
```

Include dependency graph for Controller.h:



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



Classes

- class [Controller](#)

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

Enumerations

- enum [ControlMode](#) { [LOAD_FOLLOWING](#) , [CYCLE_CHARGING](#) , [N_CONTROL_MODES](#) }

An enumeration of the types of control modes supported by PGMcpp.

5.1.1 Detailed Description

Header file the [Controller](#) class.

5.1.2 Enumeration Type Documentation

5.1.2.1 ControlMode

enum [ControlMode](#)

An enumeration of the types of control modes supported by PGMcpp.

Enumerator

LOAD_FOLLOWING	Load following control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
CYCLE_CHARGING	Cycle charging control, with in-order dispatch of non-Combustion assets and optimal dispatch of Combustion assets.
N_CONTROL_MODES	A simple hack to get the number of elements in ControlMode.

```

43     {
44         LOAD\_FOLLOWING,
45         CYCLE\_CHARGING,
46         N\_CONTROL\_MODES
47     };

```

5.2 header/ElectricalLoad.h File Reference

Header file the [ElectricalLoad](#) class.

```

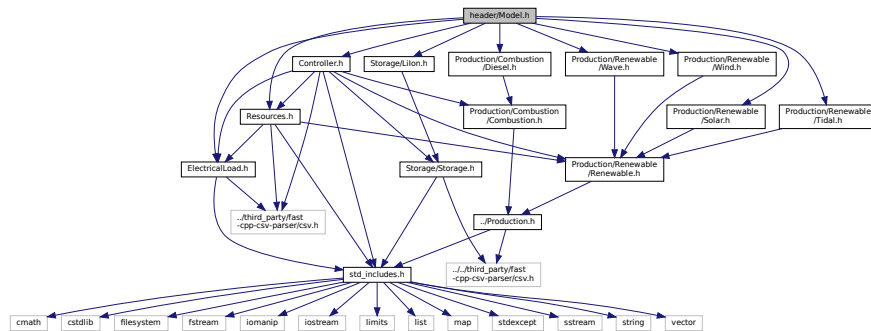
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"

```

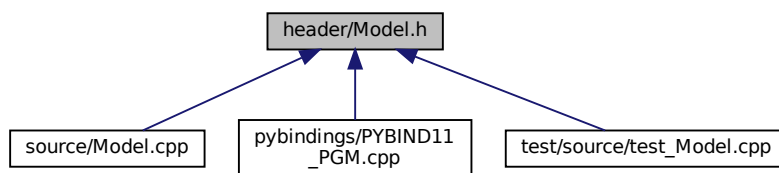


```
#include "Storage/LiIon.h"
```

Include dependency graph for Model.h:



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



Classes

- struct [ModellInputs](#)

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

- class [Model](#)

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

5.3.1 Detailed Description

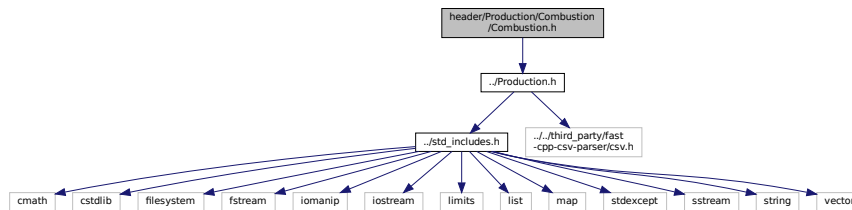
Header file the [Model](#) class.

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

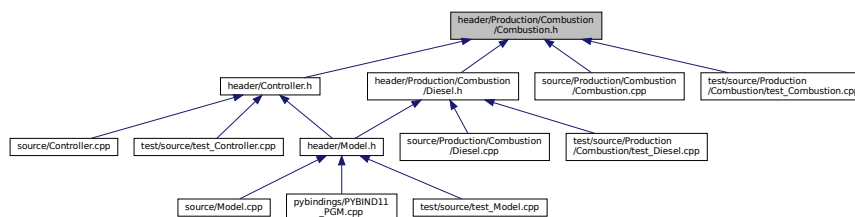
Header file the [Combustion](#) class.

```
#include "../Production.h"
```

Include dependency graph for Combustion.h:



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



Classes

- struct [CombustionInputs](#)
A structure which bundles the necessary inputs for the [Combustion](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [ProductionInputs](#).
- struct [Emissions](#)
A structure which bundles the emitted masses of various emissions chemistries.
- class [Combustion](#)
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

Enumerations

- enum [CombustionType](#) { DIESEL , N_COMBUSTION_TYPES }
An enumeration of the types of [Combustion](#) asset supported by PGMcpp.

5.4.1 Detailed Description

Header file the [Combustion](#) class.

5.4.2 Enumeration Type Documentation

5.4.2.1 CombustionType

```
enum CombustionType
```

An enumeration of the types of [Combustion](#) asset supported by PGMcpp.

Enumerator

DIESEL	A diesel generator.
N_COMBUSTION_TYPES	A simple hack to get the number of elements in CombustionType.

```

33         {
34     DIESEL,
35     N_COMBUSTION_TYPES
36 };

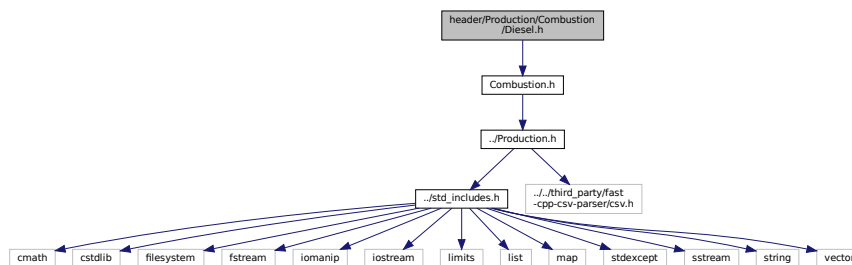
```

5.5 header/Production/Combustion/Diesel.h File Reference

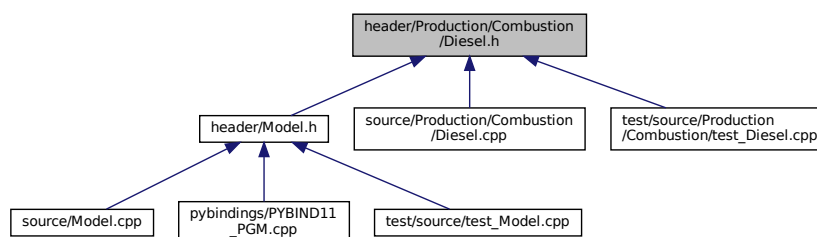
Header file the [Diesel](#) class.

```
#include "Combustion.h"
```

Include dependency graph for Diesel.h:



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



Classes

- struct [DieselInputs](#)

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

- class [Diesel](#)

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

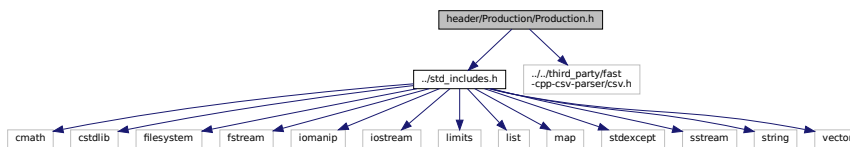
5.5.1 Detailed Description

Header file the **Diesel** class.

5.6 header/Production/Production.h File Reference

Header file the **Production** class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
Include dependency graph for Production.h:
```



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



Classes

- struct **ProductionInputs**
*A structure which bundles the necessary inputs for the **Production** constructor. Provides default values for every necessary input.*
- class **Production**
*The base class of the **Production** hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.*

5.6.1 Detailed Description

Header file the **Production** class.

Enumerator

SOLAR	A solar photovoltaic (PV) array.
TIDAL	A tidal stream turbine (or tidal energy converter, TEC)
WAVE	A wave energy converter (WEC)
WIND	A wind turbine.
N_RENEWABLE_TYPES	A simple hack to get the number of elements in RenewableType.

```

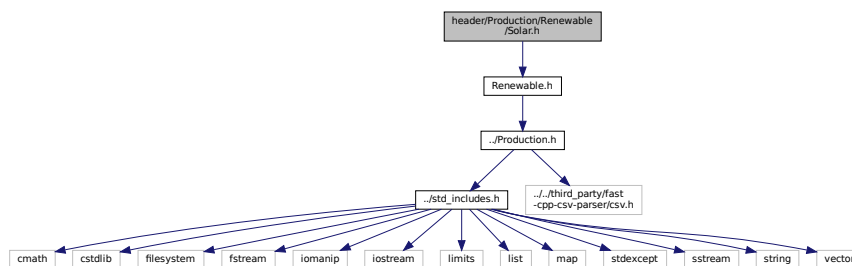
33     {
34     SOLAR,
35     TIDAL,
36     WAVE,
37     WIND,
38     N_RENEWABLE_TYPES
39 };

```

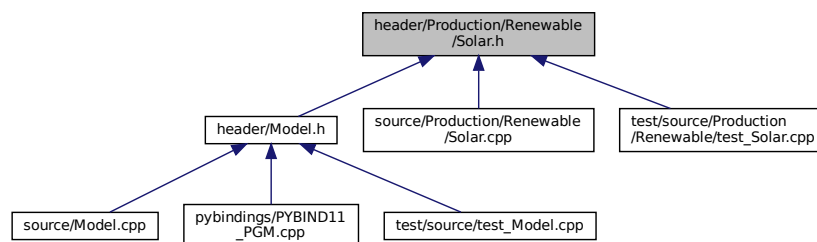
5.8 header/Production/Renewable/Solar.h File Reference

Header file the [Solar](#) class.

#include "Renewable.h"
 Include dependency graph for Solar.h:



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



Classes

- struct [SolarInputs](#)

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

- class [Solar](#)

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

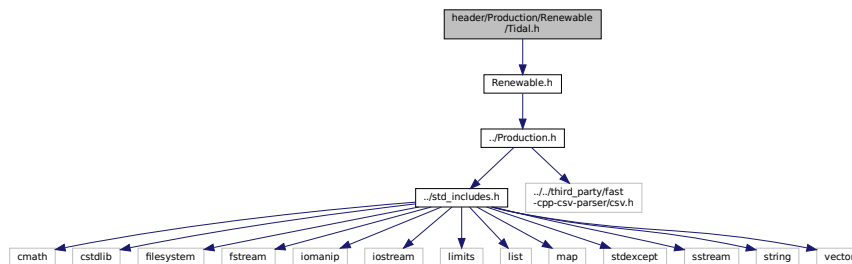
5.8.1 Detailed Description

Header file the [Solar](#) class.

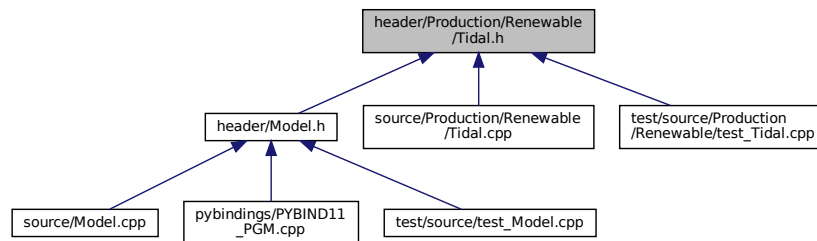
5.9 header/Production/Renewable/Tidal.h File Reference

Header file the [Tidal](#) class.

```
#include "Renewable.h"
Include dependency graph for Tidal.h:
```



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



Classes

- struct [TidalInputs](#)

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

- class [Tidal](#)

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

Enumerations

- enum [TidalPowerProductionModel](#) { [TIDAL_POWER_CUBIC](#) , [TIDAL_POWER_EXPONENTIAL](#) , [TIDAL_POWER_LOOKUP](#) , [N_TIDAL_POWER_PRODUCTION_MODELS](#) }

5.9.1 Detailed Description

Header file the [Tidal](#) class.

5.9.2 Enumeration Type Documentation

5.9.2.1 TidalPowerProductionModel

enum [TidalPowerProductionModel](#)

Enumerator

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

```

34
35     TIDAL_POWER_CUBIC,
36     TIDAL_POWER_EXPONENTIAL,
37     TIDAL_POWER_LOOKUP,
38     N_TIDAL_POWER_PRODUCTION_MODELS
39 };

```

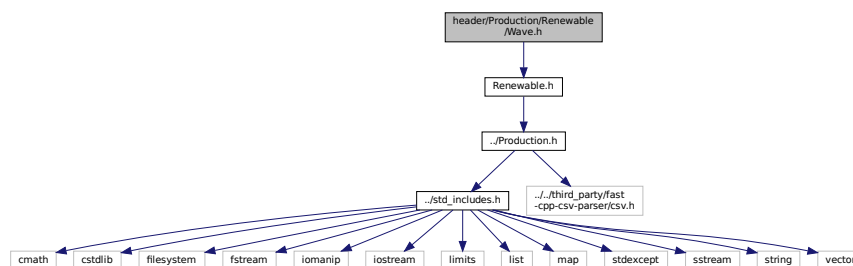
5.10 header/Production/Renewable/Wave.h File Reference

Header file the [Wave](#) class.

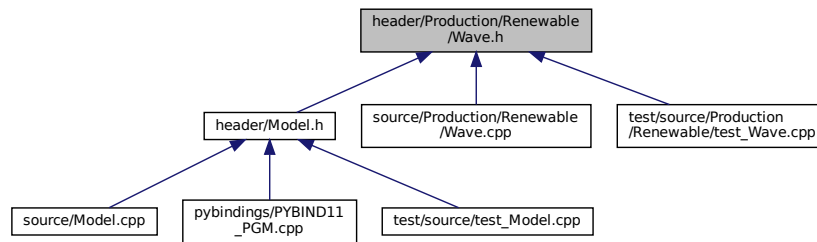
```

#include "Renewable.h"
Include dependency graph for Wave.h:

```



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



Classes

- struct [WaveInputs](#)
A structure which bundles the necessary inputs for the [Wave](#) constructor. Provides default values for every necessary input. Note that this structure encapsulates [RenewableInputs](#).
- class [Wave](#)
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

Enumerations

- enum [WavePowerProductionModel](#) { [WAVE_POWER_GAUSSIAN](#) , [WAVE_POWER_PARABOLOID](#) , [WAVE_POWER_LOOKUP](#) , [N_WAVE_POWER_PRODUCTION_MODELS](#) }

5.10.1 Detailed Description

Header file the [Wave](#) class.

5.10.2 Enumeration Type Documentation

5.10.2.1 WavePowerProductionModel

enum [WavePowerProductionModel](#)

Enumerator

WAVE_POWER_GAUSSIAN	A Gaussian power production model.
WAVE_POWER_PARABOLOID	A paraboloid power production model.
WAVE_POWER_LOOKUP	Lookup from a given performance matrix.
N_WAVE_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in WavePowerProductionModel .

```

34     {
35     WAVE_POWER_GAUSSIAN,
36     WAVE_POWER_PARABOLOID,
37     WAVE_POWER_LOOKUP,
38     N_WAVE_POWER_PRODUCTION_MODELS
39 };

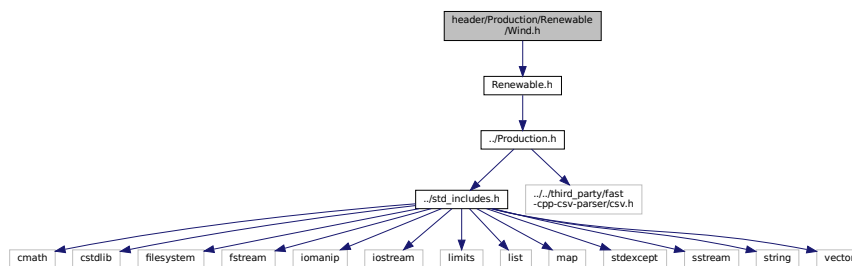
```

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

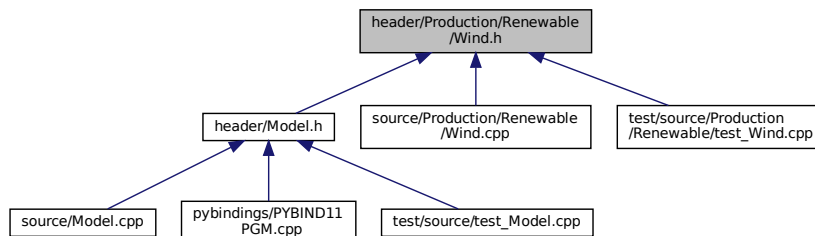
Header file the [Wind](#) class.

```
#include "Renewable.h"
```

Include dependency graph for Wind.h:



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



Classes

- struct [WindInputs](#)

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

- class [Wind](#)

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

Enumerations

- enum [WindPowerProductionModel](#) { [WIND_POWER_EXPONENTIAL](#) , [WIND_POWER_LOOKUP](#) , [N_WIND_POWER_PRODUCTION_MODELS](#) }

5.11.1 Detailed Description

Header file the [Wind](#) class.

5.11.2 Enumeration Type Documentation

5.11.2.1 WindPowerProductionModel

```
enum WindPowerProductionModel
```

Enumerator

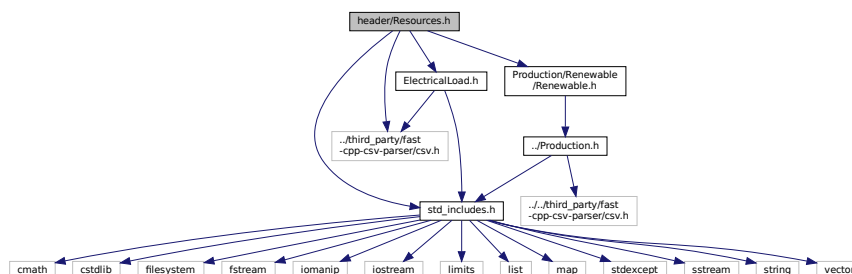
WIND_POWER_EXPONENTIAL	An exponential power production model.
WIND_POWER_LOOKUP	Lookup from a given set of power curve data.
N_WIND_POWER_PRODUCTION_MODELS	A simple hack to get the number of elements in <code>WindPowerProductionModel</code> .

```
34     {
35         WIND_POWER_EXPONENTIAL,
36         WIND_POWER_LOOKUP,
37         N_WIND_POWER_PRODUCTION_MODELS
38     };
```

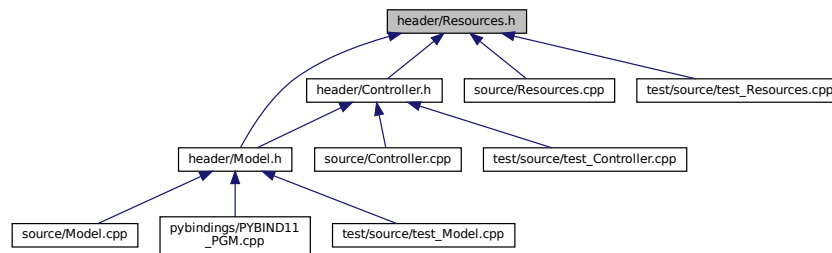
5.12 header/Resources.h File Reference

Header file the [Resources](#) class.

```
#include "std_includes.h"
#include "../third_party/fast-cpp-csv-parser/csv.h"
#include "ElectricalLoad.h"
#include "Production/Renewable/Renewable.h"
Include dependency graph for Resources.h:
```



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



Classes

- class [Resources](#)

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

5.12.1 Detailed Description

Header file the [Resources](#) class.

5.13 header/std_includes.h File Reference

Header file which simply batches together the usual, standard includes.

```

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

Include dependency graph for std_includes.h:



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



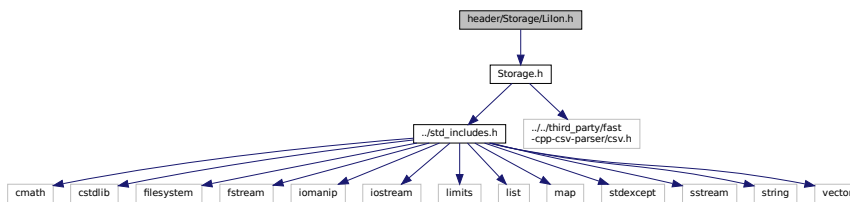
5.13.1 Detailed Description

Header file which simply batches together the usual, standard includes.

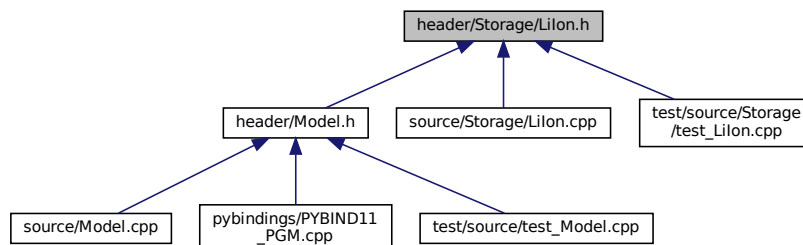
5.14 header/Storage/Lilon.h File Reference

Header file the [Lilon](#) class.

```
#include "Storage.h"
Include dependency graph for Lilon.h:
```



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



Classes

- class [Lilon](#)

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

5.14.1 Detailed Description

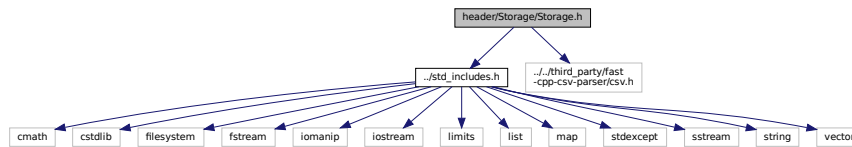
Header file the [Lilon](#) class.

5.15 header/Storage/Storage.h File Reference

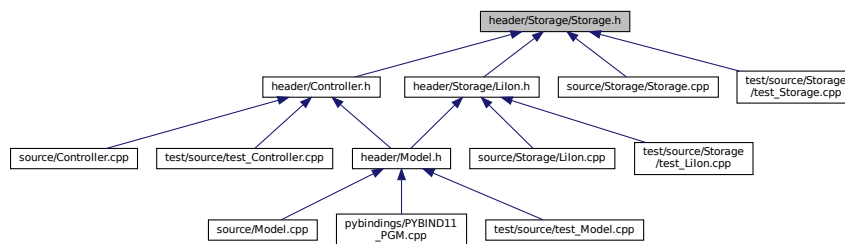
Header file the [Storage](#) class.

```
#include "../std_includes.h"
#include "../../third_party/fast-cpp-csv-parser/csv.h"
```

Include dependency graph for Storage.h:



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



Classes

- class [Storage](#)

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

5.15.1 Detailed Description

Header file the [Storage](#) class.

5.16 pybindings/PYBIND11_PGM.cpp File Reference

Python 3 bindings file for PGMcpp.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
```



```

46     .def_readwrite("mean_load_kW", &ElectricalLoad::mean_load_kW)
47     .def_readwrite("min_load_kW", &ElectricalLoad::min_load_kW)
48     .def_readwrite("dt_vec_hrs", &ElectricalLoad::dt_vec_hrs)
49     .def_readwrite("load_vec_kW", &ElectricalLoad::load_vec_kW)
50     .def_readwrite("time_vec_hrs", &ElectricalLoad::time_vec_hrs)
51
52     .def(pybind11::init<std::string>());
53 */
54 // ===== END ElectricalLoad ===== //
55
56
57
58 // ===== Model ===== //
59 /*
60 pybind11::class_<Model>(m, "Model")
61     .def(
62         pybind11::init<
63             ElectricalLoad*,
64             RenewableResources*
65         >()
66     );
67 */
68 // ===== END Model ===== //
69
70
71
72 // ===== RenewableResources ===== //
73 /*
74 pybind11::class_<RenewableResources>(m, "RenewableResources")
75     .def(pybind11::init());
76     /*
77     .def(pybind11::init<>());
78     */
79 */
80 // ===== END RenewableResources ===== //
81
82 } /* PYBIND11_MODULE() */

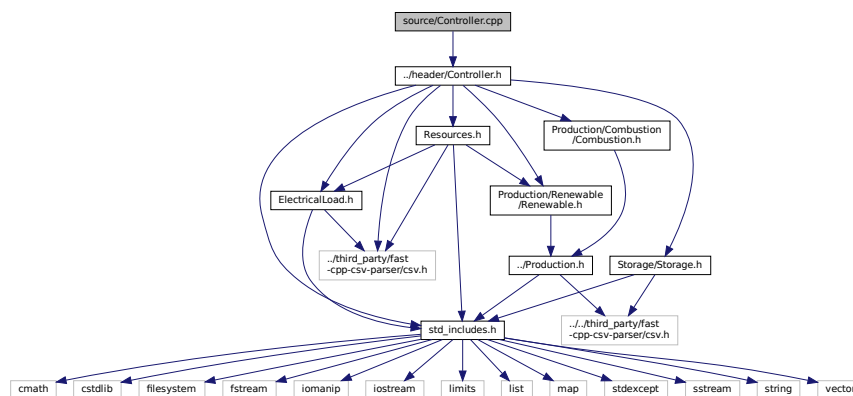
```

5.17 source/Controller.cpp File Reference

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

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

Include dependency graph for Controller.cpp:



5.17.1 Detailed Description

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

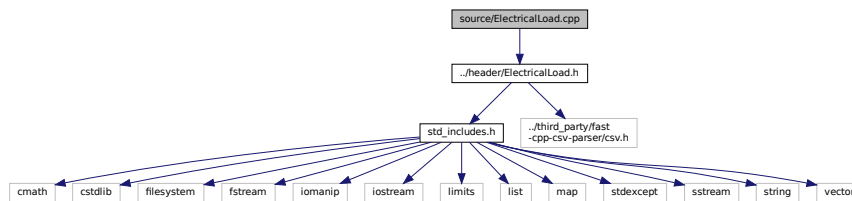
A class which contains a various dispatch control logic. Intended to serve as a component class of [Controller](#).

5.18 source/ElectricalLoad.cpp File Reference

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

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

Include dependency graph for ElectricalLoad.cpp:



5.18.1 Detailed Description

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

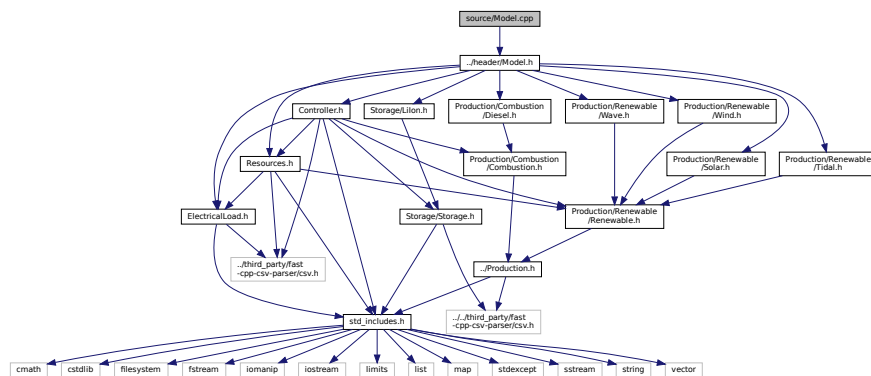
A class which contains time and electrical load data. Intended to serve as a component class of [Model](#).

5.19 source/Model.cpp File Reference

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

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

Include dependency graph for Model.cpp:



5.19.1 Detailed Description

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

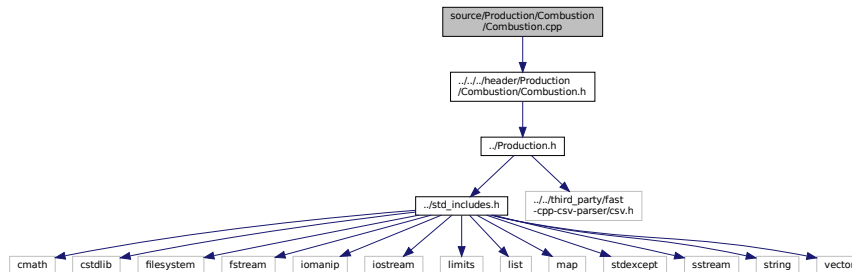
A container class which forms the centre of PGMcpp. The [Model](#) class is intended to serve as the primary user interface with the functionality of PGMcpp, and as such it contains all other classes.

5.20 source/Production/Combustion/Combustion.cpp File Reference

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

```
#include "../.../header/Production/Combustion/Combustion.h"
```

Include dependency graph for Combustion.cpp:



5.20.1 Detailed Description

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

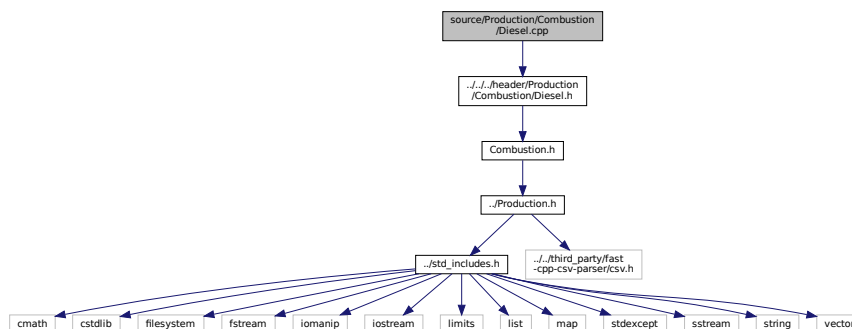
The root of the [Combustion](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the production of energy by way of combustibles.

5.21 source/Production/Combustion/Diesel.cpp File Reference

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

```
#include "../.../header/Production/Combustion/Diesel.h"
```

Include dependency graph for Diesel.cpp:



5.21.1 Detailed Description

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

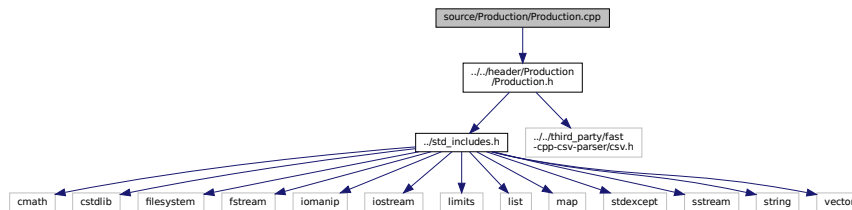
A derived class of the [Combustion](#) branch of [Production](#) which models production using a diesel generator.

5.22 source/Production/Production.cpp File Reference

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

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

Include dependency graph for Production.cpp:



5.22.1 Detailed Description

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

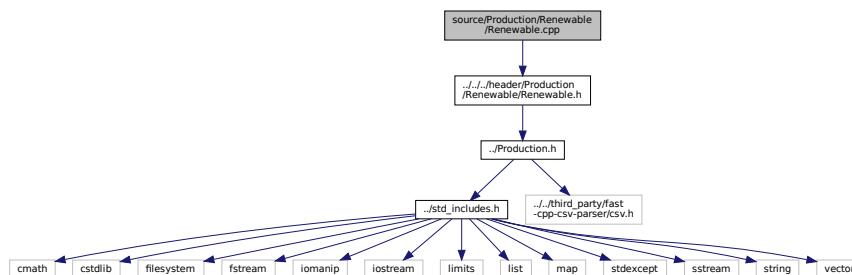
The base class of the [Production](#) hierarchy. This hierarchy contains derived classes which model the production of energy, be it renewable or otherwise.

5.23 source/Production/Renewable/Renewable.cpp File Reference

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

```
#include "../..//header/Production/Renewable/Renewable.h"
```

Include dependency graph for Renewable.cpp:



5.23.1 Detailed Description

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

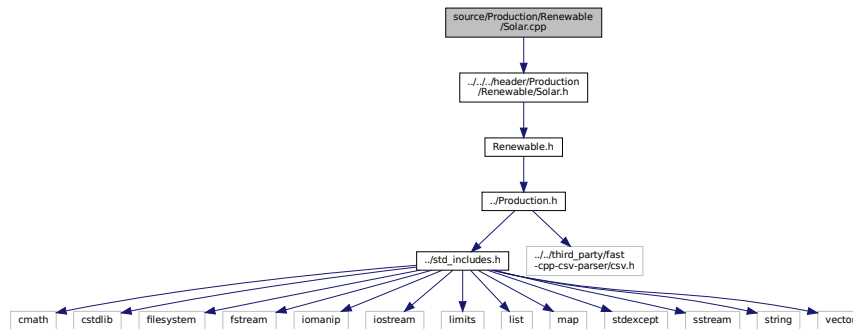
The root of the [Renewable](#) branch of the [Production](#) hierarchy. This branch contains derived classes which model the renewable production of energy.

5.24 source/Production/Renewable/Solar.cpp File Reference

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

```
#include "../.../header/Production/Renewable/Solar.h"
```

Include dependency graph for Solar.cpp:



5.24.1 Detailed Description

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

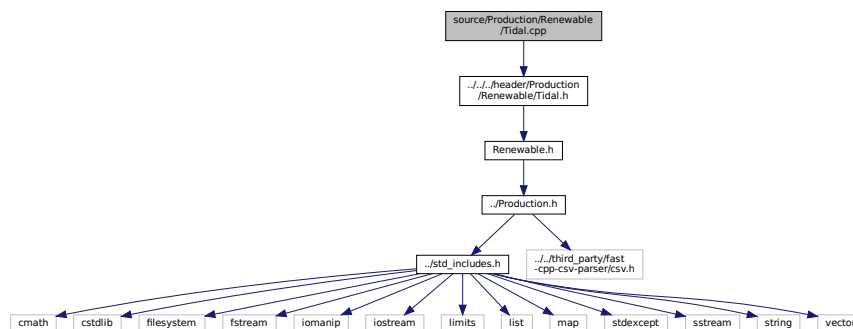
A derived class of the [Renewable](#) branch of [Production](#) which models solar production.

5.25 source/Production/Renewable/Tidal.cpp File Reference

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

```
#include "../.../header/Production/Renewable/Tidal.h"
```

Include dependency graph for Tidal.cpp:



5.25.1 Detailed Description

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

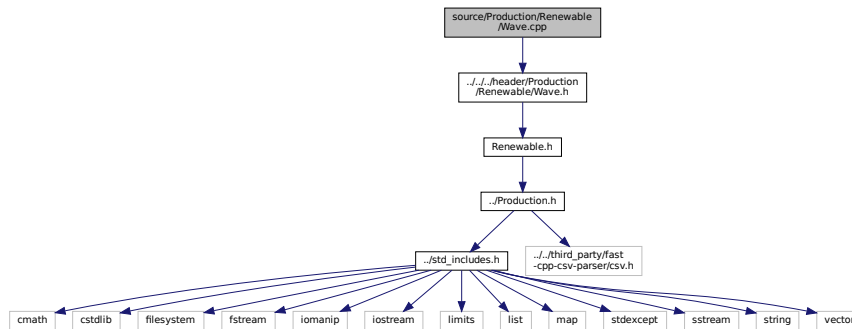
A derived class of the [Renewable](#) branch of [Production](#) which models tidal production.

5.26 source/Production/Renewable/Wave.cpp File Reference

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

```
#include "../.../header/Production/Renewable/Wave.h"
```

Include dependency graph for Wave.cpp:



5.26.1 Detailed Description

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

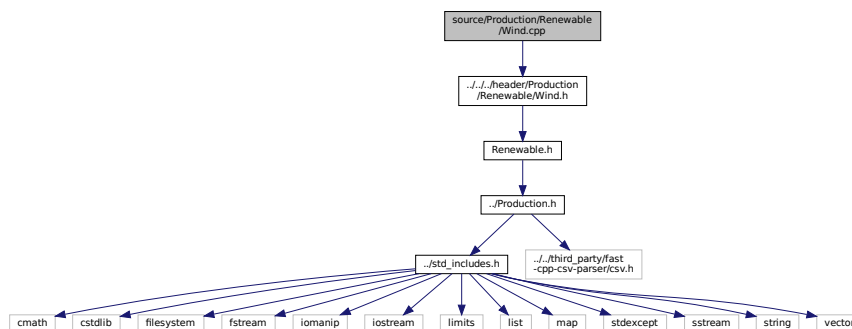
A derived class of the [Renewable](#) branch of [Production](#) which models wave production.

5.27 source/Production/Renewable/Wind.cpp File Reference

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

```
#include "../.../header/Production/Renewable/Wind.h"
```

Include dependency graph for Wind.cpp:



5.27.1 Detailed Description

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

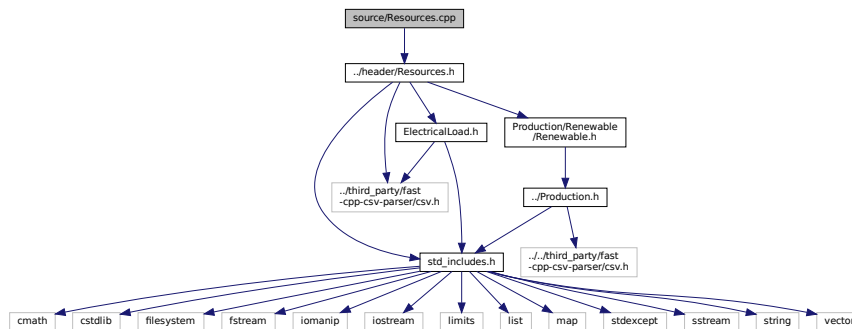
A derived class of the [Renewable](#) branch of [Production](#) which models wind production.

5.28 source/Resources.cpp File Reference

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

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

Include dependency graph for Resources.cpp:



5.28.1 Detailed Description

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

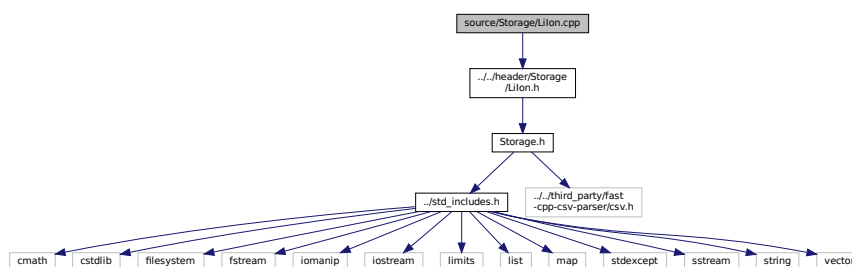
A class which contains renewable resource data. Intended to serve as a component class of [Model](#).

5.29 source/Storage/Lilon.cpp File Reference

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

```
#include "../header/Storage/LiIon.h"
```

Include dependency graph for Lilon.cpp:



5.29.1 Detailed Description

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

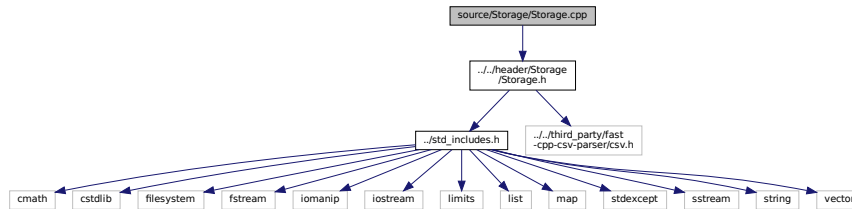
A derived class of [Storage](#) which models energy storage by way of lithium-ion batteries.

5.30 source/Storage/Storage.cpp File Reference

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

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

Include dependency graph for Storage.cpp:



5.30.1 Detailed Description

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

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

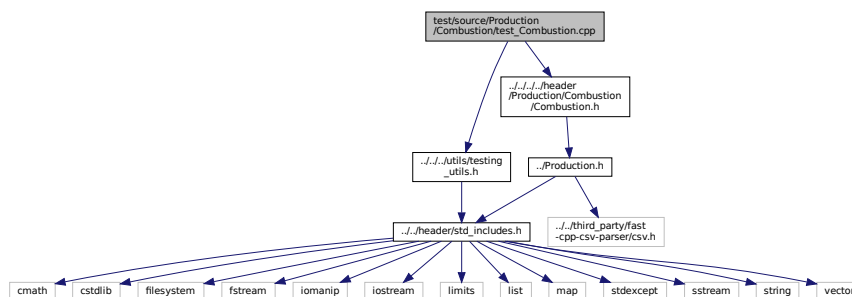
5.31 test/source/Production/Combustion/test_Combustion.cpp File Reference

Testing suite for [Combustion](#) class.

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

```
#include "../../../header/Production/Combustion/Combustion.h"
```

Include dependency graph for test_Combustion.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.31.1 Detailed Description

Testing suite for [Combustion](#) class.

A suite of tests for the [Combustion](#) class.

5.31.2 Function Documentation

5.31.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif /* _WIN32 */

    printGold("\tTesting Production <-- Combustion");
    srand(time(NULL));

    try {
        // ===== CONSTRUCTION ===== //
        CombustionInputs combustion_inputs;
        Combustion test_combustion(8760, combustion_inputs);
        // ===== END CONSTRUCTION ===== //

        // ===== ATTRIBUTES ===== //
        testTruth(
            not combustion_inputs.production_inputs.print_flag,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_combustion.fuel_consumption_vec_L.size(),
            8760,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_combustion.fuel_cost_vec.size(),
            8760,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_combustion.CO2_emissions_vec_kg.size(),
            8760,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_combustion.CO_emissions_vec_kg.size(),
            8760,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
```

```

86     test_combustion.NOx_emissions_vec_kg.size(),
87     8760,
88     __FILE__,
89     __LINE__
90 );
91
92 testFloatEquals(
93     test_combustion.SOx_emissions_vec_kg.size(),
94     8760,
95     __FILE__,
96     __LINE__
97 );
98
99 testFloatEquals(
100     test_combustion.CH4_emissions_vec_kg.size(),
101     8760,
102     __FILE__,
103     __LINE__
104 );
105
106 testFloatEquals(
107     test_combustion.PM_emissions_vec_kg.size(),
108     8760,
109     __FILE__,
110     __LINE__
111 );
112
113 // ===== END ATTRIBUTES ===== //
114
115 } /* try */
116
117
118 catch (...) {
119     //...
120
121     printGold(" ..... ");
122     printRed("FAIL");
123     std::cout << std::endl;
124     throw;
125 }
126
127
128 printGold(" ..... ");
129 printGreen("PASS");
130 std::cout << std::endl;
131 return 0;
132
133 } /* main() */

```

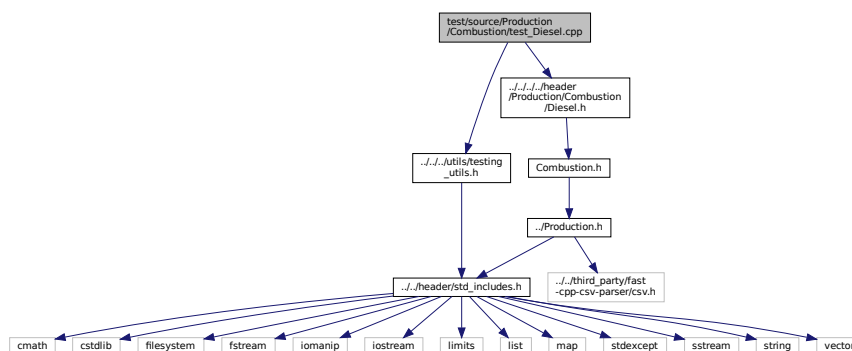
5.32 test/source/Production/Combustion/test_Diesel.cpp File Reference

Testing suite for [Diesel](#) class.

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

```
#include "../.../header/Production/Combustion/Diesel.h"
```

Include dependency graph for test_Diesel.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.32.1 Detailed Description

Testing suite for [Diesel](#) class.

A suite of tests for the [Diesel](#) class.

5.32.2 Function Documentation

5.32.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Combustion <-- Diesel");
33
34     srand(time(NULL));
35
36
37     Combustion* test_diesel_ptr;
38
39     try {
40
41         // ===== CONSTRUCTION ===== //
42
43         bool error_flag = true;
44
45         try {
46             DieselInputs bad_diesel_inputs;
47             bad_diesel_inputs.fuel_cost_L = -1;
48
49             Diesel bad_diesel(8760, bad_diesel_inputs);
50
51             error_flag = false;
52         } catch (...) {
53             // Task failed successfully! =P
54         }
55         if (not error_flag) {
56             expectedErrorNotDetected(__FILE__, __LINE__);
57         }
58
59         DieselInputs diesel_inputs;
60
61         test_diesel_ptr = new Diesel(8760, diesel_inputs);
62
63
64         // ===== END CONSTRUCTION ===== //
65
66
67
68         // ===== ATTRIBUTES ===== //
69
70         testTruth(
71             not diesel_inputs.combustion_inputs.production_inputs.print_flag,
72             __FILE__,
73             __LINE__
74         );
75
76         testFloatEquals(
77             test_diesel_ptr->type,
```

```

78     CombustionType :: DIESEL,
79     __FILE__,
80     __LINE__
81 );
82
83 testTruth(
84     test_diesel_ptr->type_str == "DIESEL",
85     __FILE__,
86     __LINE__
87 );
88
89 testFloatEquals(
90     test_diesel_ptr->linear_fuel_slope_LkWh,
91     0.265675,
92     __FILE__,
93     __LINE__
94 );
95
96 testFloatEquals(
97     test_diesel_ptr->linear_fuel_intercept_LkWh,
98     0.026676,
99     __FILE__,
100    __LINE__
101 );
102
103 testFloatEquals(
104     test_diesel_ptr->capital_cost,
105     94125.375446,
106     __FILE__,
107     __LINE__
108 );
109
110 testFloatEquals(
111     test_diesel_ptr->operation_maintenance_cost_kWh,
112     0.069905,
113     __FILE__,
114     __LINE__
115 );
116
117 testFloatEquals(
118     ((Diesel*)test_diesel_ptr)->minimum_load_ratio,
119     0.2,
120     __FILE__,
121     __LINE__
122 );
123
124 testFloatEquals(
125     ((Diesel*)test_diesel_ptr)->minimum_runtime_hrs,
126     4,
127     __FILE__,
128     __LINE__
129 );
130
131 testFloatEquals(
132     test_diesel_ptr->replace_running_hrs,
133     30000,
134     __FILE__,
135     __LINE__
136 );
137
138 // ===== END ATTRIBUTES ===== //
139
140
141
142 // ===== METHODS ===== //
143
144 // test capacity constraint
145 testFloatEquals(
146     test_diesel_ptr->requestProductionkW(0, 1, 2 * test_diesel_ptr->capacity_kW),
147     test_diesel_ptr->capacity_kW,
148     __FILE__,
149     __LINE__
150 );
151
152 // test minimum load ratio constraint
153 testFloatEquals(
154     test_diesel_ptr->requestProductionkW(
155         0,
156         1,
157         0.5 * ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
158             test_diesel_ptr->capacity_kW
159     ),
160     ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW,
161     __FILE__,
162     __LINE__
163 );
164

```



```

165 // test commit()
166 std::vector<double> dt_vec_hrs (48, 1);
167
168 std::vector<double> load_vec_kW = {
169     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
170     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
171     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
172     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
173 };
174
175 std::vector<bool> expected_is_running_vec = {
176     1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
177     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
178     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
179     1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0
180 };
181
182 double load_kW = 0;
183 double production_kW = 0;
184 double roll = 0;
185
186 for (int i = 0; i < 48; i++) {
187     roll = (double)rand() / RAND_MAX;
188
189     if (roll >= 0.95) {
190         roll = 1.25;
191     }
192
193     load_vec_kW[i] *= roll * test_diesel_ptr->capacity_kW;
194     load_kW = load_vec_kW[i];
195
196     production_kW = test_diesel_ptr->requestProductionkW(
197         i,
198         dt_vec_hrs[i],
199         load_kW
200     );
201
202     load_kW = test_diesel_ptr->commit(
203         i,
204         dt_vec_hrs[i],
205         production_kW,
206         load_kW
207     );
208
209     // load_kW <= load_vec_kW (i.e., after vs before)
210     testLessThanOrEqualTo(
211         load_kW,
212         load_vec_kW[i],
213         __FILE__,
214         __LINE__
215     );
216
217     // production = dispatch + storage + curtailment
218     testFloatEquals(
219         test_diesel_ptr->production_vec_kW[i] -
220         test_diesel_ptr->dispatch_vec_kW[i] -
221         test_diesel_ptr->storage_vec_kW[i] -
222         test_diesel_ptr->curtailment_vec_kW[i],
223         0,
224         __FILE__,
225         __LINE__
226     );
227
228     // capacity constraint
229     if (load_vec_kW[i] > test_diesel_ptr->capacity_kW) {
230         testFloatEquals(
231             test_diesel_ptr->production_vec_kW[i],
232             test_diesel_ptr->capacity_kW,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // minimum load ratio constraint
239     else if (
240         test_diesel_ptr->is_running and
241         test_diesel_ptr->production_vec_kW[i] > 0 and
242         load_vec_kW[i] <
243         ((Diesel*)test_diesel_ptr)->minimum_load_ratio * test_diesel_ptr->capacity_kW
244     ) {
245         testFloatEquals(
246             test_diesel_ptr->production_vec_kW[i],
247             ((Diesel*)test_diesel_ptr)->minimum_load_ratio *
248             test_diesel_ptr->capacity_kW,
249             __FILE__,
250             __LINE__
251         );

```

```

252     }
253
254     // minimum runtime constraint
255     testFloatEquals(
256         test_diesel_ptr->is_running_vec[i],
257         expected_is_running_vec[i],
258         __FILE__,
259         __LINE__
260     );
261
262     // O&M, fuel consumption, and emissions > 0 whenever diesel is running
263     if (test_diesel_ptr->is_running) {
264         testGreaterThan(
265             test_diesel_ptr->operation_maintenance_cost_vec[i],
266             0,
267             __FILE__,
268             __LINE__
269         );
270
271         testGreaterThan(
272             test_diesel_ptr->fuel_consumption_vec_L[i],
273             0,
274             __FILE__,
275             __LINE__
276         );
277
278         testGreaterThan(
279             test_diesel_ptr->fuel_cost_vec[i],
280             0,
281             __FILE__,
282             __LINE__
283         );
284
285         testGreaterThan(
286             test_diesel_ptr->CO2_emissions_vec_kg[i],
287             0,
288             __FILE__,
289             __LINE__
290         );
291
292         testGreaterThan(
293             test_diesel_ptr->CO_emissions_vec_kg[i],
294             0,
295             __FILE__,
296             __LINE__
297         );
298
299         testGreaterThan(
300             test_diesel_ptr->NOx_emissions_vec_kg[i],
301             0,
302             __FILE__,
303             __LINE__
304         );
305
306         testGreaterThan(
307             test_diesel_ptr->SOx_emissions_vec_kg[i],
308             0,
309             __FILE__,
310             __LINE__
311         );
312
313         testGreaterThan(
314             test_diesel_ptr->CH4_emissions_vec_kg[i],
315             0,
316             __FILE__,
317             __LINE__
318         );
319
320         testGreaterThan(
321             test_diesel_ptr->PM_emissions_vec_kg[i],
322             0,
323             __FILE__,
324             __LINE__
325         );
326     }
327
328     // O&M, fuel consumption, and emissions = 0 whenever diesel is not running
329     else {
330         testFloatEquals(
331             test_diesel_ptr->operation_maintenance_cost_vec[i],
332             0,
333             __FILE__,
334             __LINE__
335         );
336
337         testFloatEquals(
338             test_diesel_ptr->fuel_consumption_vec_L[i],

```

```

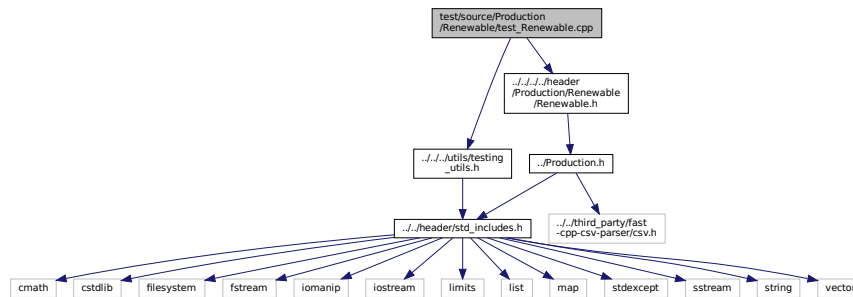
339         0,
340         __FILE__,
341         __LINE__
342     );
343
344     testFloatEquals(
345         test_diesel_ptr->fuel_cost_vec[i],
346         0,
347         __FILE__,
348         __LINE__
349     );
350
351     testFloatEquals(
352         test_diesel_ptr->CO2_emissions_vec_kg[i],
353         0,
354         __FILE__,
355         __LINE__
356     );
357
358     testFloatEquals(
359         test_diesel_ptr->CO_emissions_vec_kg[i],
360         0,
361         __FILE__,
362         __LINE__
363     );
364
365     testFloatEquals(
366         test_diesel_ptr->NOx_emissions_vec_kg[i],
367         0,
368         __FILE__,
369         __LINE__
370     );
371
372     testFloatEquals(
373         test_diesel_ptr->SOx_emissions_vec_kg[i],
374         0,
375         __FILE__,
376         __LINE__
377     );
378
379     testFloatEquals(
380         test_diesel_ptr->CH4_emissions_vec_kg[i],
381         0,
382         __FILE__,
383         __LINE__
384     );
385
386     testFloatEquals(
387         test_diesel_ptr->PM_emissions_vec_kg[i],
388         0,
389         __FILE__,
390         __LINE__
391     );
392 }
393 }
394
395 // ===== END METHODS ===== //
396
397 } /* try */
398
399
400 catch (...) {
401     delete test_diesel_ptr;
402
403     printGold(" ... ");
404     printRed("FAIL");
405     std::cout << std::endl;
406     throw;
407 }
408
409
410 delete test_diesel_ptr;
411
412 printGold(" ... ");
413 printGreen("PASS");
414 std::cout << std::endl;
415 return 0;
416
417 } /* main() */

```

5.33 test/source/Production/Renewable/test_Renewable.cpp File Reference

Testing suite for [Renewable](#) class.

```
#include "../.../utils/testing_utils.h"
#include "../.../header/Production/Renewable/Renewable.h"
Include dependency graph for test_Renewable.cpp:
```



Functions

- int [main](#) (int argc, char **argv)

5.33.1 Detailed Description

Testing suite for [Renewable](#) class.

A suite of tests for the [Renewable](#) class.

5.33.2 Function Documentation

5.33.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Production <-- Renewable");
    33
    34     srand(time(NULL));
    35
    36
    37 try {
    38
    39 // ===== CONSTRUCTION ===== //
```

```

40
41 RenewableInputs renewable_inputs;
42
43 Renewable test_renewable(8760, renewable_inputs);
44
45 // ===== END CONSTRUCTION ===== //
46
47
48
49 // ===== ATTRIBUTES ===== //
50
51 testTruth(
52     not renewable_inputs.production_inputs.print_flag,
53     __FILE__,
54     __LINE__
55 );
56
57 // ===== END ATTRIBUTES ===== //
58
59 } /* try */
60
61
62 catch (...) {
63     //...
64
65     printGold(" ..... ");
66     printRed("FAIL");
67     std::cout << std::endl;
68     throw;
69 }
70
71
72 printGold(" ..... ");
73 printGreen("PASS");
74 std::cout << std::endl;
75 return 0;
76 } /* main() */

```

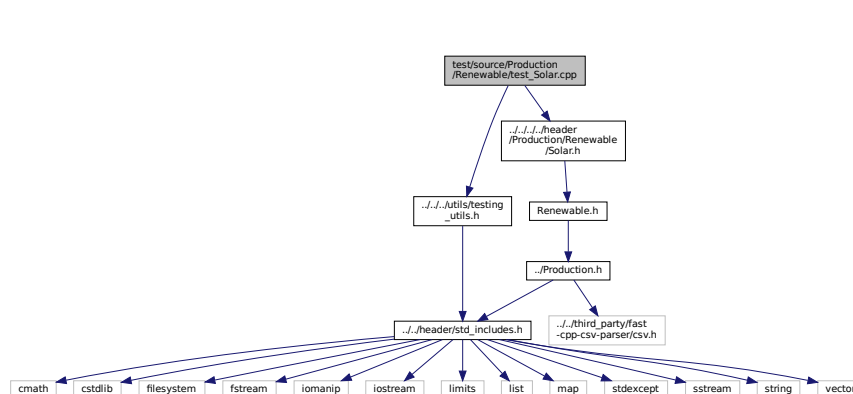
5.34 test/source/Production/Renewable/test_Solar.cpp File Reference

Testing suite for [Solar](#) class.

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

```
#include "../.../header/Production/Renewable/Solar.h"
```

Include dependency graph for test_Solar.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.34.1 Detailed Description

Testing suite for [Solar](#) class.

A suite of tests for the [Solar](#) class.

5.34.2 Function Documentation

5.34.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Solar");
    srand(time(NULL));

    Renewable* test_solar_ptr;

    try {
        // ===== CONSTRUCTION ===== //
        bool error_flag = true;
        try {
            SolarInputs bad_solar_inputs;
            bad_solar_inputs.derating = -1;
            Solar bad_solar(8760, bad_solar_inputs);
            error_flag = false;
        } catch (...) {
            // Task failed successfully! =P
        }
        if (not error_flag) {
            expectedErrorNotDetected(__FILE__, __LINE__);
        }
        SolarInputs solar_inputs;
        test_solar_ptr = new Solar(8760, solar_inputs);
        // ===== END CONSTRUCTION ===== //

        // ===== ATTRIBUTES ===== //
        testTruth(
            not solar_inputs.renewable_inputs.production_inputs.print_flag,
            __FILE__,
            __LINE__
        );
        testFloatEquals(
            test_solar_ptr->type,
            RenewableType :: SOLAR,
            __FILE__,
            __LINE__
        );
        testTruth(
            test_solar_ptr->type_str == "SOLAR",
            __FILE__,
            __LINE__
        );
    }
```

```

86
87 testFloatEquals(
88     test_solar_ptr->capital_cost,
89     350118.723363,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_solar_ptr->operation_maintenance_cost_kWh,
96     0.01,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_solar_ptr->computeProductionkW(0, 1, 2),
110     100,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_solar_ptr->computeProductionkW(0, 1, -1),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;
134 double roll = 0;
135 double solar_resource_kWm2 = 0;
136
137 for (int i = 0; i < 48; i++) {
138     roll = (double)rand() / RAND_MAX;
139
140     solar_resource_kWm2 = roll;
141
142     roll = (double)rand() / RAND_MAX;
143
144     if (roll <= 0.1) {
145         solar_resource_kWm2 = 0;
146     }
147
148     else if (roll >= 0.95) {
149         solar_resource_kWm2 = 1.25;
150     }
151
152     roll = (double)rand() / RAND_MAX;
153
154     if (roll >= 0.95) {
155         roll = 1.25;
156     }
157
158     load_vec_kW[i] *= roll * test_solar_ptr->capacity_kW;
159     load_kW = load_vec_kW[i];
160
161     production_kW = test_solar_ptr->computeProductionkW(
162         i,
163         dt_vec_hrs[i],
164         solar_resource_kWm2
165     );
166
167     load_kW = test_solar_ptr->commit(
168         i,
169         dt_vec_hrs[i],
170         production_kW,
171         load_kW
172     );

```

```

173
174 // is running (or not) as expected
175 if (solar_resource_kWm2 > 0) {
176     testTruth(
177         test_solar_ptr->is_running,
178         __FILE__,
179         __LINE__
180     );
181 }
182
183 else {
184     testTruth(
185         not test_solar_ptr->is_running,
186         __FILE__,
187         __LINE__
188     );
189 }
190
191 // load_kW <= load_vec_kW (i.e., after vs before)
192 testLessThanOrEqualTo(
193     load_kW,
194     load_vec_kW[i],
195     __FILE__,
196     __LINE__
197 );
198
199 // production = dispatch + storage + curtailment
200 testFloatEquals(
201     test_solar_ptr->production_vec_kW[i] -
202     test_solar_ptr->dispatch_vec_kW[i] -
203     test_solar_ptr->storage_vec_kW[i] -
204     test_solar_ptr->curtailment_vec_kW[i],
205     0,
206     __FILE__,
207     __LINE__
208 );
209
210 // capacity constraint
211 if (solar_resource_kWm2 > 1) {
212     testFloatEquals(
213         test_solar_ptr->production_vec_kW[i],
214         test_solar_ptr->capacity_kW,
215         __FILE__,
216         __LINE__
217     );
218 }
219
220 // resource, O&M > 0 whenever solar is running (i.e., producing)
221 if (test_solar_ptr->is_running) {
222     testGreaterThan(
223         solar_resource_kWm2,
224         0,
225         __FILE__,
226         __LINE__
227     );
228
229     testGreaterThan(
230         test_solar_ptr->operation_maintenance_cost_vec[i],
231         0,
232         __FILE__,
233         __LINE__
234     );
235 }
236
237 // resource, O&M = 0 whenever solar is not running (i.e., not producing)
238 else {
239     testFloatEquals(
240         solar_resource_kWm2,
241         0,
242         __FILE__,
243         __LINE__
244     );
245
246     testFloatEquals(
247         test_solar_ptr->operation_maintenance_cost_vec[i],
248         0,
249         __FILE__,
250         __LINE__
251     );
252 }
253 }
254
255
256 // ===== END METHODS ===== //
257
258 } /* try */
259

```



```

260
261 catch (...) {
262     delete test_solar_ptr;
263
264     printGold(" ..... ");
265     printRed("FAIL");
266     std::cout << std::endl;
267     throw;
268 }
269
270
271 delete test_solar_ptr;
272
273 printGold(" ..... ");
274 printGreen("PASS");
275 std::cout << std::endl;
276 return 0;
277 } /* main() */

```

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

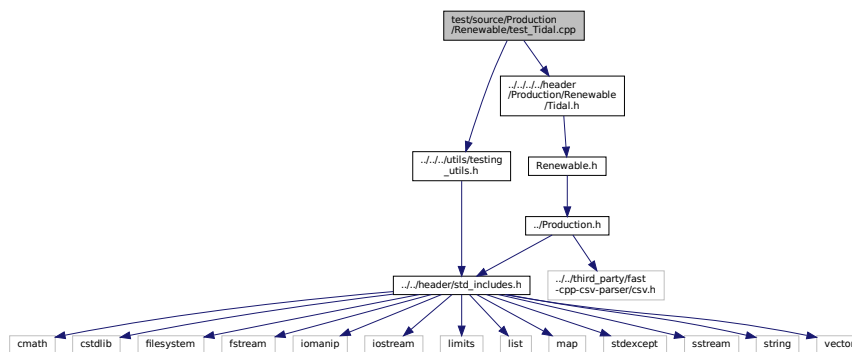
Testing suite for [Tidal](#) class.

```

#include "../.../utils/testing_utils.h"
#include "../.../header/Production/Renewable/Tidal.h"

```

Include dependency graph for test_Tidal.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.35.1 Detailed Description

Testing suite for [Tidal](#) class.

A suite of tests for the [Tidal](#) class.

5.35.2 Function Documentation

5.35.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Tidal");
33
34     srand(time(NULL));
35
36     Renewable* test_tidal_ptr;
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         bool error_flag = true;
43
44         try {
45             TidalInputs bad_tidal_inputs;
46             bad_tidal_inputs.design_speed_ms = -1;
47
48             Tidal bad_tidal(8760, bad_tidal_inputs);
49
50             error_flag = false;
51         } catch (...) {
52             // Task failed successfully! =P
53         }
54         if (not error_flag) {
55             expectedErrorNotDetected(__FILE__, __LINE__);
56         }
57
58         TidalInputs tidal_inputs;
59
60         test_tidal_ptr = new Tidal(8760, tidal_inputs);
61
62         // ===== END CONSTRUCTION ===== //
63
64
65
66         // ===== ATTRIBUTES ===== //
67
68         testTruth(
69             not tidal_inputs.renewable_inputs.production_inputs.print_flag,
70             __FILE__,
71             __LINE__
72 );
73
74         testFloatEquals(
75             test_tidal_ptr->type,
76             RenewableType :: TIDAL,
77             __FILE__,
78             __LINE__
79 );
80
81         testTruth(
82             test_tidal_ptr->type_str == "TIDAL",
83             __FILE__,
84             __LINE__
85 );
86
87         testFloatEquals(
88             test_tidal_ptr->capital_cost,
89             500237.446725,
90             __FILE__,
91             __LINE__
92 );
93
94         testFloatEquals(
95             test_tidal_ptr->operation_maintenance_cost_kWh,
96             0.069905,
97             __FILE__,
98             __LINE__
99 );
100
101         // ===== END ATTRIBUTES ===== //
102
103
104
105         // ===== METHODS ===== //
106

```

```

107 // test production constraints
108 testFloatEquals(
109     test_tidal_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_tidal_ptr->computeProductionkW(
117         0,
118         1,
119         ((Tidal*)test_tidal_ptr)->design_speed_ms
120     ),
121     test_tidal_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_tidal_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()
134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double tidal_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     tidal_resource_ms = roll * ((Tidal*)test_tidal_ptr)->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         tidal_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         tidal_resource_ms = 3 * ((Tidal*)test_tidal_ptr)->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164
165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_tidal_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_tidal_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         tidal_resource_ms
176     );
177
178     load_kW = test_tidal_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_tidal_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193 }

```

```

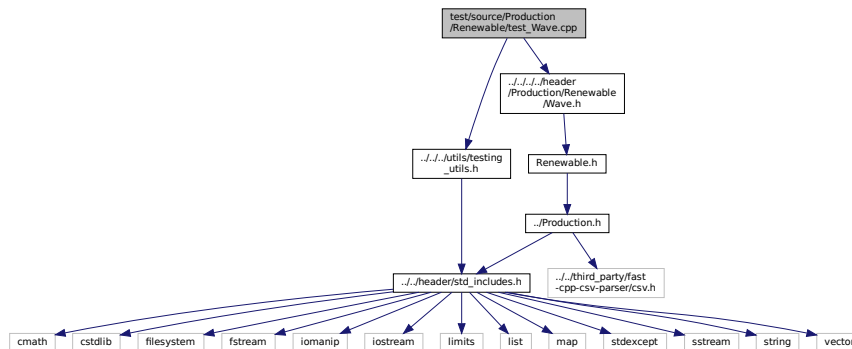
194     else {
195         testTruth(
196             not test_tidal_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_tidal_ptr->production_vec_kW[i] -
213         test_tidal_ptr->dispatch_vec_kW[i] -
214         test_tidal_ptr->storage_vec_kW[i] -
215         test_tidal_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220
221     // resource, O&M > 0 whenever tidal is running (i.e., producing)
222     if (test_tidal_ptr->is_running) {
223         testGreaterThan(
224             tidal_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229
230         testGreaterThan(
231             test_tidal_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // O&M = 0 whenever tidal is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_tidal_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248
249
250 // ===== END METHODS ===== //
251
252 } /* try */
253
254
255 catch (...) {
256     delete test_tidal_ptr;
257
258     printGold(" ..... ");
259     printRed("FAIL");
260     std::cout << std::endl;
261     throw;
262 }
263
264
265 delete test_tidal_ptr;
266
267 printGold(" ..... ");
268 printGreen("PASS");
269 std::cout << std::endl;
270 return 0;
271 } /* main() */

```

5.36 test/source/Production/Renewable/test_Wave.cpp File Reference

Testing suite for [Wave](#) class.

```
#include "../../../utils/testing_utils.h"
#include "../../../header/Production/Renewable/Wave.h"
Include dependency graph for test_Wave.cpp:
```



Functions

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

5.36.1 Detailed Description

Testing suite for [Wave](#) class.

A suite of tests for the [Wave](#) class.

5.36.2 Function Documentation

5.36.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    #ifdef _WIN32
        activateVirtualTerminal();
    #endif /* _WIN32 */

    printGold("\tTesting Production <-- Renewable <-- Wave");

    srand(time(NULL));

    Renewable* test_wave_ptr;

    try {
        // ===== CONSTRUCTION =====
        bool error_flag = true;

        try {
            WaveInputs bad_wave_inputs;
            bad_wave_inputs.design_significant_wave_height_m = -1;
        }
    }
}
```

```

47
48     Wave bad_wave(8760, bad_wave_inputs);
49
50     error_flag = false;
51 } catch (...) {
52     // Task failed successfully! =P
53 }
54 if (not error_flag) {
55     expectedErrorNotDetected(__FILE__, __LINE__);
56 }
57
58 WaveInputs wave_inputs;
59
60 test_wave_ptr = new Wave(8760, wave_inputs);
61
62 // ===== END CONSTRUCTION ===== //
63
64
65
66 // ===== ATTRIBUTES ===== //
67
68 testTruth(
69     not wave_inputs.renewable_inputs.production_inputs.print_flag,
70     __FILE__,
71     __LINE__
72 );
73
74 testFloatEquals(
75     test_wave_ptr->type,
76     RenewableType :: WAVE,
77     __FILE__,
78     __LINE__
79 );
80
81 testTruth(
82     test_wave_ptr->type_str == "WAVE",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_wave_ptr->capital_cost,
89     850831.063539,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wave_ptr->operation_maintenance_cost_kWh,
96     0.069905,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wave_ptr->computeProductionkW(0, 1, 0, rand()),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wave_ptr->computeProductionkW(0, 1, rand(), 0),
117     0,
118     __FILE__,
119     __LINE__
120 );
121
122 // test commit()
123 std::vector<double> dt_vec_hrs (48, 1);
124
125 std::vector<double> load_vec_kW = {
126     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
127     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
128     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
129     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
130 };
131
132 double load_kW = 0;
133 double production_kW = 0;

```

```

134 double roll = 0;
135 double significant_wave_height_m = 0;
136 double energy_period_s = 0;
137
138 for (int i = 0; i < 48; i++) {
139     roll = (double)rand() / RAND_MAX;
140
141     if (roll <= 0.05) {
142         roll = 0;
143     }
144
145     significant_wave_height_m = roll *
146         ((Wave*)test_wave_ptr)->design_significant_wave_height_m;
147
148     roll = (double)rand() / RAND_MAX;
149
150     if (roll <= 0.05) {
151         roll = 0;
152     }
153
154     energy_period_s = roll * ((Wave*)test_wave_ptr)->design_energy_period_s;
155
156     roll = (double)rand() / RAND_MAX;
157
158     if (roll >= 0.95) {
159         roll = 1.25;
160     }
161
162     load_vec_kW[i] *= roll * test_wave_ptr->capacity_kW;
163     load_kW = load_vec_kW[i];
164
165     production_kW = test_wave_ptr->computeProductionkW(
166         i,
167         dt_vec_hrs[i],
168         significant_wave_height_m,
169         energy_period_s
170     );
171
172     load_kW = test_wave_ptr->commit(
173         i,
174         dt_vec_hrs[i],
175         production_kW,
176         load_kW
177     );
178
179     // is running (or not) as expected
180     if (production_kW > 0) {
181         testTruth(
182             test_wave_ptr->is_running,
183             __FILE__,
184             __LINE__
185         );
186     }
187
188     else {
189         testTruth(
190             not test_wave_ptr->is_running,
191             __FILE__,
192             __LINE__
193         );
194     }
195
196     // load_kW <= load_vec_kW (i.e., after vs before)
197     testLessThanOrEqualTo(
198         load_kW,
199         load_vec_kW[i],
200         __FILE__,
201         __LINE__
202     );
203
204     // production = dispatch + storage + curtailment
205     testFloatEquals(
206         test_wave_ptr->production_vec_kW[i] -
207         test_wave_ptr->dispatch_vec_kW[i] -
208         test_wave_ptr->storage_vec_kW[i] -
209         test_wave_ptr->curtailment_vec_kW[i],
210         0,
211         __FILE__,
212         __LINE__
213     );
214
215     // resource, O&M > 0 whenever wave is running (i.e., producing)
216     if (test_wave_ptr->is_running) {
217         testGreaterThan(
218             significant_wave_height_m,
219             0,
220             __FILE__,

```

```

221         __LINE__
222     );
223
224     testGreaterThan(
225         energy_period_s,
226         0,
227         __FILE__,
228         __LINE__
229     );
230
231     testGreaterThan(
232         test_wave_ptr->operation_maintenance_cost_vec[i],
233         0,
234         __FILE__,
235         __LINE__
236     );
237 }
238
239 // O&M = 0 whenever wave is not running (i.e., not producing)
240 else {
241     testFloatEquals(
242         test_wave_ptr->operation_maintenance_cost_vec[i],
243         0,
244         __FILE__,
245         __LINE__
246     );
247 }
248 }
249 // ===== END METHODS ===== //
250
251 } /* try */
252
253
254 catch (...) {
255     delete test_wave_ptr;
256
257     printGold(" ..... ");
258     printRed("FAIL");
259     std::cout << std::endl;
260     throw;
261 }
262
263 delete test_wave_ptr;
264
265 printGold(" ..... ");
266 printGreen("PASS");
267 std::cout << std::endl;
268 return 0;
269 } /* main() */

```

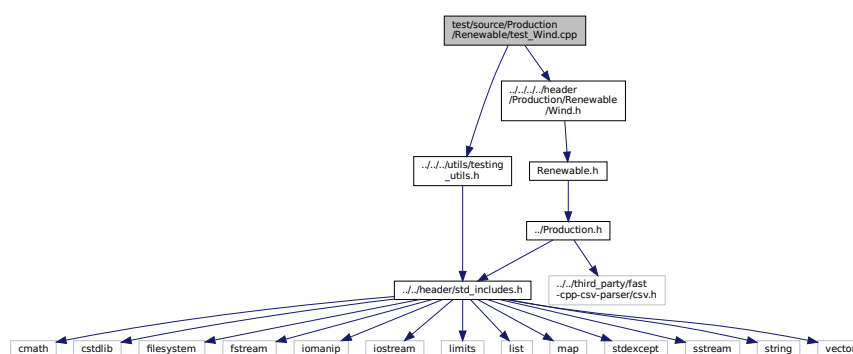
5.37 test/source/Production/Renewable/test_Wind.cpp File Reference

Testing suite for [Wind](#) class.

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

```
#include "../header/Production/Renewable/Wind.h"
```

Include dependency graph for test_Wind.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.37.1 Detailed Description

Testing suite for [Wind](#) class.

A suite of tests for the [Wind](#) class.

5.37.2 Function Documentation

5.37.2.1 main()

```
int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Production <-- Renewable <-- Wind");
33
34     srand(time(NULL));
35
36     Renewable* test_wind_ptr;
37
38     try {
39
40 // ===== CONSTRUCTION ===== //
41
42 bool error_flag = true;
43
44     try {
45         WindInputs bad_wind_inputs;
46         bad_wind_inputs.design_speed_ms = -1;
47
48         Wind bad_wind(8760, bad_wind_inputs);
49
50         error_flag = false;
51     } catch (...) {
52         // Task failed successfully! =P
53     }
54     if (not error_flag) {
55         expectedErrorNotDetected(__FILE__, __LINE__);
56     }
57
58     WindInputs wind_inputs;
59
60     test_wind_ptr = new Wind(8760, wind_inputs);
61
62 // ===== END CONSTRUCTION ===== //
63
64
65
66 // ===== ATTRIBUTES ===== //
67
68     testTruth(
69         not wind_inputs.renewable_inputs.production_inputs.print_flag,
70         __FILE__,
71         __LINE__
72 );
73
74     testFloatEquals(
75         test_wind_ptr->type,
76         RenewableType :: WIND,
77         __FILE__,
```

```

78     __LINE__
79 );
80
81 testTruth(
82     test_wind_ptr->type_str == "WIND",
83     __FILE__,
84     __LINE__
85 );
86
87 testFloatEquals(
88     test_wind_ptr->capital_cost,
89     450356.170088,
90     __FILE__,
91     __LINE__
92 );
93
94 testFloatEquals(
95     test_wind_ptr->operation_maintenance_cost_kWh,
96     0.034953,
97     __FILE__,
98     __LINE__
99 );
100
101 // ===== END ATTRIBUTES ===== //
102
103
104
105 // ===== METHODS ===== //
106
107 // test production constraints
108 testFloatEquals(
109     test_wind_ptr->computeProductionkW(0, 1, 1e6),
110     0,
111     __FILE__,
112     __LINE__
113 );
114
115 testFloatEquals(
116     test_wind_ptr->computeProductionkW(
117         0,
118         1,
119         ((Wind*)test_wind_ptr)->design_speed_ms
120     ),
121     test_wind_ptr->capacity_kW,
122     __FILE__,
123     __LINE__
124 );
125
126 testFloatEquals(
127     test_wind_ptr->computeProductionkW(0, 1, -1),
128     0,
129     __FILE__,
130     __LINE__
131 );
132
133 // test commit()
134 std::vector<double> dt_vec_hrs (48, 1);
135
136 std::vector<double> load_vec_kW = {
137     1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1,
138     1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0,
139     1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
140     1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0
141 };
142
143 double load_kW = 0;
144 double production_kW = 0;
145 double roll = 0;
146 double wind_resource_ms = 0;
147
148 for (int i = 0; i < 48; i++) {
149     roll = (double)rand() / RAND_MAX;
150
151     wind_resource_ms = roll * ((Wind*)test_wind_ptr)->design_speed_ms;
152
153     roll = (double)rand() / RAND_MAX;
154
155     if (roll <= 0.1) {
156         wind_resource_ms = 0;
157     }
158
159     else if (roll >= 0.95) {
160         wind_resource_ms = 3 * ((Wind*)test_wind_ptr)->design_speed_ms;
161     }
162
163     roll = (double)rand() / RAND_MAX;
164

```

```

165     if (roll >= 0.95) {
166         roll = 1.25;
167     }
168
169     load_vec_kW[i] *= roll * test_wind_ptr->capacity_kW;
170     load_kW = load_vec_kW[i];
171
172     production_kW = test_wind_ptr->computeProductionkW(
173         i,
174         dt_vec_hrs[i],
175         wind_resource_ms
176     );
177
178     load_kW = test_wind_ptr->commit(
179         i,
180         dt_vec_hrs[i],
181         production_kW,
182         load_kW
183     );
184
185     // is running (or not) as expected
186     if (production_kW > 0) {
187         testTruth(
188             test_wind_ptr->is_running,
189             __FILE__,
190             __LINE__
191         );
192     }
193
194     else {
195         testTruth(
196             not test_wind_ptr->is_running,
197             __FILE__,
198             __LINE__
199         );
200     }
201
202     // load_kW <= load_vec_kW (i.e., after vs before)
203     testLessThanOrEqualTo(
204         load_kW,
205         load_vec_kW[i],
206         __FILE__,
207         __LINE__
208     );
209
210     // production = dispatch + storage + curtailment
211     testFloatEquals(
212         test_wind_ptr->production_vec_kW[i] -
213         test_wind_ptr->dispatch_vec_kW[i] -
214         test_wind_ptr->storage_vec_kW[i] -
215         test_wind_ptr->curtailment_vec_kW[i],
216         0,
217         __FILE__,
218         __LINE__
219     );
220
221     // resource, O&M > 0 whenever wind is running (i.e., producing)
222     if (test_wind_ptr->is_running) {
223         testGreaterThan(
224             wind_resource_ms,
225             0,
226             __FILE__,
227             __LINE__
228         );
229
230         testGreaterThan(
231             test_wind_ptr->operation_maintenance_cost_vec[i],
232             0,
233             __FILE__,
234             __LINE__
235         );
236     }
237
238     // O&M = 0 whenever wind is not running (i.e., not producing)
239     else {
240         testFloatEquals(
241             test_wind_ptr->operation_maintenance_cost_vec[i],
242             0,
243             __FILE__,
244             __LINE__
245         );
246     }
247 }
248
249
250 // ===== END METHODS ===== //
251

```

```

252 }    /* try */
253
254
255 catch (...) {
256     delete test_wind_ptr;
257
258     printGold(" ..... ");
259     printRed("FAIL");
260     std::cout << std::endl;
261     throw;
262 }
263
264
265 delete test_wind_ptr;
266
267 printGold(" ..... ");
268 printGreen("PASS");
269 std::cout << std::endl;
270 return 0;
271 }    /* main() */

```

5.38 test/source/Production/test_Production.cpp File Reference

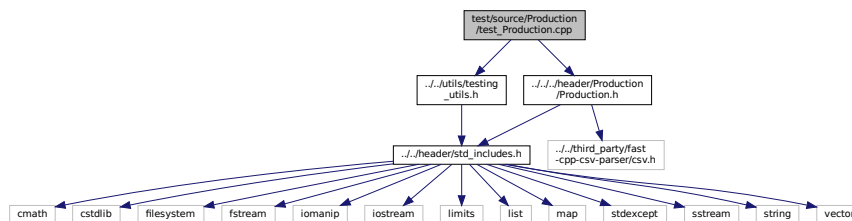
Testing suite for [Production](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Production/Production.h"

```

Include dependency graph for test_Production.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.38.1 Detailed Description

Testing suite for [Production](#) class.

A suite of tests for the [Production](#) class.

5.38.2 Function Documentation

5.38.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\n\tTesting Production");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         bool error_flag = true;
42
43         try {
44             ProductionInputs production_inputs;
45
46             Production bad_production(0, production_inputs);
47
48             error_flag = false;
49         } catch (...) {
50             // Task failed successfully! =P
51         }
52         if (not error_flag) {
53             expectedErrorNotDetected(__FILE__, __LINE__);
54         }
55
56         ProductionInputs production_inputs;
57
58         Production test_production(8760, production_inputs);
59
60         // ===== END CONSTRUCTION ===== //
61
62
63
64         // ===== ATTRIBUTES ===== //
65
66         testTruth(
67             not production_inputs.print_flag,
68             __FILE__,
69             __LINE__
70 );
71
72         testFloatEquals(
73             production_inputs.nominal_inflation_annual,
74             0.02,
75             __FILE__,
76             __LINE__
77 );
78
79         testFloatEquals(
80             production_inputs.nominal_discount_annual,
81             0.04,
82             __FILE__,
83             __LINE__
84 );
85
86         testFloatEquals(
87             test_production.n_points,
88             8760,
89             __FILE__,
90             __LINE__
91 );
92
93         testFloatEquals(
94             test_production.capacity_kW,
95             100,
96             __FILE__,
97             __LINE__
98 );
99
100         testFloatEquals(
101             test_production.real_discount_annual,
102             0.0196078431372549,
103             __FILE__,
104             __LINE__
105 );
106

```

```

107 testFloatEquals(
108     test_production.production_vec_kW.size(),
109     8760,
110     __FILE__,
111     __LINE__
112 );
113
114 testFloatEquals(
115     test_production.dispatch_vec_kW.size(),
116     8760,
117     __FILE__,
118     __LINE__
119 );
120
121 testFloatEquals(
122     test_production.storage_vec_kW.size(),
123     8760,
124     __FILE__,
125     __LINE__
126 );
127
128 testFloatEquals(
129     test_production.curtailment_vec_kW.size(),
130     8760,
131     __FILE__,
132     __LINE__
133 );
134
135 testFloatEquals(
136     test_production.capital_cost_vec.size(),
137     8760,
138     __FILE__,
139     __LINE__
140 );
141
142 testFloatEquals(
143     test_production.operation_maintenance_cost_vec.size(),
144     8760,
145     __FILE__,
146     __LINE__
147 );
148
149 // ===== END ATTRIBUTES ===== //
150
151 } /* try */
152
153 catch (...) {
154     //...
155
156     printGold(" ..... ");
157     printRed("FAIL");
158     std::cout << std::endl;
159     throw;
160 }
161
162
163
164 printGold(" ..... ");
165 printGreen("PASS");
166 std::cout << std::endl;
167 return 0;
168
169 } /* main() */

```

5.39 test/source/Storage/test_Lilon.cpp File Reference

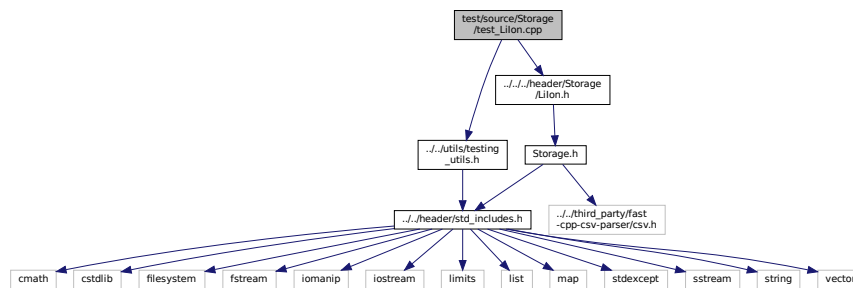
Testing suite for [Lilon](#) class.

```

#include "../utils/testing_utils.h"
#include "../header/Storage/LiIon.h"

```

Include dependency graph for test_Lilon.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.39.1 Detailed Description

Testing suite for [Lilon](#) class.

A suite of tests for the [Lilon](#) class.

5.39.2 Function Documentation

5.39.2.1 main()

```

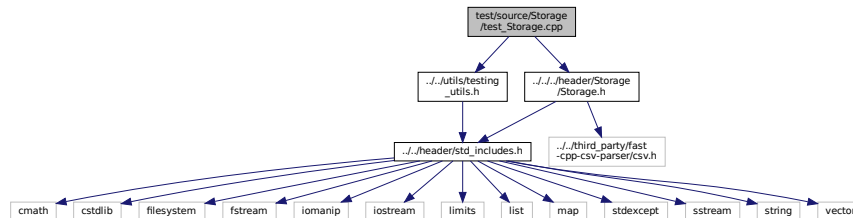
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Storage <-- LiIon");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38         //...
    39     }
    40
    41     catch (...) {
    42         //...
    43
    44         printGold(" ..... ");
    45         printRed("FAIL");
    46         std::cout << std::endl;
    47         throw;
    48     }
    49
    50
    51     printGold(" ..... ");
    52     printGreen("PASS");
    53     std::cout << std::endl;
    54     return 0;
    55 } /* main() */

```

5.40 test/source/Storage/test_Storage.cpp File Reference

Testing suite for [Storage](#) class.

```
#include "../utils/testing_utils.h"
#include "../../header/Storage/Storage.h"
Include dependency graph for test_Storage.cpp:
```



Functions

- [int main](#) (int argc, char **argv)

5.40.1 Detailed Description

Testing suite for [Storage](#) class.

A suite of tests for the [Storage](#) class.

5.40.2 Function Documentation

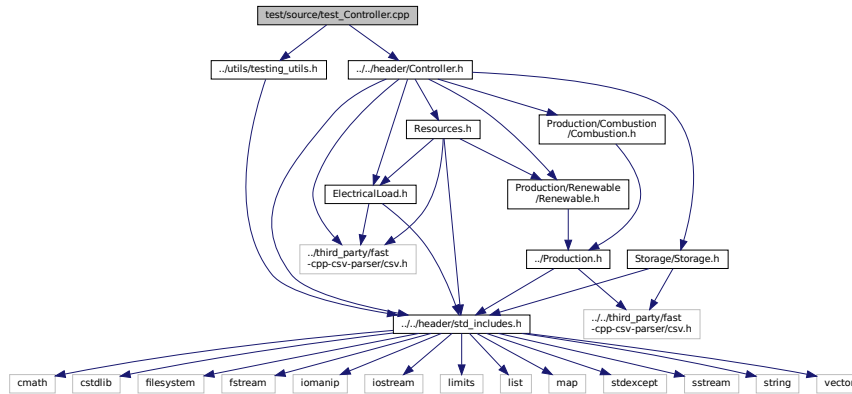
5.40.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Storage");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
    38         //...
    39     }
    40
    41     catch (...) {
    42         //...
    43
    44         printGold(" ..... ");
    45         printRed("FAIL");
    46         std::cout << std::endl;
    47         throw;
    48     }
    49
    50
    51     printGold(" ..... ");
    52     printGreen("PASS");
    53     std::cout << std::endl;
    54     return 0;
    55 } /* main() */
```


5.41 test/source/test_Controller.cpp File Reference

Testing suite for [Controller](#) class.

```
#include "../utils/testing_utils.h"
#include "../../header/Controller.h"
Include dependency graph for test_Controller.cpp:
```



Functions

- int [main](#) (int argc, char **argv)

5.41.1 Detailed Description

Testing suite for [Controller](#) class.

A suite of tests for the [Controller](#) class.

5.41.2 Function Documentation

5.41.2.1 main()

```
int main (
    int argc,
    char ** argv )
{
    27 {
    28     #ifdef _WIN32
    29         activateVirtualTerminal();
    30     #endif /* _WIN32 */
    31
    32     printGold("\tTesting Controller");
    33
    34     srand(time(NULL));
    35
    36
    37     try {
```

```

38
39 // ===== CONSTRUCTION ===== //
40
41 Controller test_controller;
42
43 // ===== END CONSTRUCTION =====//
44
45
46
47 // ===== ATTRIBUTES ===== //
48
49 //...
50
51 // ===== END ATTRIBUTES ===== //
52
53
54
55 // ===== METHODS ===== //
56
57 //...
58
59 // ===== END METHODS ===== //
60
61 } /* try */
62
63
64 catch (...) {
65     //...
66
67     printGold(" ..... ");
68     printRed("FAIL");
69     std::cout << std::endl;
70     throw;
71 }
72
73
74 printGold(" ..... ");
75 printGreen("PASS");
76 std::cout << std::endl;
77 return 0;
78 } /* main() */

```

5.42 test/source/test_ElectricalLoad.cpp File Reference

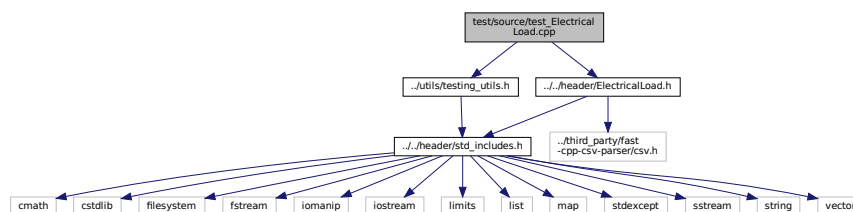
Testing suite for [ElectricalLoad](#) class.

```

#include "../utils/testing_utils.h"
#include "../..header/ElectricalLoad.h"

```

Include dependency graph for test_ElectricalLoad.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.42.1 Detailed Description

Testing suite for [ElectricalLoad](#) class.

A suite of tests for the [ElectricalLoad](#) class.

5.42.2 Function Documentation

5.42.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting ElectricalLoad");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         std::string path_2_electrical_load_time_series =
42             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
43
44         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
45
46         // ===== END CONSTRUCTION ===== //
47
48
49
50         // ===== ATTRIBUTES ===== //
51
52         testTruth(
53             test_electrical_load.path_2_electrical_load_time_series ==
54             path_2_electrical_load_time_series,
55             __FILE__,
56             __LINE__
57 );
58
59         testFloatEquals(
60             test_electrical_load.n_points,
61             8760,
62             __FILE__,
63             __LINE__
64 );
65
66         testFloatEquals(
67             test_electrical_load.n_years,
68             0.999886,
69             __FILE__,
70             __LINE__
71 );
72
73         testFloatEquals(
74             test_electrical_load.min_load_kW,
75             82.1211213927802,
76             __FILE__,
77             __LINE__
78 );
79
80         testFloatEquals(
81             test_electrical_load.mean_load_kW,
82             258.373472633202,
83             __FILE__,
84             __LINE__
85 );
86
87
88         testFloatEquals(
89             test_electrical_load.max_load_kW,
90             500,
91             __FILE__,
92             __LINE__
93 );
94
95
96     std::vector<double> expected_dt_vec_hrs (48, 1);
97

```

```

98 std::vector<double> expected_time_vec_hrs = {
99     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
100    12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
101    24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
102    36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
103 };
104
105 std::vector<double> expected_load_vec_kW = {
106     360.253836463674,
107     355.171277826775,
108     353.776453532298,
109     353.75405737934,
110     346.592867404975,
111     340.132411175118,
112     337.354867340578,
113     340.644115618736,
114     363.639028500678,
115     378.787797779238,
116     372.215798201712,
117     395.093925731298,
118     402.325427142659,
119     386.907725462306,
120     380.709170928091,
121     372.062070914977,
122     372.328646856954,
123     391.841444284136,
124     394.029351759596,
125     383.369407765254,
126     381.093099675206,
127     382.604158946193,
128     390.744843709034,
129     383.13949492437,
130     368.150393976985,
131     364.629744480226,
132     363.572736804082,
133     359.854924202248,
134     355.207590170267,
135     349.094656012401,
136     354.365935871597,
137     343.380608328546,
138     404.673065729266,
139     486.296896820126,
140     480.225974100847,
141     457.318764401085,
142     418.177339948609,
143     414.399018364126,
144     409.678420185754,
145     404.768766016563,
146     401.699589920585,
147     402.44339040654,
148     398.138372541906,
149     396.010498627646,
150     390.165117432277,
151     375.850429417013,
152     365.567100746484,
153     365.429624610923
154 };
155
156 for (int i = 0; i < 48; i++) {
157     testFloatEquals(
158         test_electrical_load.dt_vec_hrs[i],
159         expected_dt_vec_hrs[i],
160         __FILE__,
161         __LINE__
162     );
163
164     testFloatEquals(
165         test_electrical_load.time_vec_hrs[i],
166         expected_time_vec_hrs[i],
167         __FILE__,
168         __LINE__
169     );
170
171     testFloatEquals(
172         test_electrical_load.load_vec_kW[i],
173         expected_load_vec_kW[i],
174         __FILE__,
175         __LINE__
176     );
177 }
178
179 // ===== END ATTRIBUTES ===== //
180
181 } /* try */
182
183
184 catch (...) {

```

```

185     //...
186
187     printGold(" ..... ");
188     printRed("FAIL");
189     std::cout << std::endl;
190     throw;
191 }
192
193
194 printGold(" ..... ");
195 printGreen("PASS");
196 std::cout << std::endl;
197 return 0;
198 } /* main() */

```

5.43 test/source/test_Model.cpp File Reference

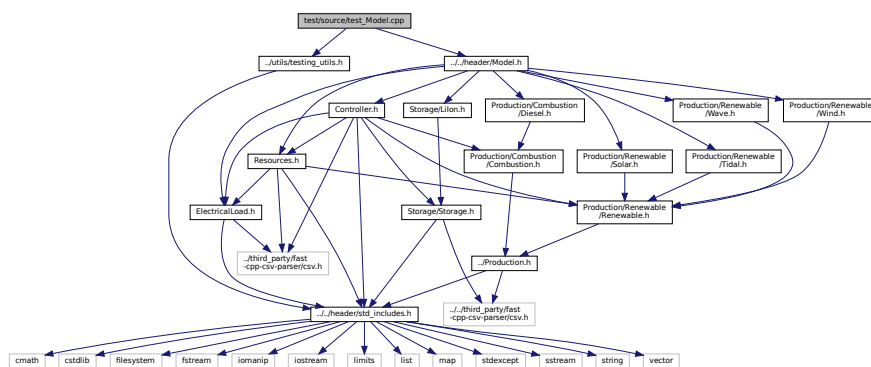
Testing suite for [Model](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Model.h"

```

Include dependency graph for test_Model.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.43.1 Detailed Description

Testing suite for [Model](#) class.

A suite of tests for the [Model](#) class.

5.43.2 Function Documentation

5.43.2.1 main()

```

int main (
    int argc,
    char ** argv )
27 {
28     #ifdef _WIN32
29         activateVirtualTerminal();
30     #endif /* _WIN32 */
31
32     printGold("\tTesting Model");
33
34     srand(time(NULL));
35
36
37     try {
38
39         // ===== CONSTRUCTION ===== //
40
41         bool error_flag = true;
42
43         try {
44             ModelInputs bad_model_inputs;
45             bad_model_inputs.path_2_electrical_load_time_series =
46                 "data/test/bad_path_240984069830.csv";
47
48             Model bad_model(bad_model_inputs);
49
50             error_flag = false;
51         } catch (...) {
52             // Task failed successfully! =P
53         }
54         if (not error_flag) {
55             expectedErrorNotDetected(__FILE__, __LINE__);
56         }
57
58         std::string path_2_electrical_load_time_series =
59             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
60
61         ModelInputs test_model_inputs;
62         test_model_inputs.path_2_electrical_load_time_series =
63             path_2_electrical_load_time_series;
64
65         Model test_model(test_model_inputs);
66
67         // ===== END CONSTRUCTION ===== //
68
69
70         // ===== ATTRIBUTES ===== //
71
72         testTruth(
73             test_model.electrical_load.path_2_electrical_load_time_series ==
74             path_2_electrical_load_time_series,
75             __FILE__,
76             __LINE__
77 );
78
79         testFloatEquals(
80             test_model.electrical_load.n_points,
81             8760,
82             __FILE__,
83             __LINE__
84 );
85
86         testFloatEquals(
87             test_model.electrical_load.n_years,
88             0.999886,
89             __FILE__,
90             __LINE__
91 );
92
93         testFloatEquals(
94             test_model.electrical_load.min_load_kW,
95             82.1211213927802,
96             __FILE__,
97             __LINE__
98 );
99
100        testFloatEquals(
101            test_model.electrical_load.mean_load_kW,
102            258.373472633202,
103            __FILE__,
104            __LINE__
105 );
106

```

```

107
108 testFloatEquals(
109     test_model.electrical_load.max_load_kW,
110     500,
111     __FILE__,
112     __LINE__
113 );
114
115
116 std::vector<double> expected_dt_vec_hrs (48, 1);
117
118 std::vector<double> expected_time_vec_hrs = {
119     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
120     12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
121     24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
122     36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
123 };
124
125 std::vector<double> expected_load_vec_kW = {
126     360.253836463674,
127     355.171277826775,
128     353.776453532298,
129     353.75405737934,
130     346.592867404975,
131     340.132411175118,
132     337.354867340578,
133     340.644115618736,
134     363.639028500678,
135     378.787797779238,
136     372.215798201712,
137     395.093925731298,
138     402.325427142659,
139     386.907725462306,
140     380.709170928091,
141     372.062070914977,
142     372.328646856954,
143     391.841444284136,
144     394.029351759596,
145     383.369407765254,
146     381.093099675206,
147     382.604158946193,
148     390.744843709034,
149     383.13949492437,
150     368.150393976985,
151     364.629744480226,
152     363.572736804082,
153     359.854924202248,
154     355.207590170267,
155     349.094656012401,
156     354.365935871597,
157     343.380608328546,
158     404.673065729266,
159     486.296896820126,
160     480.225974100847,
161     457.318764401085,
162     418.177339948609,
163     414.399018364126,
164     409.678420185754,
165     404.768766016563,
166     401.699589920585,
167     402.44339040654,
168     398.138372541906,
169     396.010498627646,
170     390.165117432277,
171     375.850429417013,
172     365.567100746484,
173     365.429624610923
174 };
175
176 for (int i = 0; i < 48; i++) {
177     testFloatEquals(
178         test_model.electrical_load.dt_vec_hrs[i],
179         expected_dt_vec_hrs[i],
180         __FILE__,
181         __LINE__
182     );
183
184     testFloatEquals(
185         test_model.electrical_load.time_vec_hrs[i],
186         expected_time_vec_hrs[i],
187         __FILE__,
188         __LINE__
189     );
190
191     testFloatEquals(
192         test_model.electrical_load.load_vec_kW[i],
193         expected_load_vec_kW[i],

```

```

194     __FILE__,
195     __LINE__
196 );
197 }
198
199 // ===== END ATTRIBUTES ===== //
200
201
202
203 // ===== METHODS ===== //
204
205 // add Solar resource
206 int solar_resource_key = 0;
207 std::string path_2_solar_resource_data =
208     "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
209
210 test_model.addResource(
211     RenewableType :: SOLAR,
212     path_2_solar_resource_data,
213     solar_resource_key
214 );
215
216 std::vector<double> expected_solar_resource_vec_kWm2 = {
217     0,
218     0,
219     0,
220     0,
221     0,
222     0,
223     8.51702662684015E-05,
224     0.000348341567045,
225     0.00213793728593,
226     0.004099863613322,
227     0.000997135230553,
228     0.009534527624657,
229     0.022927996790616,
230     0.0136071715294,
231     0.002535134127751,
232     0.005206897515821,
233     0.005627658648597,
234     0.000701186722215,
235     0.00017119827089,
236     0,
237     0,
238     0,
239     0,
240     0,
241     0,
242     0,
243     0,
244     0,
245     0,
246     0,
247     0,
248     0.000141055102242,
249     0.00084525014743,
250     0.024893647822702,
251     0.091245556190749,
252     0.158722176731637,
253     0.152859680515876,
254     0.149922903895116,
255     0.13049996570866,
256     0.03081254222795,
257     0.001218928911125,
258     0.000206092647423,
259     0,
260     0,
261     0,
262     0,
263     0,
264     0
265 };
266
267 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
268     testFloatEquals(
269         test_model.resources.resource_map_1D[solar_resource_key][i],
270         expected_solar_resource_vec_kWm2[i],
271         __FILE__,
272         __LINE__
273     );
274 }
275
276
277 // add Tidal resource
278 int tidal_resource_key = 1;
279 std::string path_2_tidal_resource_data =
280     "data/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv";

```



```

281
282 test_model.addResource(
283     RenewableType :: TIDAL,
284     path_2_tidal_resource_data,
285     tidal_resource_key
286 );
287
288
289 // add Wave resource
290 int wave_resource_key = 2;
291 std::string path_2_wave_resource_data =
292     "data/test/waves_H_s_peak-8m_T_e_peak-15s_lyr_dt-1hr.csv";
293
294 test_model.addResource(
295     RenewableType :: WAVE,
296     path_2_wave_resource_data,
297     wave_resource_key
298 );
299
300
301 // add Wind resource
302 int wind_resource_key = 3;
303 std::string path_2_wind_resource_data =
304     "data/test/wind_speed_peak-25ms_lyr_dt-1hr.csv";
305
306 test_model.addResource(
307     RenewableType :: WIND,
308     path_2_wind_resource_data,
309     wind_resource_key
310 );
311
312
313 // add Diesel assets
314 DieselInputs diesel_inputs;
315 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
316
317 test_model.addDiesel(diesel_inputs);
318
319 testFloatEquals(
320     test_model.combustion_ptr_vec.size(),
321     1,
322     __FILE__,
323     __LINE__
324 );
325
326 testFloatEquals(
327     test_model.combustion_ptr_vec[0]->type,
328     CombustionType :: DIESEL,
329     __FILE__,
330     __LINE__
331 );
332
333 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 150;
334
335 test_model.addDiesel(diesel_inputs);
336
337 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 250;
338
339 test_model.addDiesel(diesel_inputs);
340
341 testFloatEquals(
342     test_model.combustion_ptr_vec.size(),
343     3,
344     __FILE__,
345     __LINE__
346 );
347
348 std::vector<int> expected_diesel_capacity_vec_kW = {100, 150, 250};
349
350 for (int i = 0; i < 3; i++) {
351     testFloatEquals(
352         test_model.combustion_ptr_vec[i]->capacity_kW,
353         expected_diesel_capacity_vec_kW[i],
354         __FILE__,
355         __LINE__
356     );
357 }
358
359 diesel_inputs.combustion_inputs.production_inputs.capacity_kW = 100;
360
361 for (int i = 0; i < 2 * ((double)rand() / RAND_MAX); i++) {
362     test_model.addDiesel(diesel_inputs);
363 }
364
365
366 // add Solar asset
367 SolarInputs solar_inputs;

```

```
368 solar_inputs.resource_key = solar_resource_key;
369
370 test_model.addSolar(solar_inputs);
371
372 testFloatEquals(
373     test_model.renewable_ptr_vec.size(),
374     1,
375     __FILE__,
376     __LINE__
377 );
378
379 testFloatEquals(
380     test_model.renewable_ptr_vec[0]->type,
381     RenewableType :: SOLAR,
382     __FILE__,
383     __LINE__
384 );
385
386
387 // add Tidal asset
388 TidalInputs tidal_inputs;
389 tidal_inputs.resource_key = tidal_resource_key;
390
391 test_model.addTidal(tidal_inputs);
392
393 testFloatEquals(
394     test_model.renewable_ptr_vec.size(),
395     2,
396     __FILE__,
397     __LINE__
398 );
399
400 testFloatEquals(
401     test_model.renewable_ptr_vec[1]->type,
402     RenewableType :: TIDAL,
403     __FILE__,
404     __LINE__
405 );
406
407
408 // add Wave asset
409 WaveInputs wave_inputs;
410 wave_inputs.resource_key = wave_resource_key;
411
412 test_model.addWave(wave_inputs);
413
414 testFloatEquals(
415     test_model.renewable_ptr_vec.size(),
416     3,
417     __FILE__,
418     __LINE__
419 );
420
421 testFloatEquals(
422     test_model.renewable_ptr_vec[2]->type,
423     RenewableType :: WAVE,
424     __FILE__,
425     __LINE__
426 );
427
428
429 // add Wind asset
430 WindInputs wind_inputs;
431 wind_inputs.resource_key = wind_resource_key;
432
433 test_model.addWind(wind_inputs);
434
435 testFloatEquals(
436     test_model.renewable_ptr_vec.size(),
437     4,
438     __FILE__,
439     __LINE__
440 );
441
442 testFloatEquals(
443     test_model.renewable_ptr_vec[3]->type,
444     RenewableType :: WIND,
445     __FILE__,
446     __LINE__
447 );
448
449
450 // run
451 test_model.run();
452
453 for (int i = 0; i < test_model.electrical_load.n_points; i++) {
454     testLessThanOrEqualTo(
```

```

455         test_model.controller.net_load_vec_kW[i],
456         test_model.electrical_load.max_load_kW,
457         __FILE__,
458         __LINE__
459     );
460 }
461
462 testGreaterThan(
463     test_model.net_present_cost,
464     0,
465     __FILE__,
466     __LINE__
467 );
468
469 testFloatEquals(
470     test_model.total_dispatch_discharge_kWh,
471     2263351.62026685,
472     __FILE__,
473     __LINE__
474 );
475
476 testGreaterThan(
477     test_model.levelized_cost_of_energy_kWh,
478     0,
479     __FILE__,
480     __LINE__
481 );
482
483 testGreaterThan(
484     test_model.total_fuel_consumed_L,
485     0,
486     __FILE__,
487     __LINE__
488 );
489
490 testGreaterThan(
491     test_model.total_emissions.CO2_kg,
492     0,
493     __FILE__,
494     __LINE__
495 );
496
497 testGreaterThan(
498     test_model.total_emissions.CO_kg,
499     0,
500     __FILE__,
501     __LINE__
502 );
503
504 testGreaterThan(
505     test_model.total_emissions.NOx_kg,
506     0,
507     __FILE__,
508     __LINE__
509 );
510
511 testGreaterThan(
512     test_model.total_emissions.SOx_kg,
513     0,
514     __FILE__,
515     __LINE__
516 );
517
518 testGreaterThan(
519     test_model.total_emissions.CH4_kg,
520     0,
521     __FILE__,
522     __LINE__
523 );
524
525 testGreaterThan(
526     test_model.total_emissions.PM_kg,
527     0,
528     __FILE__,
529     __LINE__
530 );
531
532 // ===== END METHODS ===== //
533
534 } /* try */
535
536
537 catch (...) {
538     //...
539
540     printGold(" ..... ");
541     printRed("FAIL");

```

```

542     std::cout << std::endl;
543     throw;
544 }
545
546
547 printGold(" ..... ");
548 printGreen("PASS");
549 std::cout << std::endl;
550 return 0;
551 } /* main() */

```

5.44 test/source/test_Resources.cpp File Reference

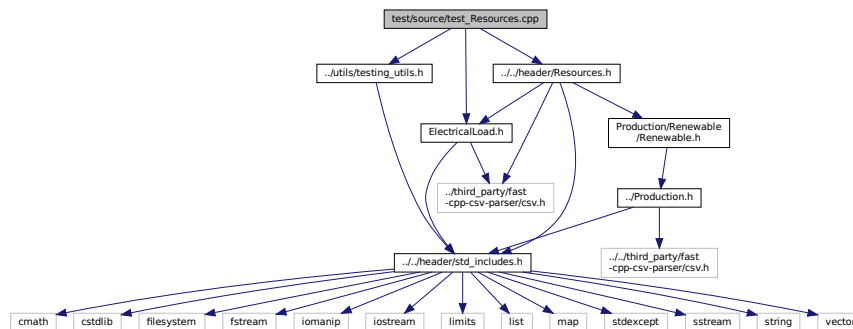
Testing suite for [Resources](#) class.

```

#include "../utils/testing_utils.h"
#include "../../header/Resources.h"
#include "../../header/ElectricalLoad.h"

```

Include dependency graph for test_Resources.cpp:



Functions

- int [main](#) (int argc, char **argv)

5.44.1 Detailed Description

Testing suite for [Resources](#) class.

A suite of tests for the [Resources](#) class.

5.44.2 Function Documentation

5.44.2.1 main()

```

int main (
    int argc,
    char ** argv )

28 {
29     #ifdef _WIN32
30         activateVirtualTerminal();
31     #endif /* _WIN32 */
32
33     printGold("\tTesting Resources");
34
35     srand(time(NULL));
36
37
38     try {
39
40         // ===== CONSTRUCTION ===== //
41
42         std::string path_2_electrical_load_time_series =
43             "data/test/electrical_load_generic_peak-500kW_1yr_dt-1hr.csv";
44
45         ElectricalLoad test_electrical_load(path_2_electrical_load_time_series);
46
47         Resources test_resources;
48
49         // ===== END CONSTRUCTION ===== //
50
51
52
53         // ===== ATTRIBUTES ===== //
54
55         testFloatEquals(
56             test_resources.resource_map_1D.size(),
57             0,
58             __FILE__,
59             __LINE__
60         );
61
62         testFloatEquals(
63             test_resources.path_map_1D.size(),
64             0,
65             __FILE__,
66             __LINE__
67         );
68
69         testFloatEquals(
70             test_resources.resource_map_2D.size(),
71             0,
72             __FILE__,
73             __LINE__
74         );
75
76         testFloatEquals(
77             test_resources.path_map_2D.size(),
78             0,
79             __FILE__,
80             __LINE__
81         );
82
83         // ===== END ATTRIBUTES ===== //
84
85
86         // ===== METHODS ===== //
87
88         int solar_resource_key = 0;
89         std::string path_2_solar_resource_data =
90             "data/test/solar_GHI_peak-1kWm2_1yr_dt-1hr.csv";
91
92         test_resources.addResource(
93             RenewableType::SOLAR,
94             path_2_solar_resource_data,
95             solar_resource_key,
96             &test_electrical_load
97         );
98
99         bool error_flag = true;
100         try {
101             test_resources.addResource(
102                 RenewableType::SOLAR,
103                 path_2_solar_resource_data,
104                 solar_resource_key,
105                 &test_electrical_load
106             );
107

```

```

108     error_flag = false;
109 } catch (...) {
110     // Task failed successfully! =P
111 }
112 if (not error_flag) {
113     expectedErrorNotDetected(__FILE__, __LINE__);
114 }
115
116
117 try {
118     std::string path_2_solar_resource_data_BAD_TIMES =
119         "data/test/solar_GHI_peak-1kWm2_lyr_dt-1hr_BAD_TIMES.csv";
120
121     test_resources.addResource(
122         RenewableType::SOLAR,
123         path_2_solar_resource_data_BAD_TIMES,
124         -1,
125         &test_electrical_load
126     );
127
128     error_flag = false;
129 } catch (...) {
130     // Task failed successfully! =P
131 }
132 if (not error_flag) {
133     expectedErrorNotDetected(__FILE__, __LINE__);
134 }
135
136
137 try {
138     std::string path_2_solar_resource_data_BAD_LENGTH =
139         "data/test/solar_GHI_peak-1kWm2_lyr_dt-1hr_BAD_LENGTH.csv";
140
141     test_resources.addResource(
142         RenewableType::SOLAR,
143         path_2_solar_resource_data_BAD_LENGTH,
144         -2,
145         &test_electrical_load
146     );
147
148     error_flag = false;
149 } catch (...) {
150     // Task failed successfully! =P
151 }
152 if (not error_flag) {
153     expectedErrorNotDetected(__FILE__, __LINE__);
154 }
155
156 std::vector<double> expected_solar_resource_vec_kWm2 = {
157     0,
158     0,
159     0,
160     0,
161     0,
162     0,
163     8.51702662684015E-05,
164     0.000348341567045,
165     0.00213793728593,
166     0.004099863613322,
167     0.000997135230553,
168     0.009534527624657,
169     0.022927996790616,
170     0.0136071715294,
171     0.002535134127751,
172     0.005206897515821,
173     0.005627658648597,
174     0.000701186722215,
175     0.00017119827089,
176     0,
177     0,
178     0,
179     0,
180     0,
181     0,
182     0,
183     0,
184     0,
185     0,
186     0,
187     0,
188     0.000141055102242,
189     0.00084525014743,
190     0.024893647822702,
191     0.091245556190749,
192     0.158722176731637,
193     0.152859680515876,
194     0.149922903895116,

```

```

195     0.13049996570866,
196     0.03081254222795,
197     0.001218928911125,
198     0.000206092647423,
199     0,
200     0,
201     0,
202     0,
203     0,
204     0
205 };
206
207 for (size_t i = 0; i < expected_solar_resource_vec_kWm2.size(); i++) {
208     testFloatEquals(
209         test_resources.resource_map_1D[solar_resource_key][i],
210         expected_solar_resource_vec_kWm2[i],
211         __FILE__,
212         __LINE__
213     );
214 }
215
216
217 int tidal_resource_key = 1;
218 std::string path_2_tidal_resource_data =
219     "data/test/tidal_speed_peak-3ms_1yr_dt-1hr.csv";
220
221 test_resources.addResource(
222     RenewableType::TIDAL,
223     path_2_tidal_resource_data,
224     tidal_resource_key,
225     &test_electrical_load
226 );
227
228 std::vector<double> expected_tidal_resource_vec_ms = {
229     0.347439913040533,
230     0.770545522195602,
231     0.731352084836198,
232     0.293389814389542,
233     0.209959110813115,
234     0.610609623896497,
235     1.78067162013604,
236     2.53522775118089,
237     2.75966627832024,
238     2.52101111143895,
239     2.05389330201031,
240     1.3461515862445,
241     0.28909254878384,
242     0.897754086048563,
243     1.71406453837407,
244     1.85047408742869,
245     1.71507908595979,
246     1.33540349705416,
247     0.434586143463003,
248     0.500623815700637,
249     1.37172172646733,
250     1.68294125491228,
251     1.56101300975417,
252     1.04925834219412,
253     0.211395463930223,
254     1.03720048903385,
255     1.85059536356448,
256     1.85203242794517,
257     1.4091471616277,
258     0.767776539039899,
259     0.251464906990961,
260     1.47018469375652,
261     2.36260493698197,
262     2.46653750048625,
263     2.12851908739291,
264     1.62783753197988,
265     0.734594890957439,
266     0.441886297300355,
267     1.6574418350918,
268     2.0684558286637,
269     1.87717416992136,
270     1.58871262337931,
271     1.03451227609235,
272     0.193371305159817,
273     0.976400122458815,
274     1.6583227369707,
275     1.76690616570953,
276     1.54801328553115
277 };
278
279 for (size_t i = 0; i < expected_tidal_resource_vec_ms.size(); i++) {
280     testFloatEquals(
281         test_resources.resource_map_1D[tidal_resource_key][i],

```

```
282         expected_tidal_resource_vec_ms[i],
283         __FILE__,
284         __LINE__
285     );
286 }
287
288
289 int wave_resource_key = 2;
290 std::string path_2_wave_resource_data =
291     "data/test/waves_H_s_peak-8m_T_e_peak-15s_1yr_dt-1hr.csv";
292
293 test_resources.addResource(
294     RenewableType::WAVE,
295     path_2_wave_resource_data,
296     wave_resource_key,
297     &test_electrical_load
298 );
299
300 std::vector<double> expected_significant_wave_height_vec_m = {
301     4.26175222125028,
302     4.25020976167872,
303     4.25656524330349,
304     4.27193854786718,
305     4.28744955711233,
306     4.29421815278154,
307     4.2839937266082,
308     4.25716982457976,
309     4.22419391611483,
310     4.19588925217606,
311     4.17338788587412,
312     4.14672746914214,
313     4.10560041173665,
314     4.05074966447193,
315     3.9953696962433,
316     3.95316976150866,
317     3.92771018142378,
318     3.91129562488595,
319     3.89558312094911,
320     3.87861093931749,
321     3.86538307240754,
322     3.86108961027929,
323     3.86459448853189,
324     3.86796474016882,
325     3.86357412779993,
326     3.85554872014731,
327     3.86044266668675,
328     3.89445961915999,
329     3.95554798115731,
330     4.02265508610476,
331     4.07419587011404,
332     4.10314247143958,
333     4.11738045085928,
334     4.12554995596708,
335     4.12923992001675,
336     4.1229292327442,
337     4.10123955307441,
338     4.06748827895363,
339     4.0336230651344,
340     4.01134236393876,
341     4.00136570034559,
342     3.99368787690411,
343     3.97820924247644,
344     3.95369335178055,
345     3.92742545608532,
346     3.90683362771686,
347     3.89331520944006,
348     3.88256045801583
349 };
350
351 std::vector<double> expected_energy_period_vec_s = {
352     10.4456008226821,
353     10.4614151137651,
354     10.4462827795433,
355     10.4127692097884,
356     10.3734397942723,
357     10.3408599227669,
358     10.32637292093,
359     10.3245412676322,
360     10.310409818185,
361     10.2589529840966,
362     10.1728100603103,
363     10.0862908658929,
364     10.03480243813,
365     10.023673635806,
366     10.0243418565116,
367     10.0063487117653,
368     9.9605030228607,
```



```

369     9.9011999635568,
370     9.84451822125472,
371     9.79726875879626,
372     9.75614594835158,
373     9.7173447961368,
374     9.68342904390577,
375     9.66380508567062,
376     9.6674009575699,
377     9.68927134575103,
378     9.70979984863046,
379     9.70967357906908,
380     9.68983025704562,
381     9.6722855524805,
382     9.67973599910003,
383     9.71977125328293,
384     9.78450442291421,
385     9.86532355233449,
386     9.96158937600019,
387     10.0807018356507,
388     10.2291022504937,
389     10.39458528356,
390     10.5464393581004,
391     10.6553277500484,
392     10.7245553190084,
393     10.7893127285064,
394     10.8846512240849,
395     11.0148158739075,
396     11.1544325654719,
397     11.2772785848343,
398     11.3744362756187,
399     11.4533643503183
400 };
401
402 for (size_t i = 0; i < expected_significant_wave_height_vec_m.size(); i++) {
403     testFloatEquals (
404         test_resources.resource_map_2D[wave_resource_key][i][0],
405         expected_significant_wave_height_vec_m[i],
406         __FILE__,
407         __LINE__
408     );
409
410     testFloatEquals (
411         test_resources.resource_map_2D[wave_resource_key][i][1],
412         expected_energy_period_vec_s[i],
413         __FILE__,
414         __LINE__
415     );
416 }
417
418
419 int wind_resource_key = 3;
420 std::string path_2_wind_resource_data =
421     "data/test/wind_speed_peak-25ms_1yr_dt-1hr.csv";
422
423 test_resources.addResource (
424     RenewableType::WIND,
425     path_2_wind_resource_data,
426     wind_resource_key,
427     &test_electrical_load
428 );
429
430 std::vector<double> expected_wind_resource_vec_ms = {
431     6.88566688469997,
432     5.02177105466549,
433     3.74211715899568,
434     5.67169579985362,
435     4.90670669971858,
436     4.29586955031368,
437     7.41155377205065,
438     10.2243290476943,
439     13.1258696725555,
440     13.7016198628274,
441     16.2481482330233,
442     16.5096744355418,
443     13.4354482206162,
444     14.0129230731609,
445     14.5554549260515,
446     13.4454539065912,
447     13.3447169512094,
448     11.7372615098554,
449     12.7200070078013,
450     10.6421127908149,
451     6.09869498990661,
452     5.66355596602321,
453     4.97316966910831,
454     3.48937138360567,
455     2.15917470979169,

```

```

456     1.29061103587027,
457     3.43475751425219,
458     4.11706326260927,
459     4.28905275747408,
460     5.75850263196241,
461     8.98293663055264,
462     11.7069822941315,
463     12.4031987075858,
464     15.4096570910089,
465     16.6210843829552,
466     13.3421219142573,
467     15.2112831900548,
468     18.350864533037,
469     15.8751799822971,
470     15.3921198799796,
471     15.9729192868434,
472     12.4728950178772,
473     10.177050481096,
474     10.7342247355551,
475     8.98846695631389,
476     4.14671169124739,
477     3.17256452697149,
478     3.40036336968628
479 };
480
481 for (size_t i = 0; i < expected_wind_resource_vec_ms.size(); i++) {
482     testFloatEquals(
483         test_resources.resource_map_1D[wind_resource_key][i],
484         expected_wind_resource_vec_ms[i],
485         __FILE__,
486         __LINE__
487     );
488 }
489
490 // ===== END METHODS ===== //
491
492 } /* try */
493
494
495 catch (...) {
496     printGold(" ..... ");
497     printRed("FAIL");
498     std::cout << std::endl;
499     throw;
500 }
501
502
503 printGold(" ..... ");
504 printGreen("PASS");
505 std::cout << std::endl;
506 return 0;
507 } /* main() */

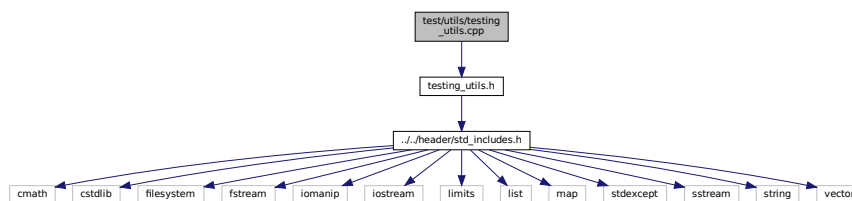
```

5.45 test/utils/testing_utils.cpp File Reference

Header file for various PGMcpp testing utilities.

```
#include "testing_utils.h"
```

Include dependency graph for testing_utils.cpp:



Functions

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

5.45.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.45.2 Function Documentation

5.45.2.1 `expectedErrorNotDetected()`

```
void expectedErrorNotDetected (
    std::string file,
    int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```
432 {
433     std::string error_str = "\n ERROR   failed to throw expected error prior to line ";
434     error_str += std::to_string(line);
```

```
435     error_str += " of ";
436     error_str += file;
437
438     #ifdef _WIN32
439         std::cout << error_str << std::endl;
440     #endif
441
442     throw std::runtime_error(error_str);
443     return;
444 } /* expectedErrorNotDetected() */
```

5.45.2.2 printGold()

```
void printGold (
    std::string input_str )
```

A function that sends gold text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```
84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */
```

5.45.2.3 printGreen()

```
void printGreen (
    std::string input_str )
```

A function that sends green text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```
64 {
65     std::cout << "\x1B[32m" << input_str << "\033[0m";
66     return;
67 } /* printGreen() */
```

5.45.2.4 printRed()

```
void printRed (
    std::string input_str )
```

A function that sends red text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to <code>std::cout</code> .
------------------	---

```

104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */

```

5.45.2.5 testFloatEquals()

```

void testFloatEquals (
    double x,
    double y,
    std::string file,
    int line )

```

Tests for the equality of two floating point numbers *x* and *y* (to within `FLOAT_TOLERANCE`).

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in " <code>__FILE__</code> ").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in " <code>__LINE__</code> ").

```

138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154
155     #ifdef _WIN32
156         std::cout << error_str << std::endl;
157     #endif
158
159     throw std::runtime_error(error_str);
160     return;
161 } /* testFloatEquals() */

```

5.45.2.6 testGreaterThan()

```

void testGreaterThan (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x > y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209
210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */

```

5.45.2.7 testGreaterThanOrEqualTo()

```

void testGreaterThanOrEqualTo (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x \geq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

242 {
243     if (x >= y) {
244         return;
245     }
246
247     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
248     error_str += file;
249     error_str += "\tline ";
250     error_str += std::to_string(line);
251     error_str += ":\t\n";
252     error_str += std::to_string(x);
253     error_str += " is not greater than or equal to ";
254     error_str += std::to_string(y);
255     error_str += "\n";
256
257     #ifdef _WIN32
258         std::cout << error_str << std::endl;
259     #endif
260
261     throw std::runtime_error(error_str);

```

```

262     return;
263 } /* testGreaterThanOrEqualTo() */

```

5.45.2.8 testLessThan()

```

void testLessThan (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x < y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

293 {
294     if (x < y) {
295         return;
296     }
297
298     std::string error_str = "ERROR: testLessThan():\t in ";
299     error_str += file;
300     error_str += "\tline ";
301     error_str += std::to_string(line);
302     error_str += ":\t\n";
303     error_str += std::to_string(x);
304     error_str += " is not less than ";
305     error_str += std::to_string(y);
306     error_str += "\n";
307
308     #ifdef _WIN32
309         std::cout << error_str << std::endl;
310     #endif
311
312     throw std::runtime_error(error_str);
313     return;
314 } /* testLessThan() */

```

5.45.2.9 testLessThanOrEqualTo()

```

void testLessThanOrEqualTo (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x \leq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.45.2.10 testTruth()

```

void testTruth (
    bool statement,
    std::string file,
    int line )

```

Tests if the given statement is true.

Parameters

<i>statement</i>	The statement whose truth is to be tested ("1 == 0", for example).
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

392 {
393     if (statement) {
394         return;
395     }
396
397     std::string error_str = "ERROR: testTruth():\t in ";
398     error_str += file;
399     error_str += "\tline ";
400     error_str += std::to_string(line);
401     error_str += ":\t\n";
402     error_str += "Given statement is not true";
403
404     #ifdef _WIN32
405         std::cout << error_str << std::endl;
406     #endif
407
408     throw std::runtime_error(error_str);
409     return;
410 } /* testTruth() */

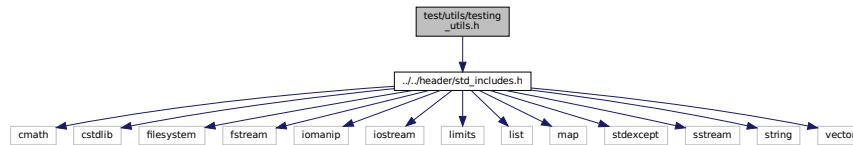
```

5.46 test/utls/testing_utils.h File Reference

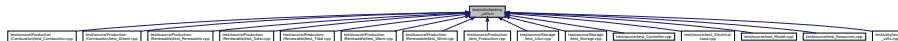
Header file for various PGMcpp testing utilities.


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

Include dependency graph for testing_utils.h:



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



Macros

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

Functions

- void `printGreen` (std::string)
A function that sends green text to std::cout.
- void `printGold` (std::string)
A function that sends gold text to std::cout.
- void `printRed` (std::string)
A function that sends red text to std::cout.
- void `testFloatEquals` (double, double, std::string, int)
Tests for the equality of two floating point numbers *x* and *y* (to within `FLOAT_TOLERANCE`).
- void `testGreaterThan` (double, double, std::string, int)
Tests if *x* > *y*.
- void `testGreaterThanOrEqualTo` (double, double, std::string, int)
Tests if *x* >= *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.

5.46.1 Detailed Description

Header file for various PGMcpp testing utilities.

This is a library of utility functions used throughout the various test suites.

5.46.2 Macro Definition Documentation

5.46.2.1 FLOAT_TOLERANCE

```
#define FLOAT_TOLERANCE 1e-6
```

A tolerance for application to floating point equality tests.

5.46.3 Function Documentation

5.46.3.1 expectedErrorNotDetected()

```
void expectedErrorNotDetected (
    std::string file,
    int line )
```

A utility function to print out a meaningful error message whenever an expected error fails to be thrown/caught/detected.

Parameters

<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
432 {
433     std::string error_str = "\n ERROR   failed to throw expected error prior to line ";
434     error_str += std::to_string(line);
435     error_str += " of ";
436     error_str += file;
437
438     #ifdef _WIN32
439         std::cout << error_str << std::endl;
440     #endif
441
442     throw std::runtime_error(error_str);
443     return;
444 } /* expectedErrorNotDetected() */
```

5.46.3.2 printGold()

```
void printGold (
    std::string input_str )
```

A function that sends gold text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

84 {
85     std::cout << "\x1B[33m" << input_str << "\033[0m";
86     return;
87 } /* printGold() */

```

5.46.3.3 printGreen()

```

void printGreen (
    std::string input_str )

```

A function that sends green text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

64 {
65     std::cout << "\x1B[32m" << input_str << "\033[0m";
66     return;
67 } /* printGreen() */

```

5.46.3.4 printRed()

```

void printRed (
    std::string input_str )

```

A function that sends red text to std::cout.

Parameters

<i>input_str</i>	The text of the string to be sent to std::cout.
------------------	---

```

104 {
105     std::cout << "\x1B[31m" << input_str << "\033[0m";
106     return;
107 } /* printRed() */

```

5.46.3.5 testFloatEquals()

```

void testFloatEquals (
    double x,
    double y,
    std::string file,
    int line )

```

Tests for the equality of two floating point numbers *x* and *y* (to within FLOAT_TOLERANCE).

Parameters

<i>x</i>	The first of two numbers to test.
----------	-----------------------------------

Parameters

<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

138 {
139     if (fabs(x - y) <= FLOAT_TOLERANCE) {
140         return;
141     }
142
143     std::string error_str = "ERROR: testFloatEquals():\t in ";
144     error_str += file;
145     error_str += "\tline ";
146     error_str += std::to_string(line);
147     error_str += ":\t\n";
148     error_str += std::to_string(x);
149     error_str += " and ";
150     error_str += std::to_string(y);
151     error_str += " are not equal to within +/- ";
152     error_str += std::to_string(FLOAT_TOLERANCE);
153     error_str += "\n";
154
155     #ifdef _WIN32
156         std::cout << error_str << std::endl;
157     #endif
158
159     throw std::runtime_error(error_str);
160     return;
161 } /* testFloatEquals() */

```

5.46.3.6 testGreaterThan()

```

void testGreaterThan (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x > y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

191 {
192     if (x > y) {
193         return;
194     }
195
196     std::string error_str = "ERROR: testGreaterThan():\t in ";
197     error_str += file;
198     error_str += "\tline ";
199     error_str += std::to_string(line);
200     error_str += ":\t\n";
201     error_str += std::to_string(x);
202     error_str += " is not greater than ";
203     error_str += std::to_string(y);
204     error_str += "\n";
205
206     #ifdef _WIN32
207         std::cout << error_str << std::endl;
208     #endif
209

```

```

210     throw std::runtime_error(error_str);
211     return;
212 } /* testGreaterThan() */

```

5.46.3.7 testGreaterThanOrEqualTo()

```

void testGreaterThanOrEqualTo (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x \geq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

242 {
243     if (x >= y) {
244         return;
245     }
246
247     std::string error_str = "ERROR: testGreaterThanOrEqualTo():\t in ";
248     error_str += file;
249     error_str += "\tline ";
250     error_str += std::to_string(line);
251     error_str += ":\t\n";
252     error_str += std::to_string(x);
253     error_str += " is not greater than or equal to ";
254     error_str += std::to_string(y);
255     error_str += "\n";
256
257     #ifdef _WIN32
258         std::cout << error_str << std::endl;
259     #endif
260
261     throw std::runtime_error(error_str);
262     return;
263 } /* testGreaterThanOrEqualTo() */

```

5.46.3.8 testLessThan()

```

void testLessThan (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x < y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

293 {
294     if (x < y) {
295         return;
296     }
297
298     std::string error_str = "ERROR: testLessThan():\t in ";
299     error_str += file;
300     error_str += "\tline ";
301     error_str += std::to_string(line);
302     error_str += ":\t\n";
303     error_str += std::to_string(x);
304     error_str += " is not less than ";
305     error_str += std::to_string(y);
306     error_str += "\n";
307
308     #ifdef _WIN32
309         std::cout << error_str << std::endl;
310     #endif
311
312     throw std::runtime_error(error_str);
313     return;
314 } /* testLessThan() */

```

5.46.3.9 testLessThanOrEqualTo()

```

void testLessThanOrEqualTo (
    double x,
    double y,
    std::string file,
    int line )

```

Tests if $x \leq y$.

Parameters

<i>x</i>	The first of two numbers to test.
<i>y</i>	The second of two numbers to test.
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```

344 {
345     if (x <= y) {
346         return;
347     }
348
349     std::string error_str = "ERROR: testLessThanOrEqualTo():\t in ";
350     error_str += file;
351     error_str += "\tline ";
352     error_str += std::to_string(line);
353     error_str += ":\t\n";
354     error_str += std::to_string(x);
355     error_str += " is not less than or equal to ";
356     error_str += std::to_string(y);
357     error_str += "\n";
358
359     #ifdef _WIN32
360         std::cout << error_str << std::endl;
361     #endif
362
363     throw std::runtime_error(error_str);
364     return;
365 } /* testLessThanOrEqualTo() */

```

5.46.3.10 testTruth()

```

void testTruth (

```

```
bool statement,  
std::string file,  
int line )
```

Tests if the given statement is true.

Parameters

<i>statement</i>	The statement whose truth is to be tested ("1 == 0", for example).
<i>file</i>	The file in which the test is applied (you should be able to just pass in "__FILE__").
<i>line</i>	The line of the file in which the test is applied (you should be able to just pass in "__LINE__").

```
392 {  
393     if (statement) {  
394         return;  
395     }  
396  
397     std::string error_str = "ERROR: testTruth():\t in ";  
398     error_str += file;  
399     error_str += "\tline ";  
400     error_str += std::to_string(line);  
401     error_str += ":\t\n";  
402     error_str += "Given statement is not true";  
403  
404     #ifdef _WIN32  
405         std::cout << error_str << std::endl;  
406     #endif  
407  
408     throw std::runtime_error(error_str);  
409     return;  
410 } /* testTruth() */
```


Bibliography

- Dr. B. Buckham, Dr. C. Crawford, Dr. I. Beya Marshall, and Dr. B. Whitby. Wei Wai Kum Tidal Prefeasibility Study - Tidal Resource Assessment. Technical report, PRIMED, 2023. Internal: P2202E_BRKLYG+WEI WAI KUM_R01_V20230613v3. [110](#)
- HOMER. Capital Recovery Factor, 2023a. URL https://www.homerenergy.com/products/pro/docs/latest/capital_recovery_factor.html. [73](#)
- HOMER. Discount Factor, 2023b. URL https://www.homerenergy.com/products/pro/docs/latest/discount_factor.html. [71](#), [73](#)
- HOMER. Fuel Curve, 2023c. URL https://www.homerenergy.com/products/pro/docs/latest/fuel_curve.html. [38](#), [43](#)
- HOMER. Generator Fuel Curve Intercept Coefficient, 2023d. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_intercept_coefficient.html. [38](#), [43](#)
- HOMER. Generator Fuel Curve Slope, 2023e. URL https://www.homerenergy.com/products/pro/docs/latest/generator_fuel_curve_slope.html. [38](#), [43](#)
- 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. [101](#)
- HOMER. Levelized Cost of Energy, 2023g. URL https://www.homerenergy.com/products/pro/docs/latest/levelized_cost_of_energy.html. [73](#)
- HOMER. Real Discount Rate, 2023h. URL https://www.homerenergy.com/products/pro/docs/latest/real_discount_rate.html. [71](#)
- HOMER. Total Annualized Cost, 2023i. URL https://www.homerenergy.com/products/pro/docs/latest/total_annualized_cost.html. [73](#)
- 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. [112](#), [123](#), [124](#)
- 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. [122](#)

Index

- __applyCycleChargingControl_CHARGING
Controller, [20](#)
- __applyCycleChargingControl_DISCHARGING
Controller, [21](#)
- __applyLoadFollowingControl_CHARGING
Controller, [22](#)
- __applyLoadFollowingControl_DISCHARGING
Controller, [23](#)
- __checkInputs
Combustion, [11](#)
Diesel, [36](#)
Model, [58](#)
Production, [70](#)
Renewable, [82](#)
Solar, [99](#)
Tidal, [109](#)
Wave, [120](#)
Wind, [132](#)
- __checkResourceKey1D
Resources, [88](#)
- __checkResourceKey2D
Resources, [89](#)
- __checkTimePoint
Resources, [89](#)
- __computeCubicProductionkW
Tidal, [109](#)
- __computeEconomics
Model, [58](#)
- __computeExponentialProductionkW
Tidal, [110](#)
Wind, [132](#)
- __computeFuelAndEmissions
Model, [58](#)
- __computeGaussianProductionkW
Wave, [120](#)
- __computeLevellizedCostOfEnergy
Model, [59](#)
- __computeLookupProductionkW
Tidal, [111](#)
Wave, [121](#)
Wind, [133](#)
- __computeNetLoad
Controller, [24](#)
- __computeNetPresentCost
Model, [59](#)
- __computeParaboloidProductionkW
Wave, [121](#)
- __computeRealDiscountAnnual
Production, [71](#)
- __constructCombustionMap
Controller, [25](#)
- __getGenericCapitalCost
Diesel, [37](#)
Solar, [100](#)
Tidal, [111](#)
Wave, [123](#)
Wind, [134](#)
- __getGenericFuelIntercept
Diesel, [38](#)
- __getGenericFuelSlope
Diesel, [38](#)
- __getGenericOpMaintCost
Diesel, [38](#)
Solar, [100](#)
Tidal, [112](#)
Wave, [123](#)
Wind, [134](#)
- __getRenewableProduction
Controller, [26](#)
- __handleCombustionDispatch
Controller, [27](#)
- __handleReplacement
Production, [72](#)
- __handleStartStop
Diesel, [39](#)
Renewable, [83](#)
- __handleStorageCharging
Controller, [28, 29](#)
- __handleStorageDischarging
Controller, [29](#)
- __readSolarResource
Resources, [90](#)
- __readTidalResource
Resources, [91](#)
- __readWaveResource
Resources, [92](#)
- __readWindResource
Resources, [93](#)
- __throwLengthError
Resources, [94](#)
- ~Combustion
Combustion, [10](#)
- ~Controller
Controller, [20](#)
- ~Diesel
Diesel, [35](#)
- ~ElectricalLoad
ElectricalLoad, [48](#)

- ~Lilon
 - Lilon, [54](#)
- ~Model
 - Model, [57](#)
- ~Production
 - Production, [70](#)
- ~Renewable
 - Renewable, [82](#)
- ~Resources
 - Resources, [87](#)
- ~Solar
 - Solar, [99](#)
- ~Storage
 - Storage, [105](#)
- ~Tidal
 - Tidal, [109](#)
- ~Wave
 - Wave, [119](#)
- ~Wind
 - Wind, [132](#)
- addDiesel
 - Model, [60](#)
- addResource
 - Model, [60](#)
 - Resources, [94](#)
- addSolar
 - Model, [61](#)
- addTidal
 - Model, [61](#)
- addWave
 - Model, [62](#)
- addWind
 - Model, [62](#)
- applyDispatchControl
 - Controller, [30](#)
- capacity_kW
 - Production, [74](#)
 - ProductionInputs, [79](#)
- capital_cost
 - DieselInputs, [43](#)
 - Production, [74](#)
 - SolarInputs, [103](#)
 - TidalInputs, [116](#)
 - WaveInputs, [128](#)
 - WindInputs, [138](#)
- capital_cost_vec
 - Production, [74](#)
- CH4_emissions_intensity_kgL
 - Combustion, [14](#)
 - DieselInputs, [44](#)
- CH4_emissions_vec_kg
 - Combustion, [14](#)
- CH4_kg
 - Emissions, [52](#)
- clear
 - Controller, [31](#)
 - ElectricalLoad, [48](#)
 - Model, [62](#)
 - Resources, [95](#)
- CO2_emissions_intensity_kgL
 - Combustion, [14](#)
 - DieselInputs, [44](#)
- CO2_emissions_vec_kg
 - Combustion, [15](#)
- CO2_kg
 - Emissions, [52](#)
- CO_emissions_intensity_kgL
 - Combustion, [15](#)
 - DieselInputs, [44](#)
- CO_emissions_vec_kg
 - Combustion, [15](#)
- CO_kg
 - Emissions, [52](#)
- Combustion, [7](#)
 - __checkInputs, [11](#)
 - ~Combustion, [10](#)
 - CH4_emissions_intensity_kgL, [14](#)
 - CH4_emissions_vec_kg, [14](#)
 - CO2_emissions_intensity_kgL, [14](#)
 - CO2_emissions_vec_kg, [15](#)
 - CO_emissions_intensity_kgL, [15](#)
 - CO_emissions_vec_kg, [15](#)
 - Combustion, [9, 10](#)
 - commit, [11](#)
 - computeEconomics, [12](#)
 - computeFuelAndEmissions, [13](#)
 - fuel_consumption_vec_L, [15](#)
 - fuel_cost_L, [15](#)
 - fuel_cost_vec, [15](#)
 - getEmissionskg, [13](#)
 - getFuelConsumptionL, [13](#)
 - linear_fuel_intercept_LkWh, [16](#)
 - linear_fuel_slope_LkWh, [16](#)
 - NOx_emissions_intensity_kgL, [16](#)
 - NOx_emissions_vec_kg, [16](#)
 - PM_emissions_intensity_kgL, [16](#)
 - PM_emissions_vec_kg, [16](#)
 - requestProductionkW, [14](#)
 - SOx_emissions_intensity_kgL, [17](#)
 - SOx_emissions_vec_kg, [17](#)
 - total_emissions, [17](#)
 - total_fuel_consumed_L, [17](#)
 - type, [17](#)
- Combustion.h
 - CombustionType, [143](#)
 - DIESEL, [144](#)
 - N_COMBUSTION_TYPES, [144](#)
- combustion_inputs
 - DieselInputs, [44](#)
- combustion_map
 - Controller, [32](#)
- combustion_ptr_vec
 - Model, [64](#)
- CombustionInputs, [18](#)
 - production_inputs, [18](#)

- CombustionType
 - Combustion.h, [143](#)
- commit
 - Combustion, [11](#)
 - Diesel, [40](#)
 - Production, [72](#)
 - Renewable, [83](#)
 - Solar, [100](#)
 - Tidal, [112](#)
 - Wave, [124](#)
 - Wind, [134](#)
- computeEconomics
 - Combustion, [12](#)
 - Production, [73](#)
 - Renewable, [84](#)
- computeFuelAndEmissions
 - Combustion, [13](#)
- computeProductionkW
 - Renewable, [84](#)
 - Solar, [101](#)
 - Tidal, [113](#)
 - Wave, [125](#)
 - Wind, [135](#)
- control_mode
 - Controller, [32](#)
 - ModelInputs, [66](#)
- Controller, [19](#)
 - __applyCycleChargingControl_CHARGING, [20](#)
 - __applyCycleChargingControl_DISCHARGING, [21](#)
 - __applyLoadFollowingControl_CHARGING, [22](#)
 - __applyLoadFollowingControl_DISCHARGING, [23](#)
 - __computeNetLoad, [24](#)
 - __constructCombustionMap, [25](#)
 - __getRenewableProduction, [26](#)
 - __handleCombustionDispatch, [27](#)
 - __handleStorageCharging, [28, 29](#)
 - __handleStorageDischarging, [29](#)
 - ~Controller, [20](#)
 - applyDispatchControl, [30](#)
 - clear, [31](#)
 - combustion_map, [32](#)
 - control_mode, [32](#)
 - Controller, [20](#)
 - init, [31](#)
 - missed_load_vec_kW, [32](#)
 - net_load_vec_kW, [32](#)
- controller
 - Model, [64](#)
- Controller.h
 - ControlMode, [140](#)
 - CYCLE_CHARGING, [140](#)
 - LOAD_FOLLOWING, [140](#)
 - N_CONTROL_MODES, [140](#)
- ControlMode
 - Controller.h, [140](#)
- curtailment_vec_kW
 - Production, [74](#)
- CYCLE_CHARGING
 - Controller.h, [140](#)
- derating
 - Solar, [102](#)
 - SolarInputs, [104](#)
- design_energy_period_s
 - Wave, [126](#)
 - WaveInputs, [128](#)
- design_significant_wave_height_m
 - Wave, [126](#)
 - WaveInputs, [128](#)
- design_speed_ms
 - Tidal, [114](#)
 - TidalInputs, [116](#)
 - Wind, [136](#)
 - WindInputs, [138](#)
- DIESEL
 - Combustion.h, [144](#)
- Diesel, [33](#)
 - __checkInputs, [36](#)
 - __getGenericCapitalCost, [37](#)
 - __getGenericFuelIntercept, [38](#)
 - __getGenericFuelSlope, [38](#)
 - __getGenericOpMaintCost, [38](#)
 - __handleStartStop, [39](#)
 - ~Diesel, [35](#)
 - commit, [40](#)
 - Diesel, [34, 35](#)
 - minimum_load_ratio, [41](#)
 - minimum_runtime_hrs, [41](#)
 - requestProductionkW, [40](#)
 - time_since_last_start_hrs, [41](#)
- DieselInputs, [42](#)
 - capital_cost, [43](#)
 - CH4_emissions_intensity_kgL, [44](#)
 - CO2_emissions_intensity_kgL, [44](#)
 - CO_emissions_intensity_kgL, [44](#)
 - combustion_inputs, [44](#)
 - fuel_cost_L, [44](#)
 - linear_fuel_intercept_LkWh, [44](#)
 - linear_fuel_slope_LkWh, [45](#)
 - minimum_load_ratio, [45](#)
 - minimum_runtime_hrs, [45](#)
 - NOx_emissions_intensity_kgL, [45](#)
 - operation_maintenance_cost_kWh, [45](#)
 - PM_emissions_intensity_kgL, [46](#)
 - replace_running_hrs, [46](#)
 - SOx_emissions_intensity_kgL, [46](#)
- dispatch_vec_kW
 - Production, [75](#)
- dt_vec_hrs
 - ElectricalLoad, [50](#)
- electrical_load
 - Model, [64](#)
- ElectricalLoad, [46](#)
 - ~ElectricalLoad, [48](#)
 - clear, [48](#)
 - dt_vec_hrs, [50](#)

- ElectricalLoad, [47](#), [48](#)
- load_vec_kW, [50](#)
- max_load_kW, [50](#)
- mean_load_kW, [50](#)
- min_load_kW, [50](#)
- n_points, [51](#)
- n_years, [51](#)
- path_2_electrical_load_time_series, [51](#)
- readLoadData, [49](#)
- time_vec_hrs, [51](#)
- Emissions, [51](#)
 - CH4_kg, [52](#)
 - CO2_kg, [52](#)
 - CO_kg, [52](#)
 - NOx_kg, [52](#)
 - PM_kg, [53](#)
 - SOx_kg, [53](#)
- expectedErrorNotDetected
 - testing_utils.cpp, [211](#)
 - testing_utils.h, [218](#)
- FLOAT_TOLERANCE
 - testing_utils.h, [218](#)
- fuel_consumption_vec_L
 - Combustion, [15](#)
- fuel_cost_L
 - Combustion, [15](#)
 - DieselInputs, [44](#)
- fuel_cost_vec
 - Combustion, [15](#)
- getEmissionskg
 - Combustion, [13](#)
- getFuelConsumptionL
 - Combustion, [13](#)
- header/Controller.h, [139](#)
- header/ElectricalLoad.h, [140](#)
- header/Model.h, [141](#)
- header/Production/Combustion/Combustion.h, [142](#)
- header/Production/Combustion/Diesel.h, [144](#)
- header/Production/Production.h, [145](#)
- header/Production/Renewable/Renewable.h, [146](#)
- header/Production/Renewable/Solar.h, [147](#)
- header/Production/Renewable/Tidal.h, [148](#)
- header/Production/Renewable/Wave.h, [149](#)
- header/Production/Renewable/Wind.h, [151](#)
- header/Resources.h, [152](#)
- header/std_includes.h, [153](#)
- header/Storage/Lilon.h, [154](#)
- header/Storage/Storage.h, [155](#)
- init
 - Controller, [31](#)
- is_running
 - Production, [75](#)
- is_running_vec
 - Production, [75](#)
- is_sunk
 - Production, [75](#)
 - ProductionInputs, [79](#)
- levellized_cost_of_energy_kWh
 - Model, [64](#)
 - Production, [75](#)
- Lilon, [53](#)
 - ~Lilon, [54](#)
 - Lilon, [54](#)
- linear_fuel_intercept_LkWh
 - Combustion, [16](#)
 - DieselInputs, [44](#)
- linear_fuel_slope_LkWh
 - Combustion, [16](#)
 - DieselInputs, [45](#)
- LOAD_FOLLOWING
 - Controller.h, [140](#)
- load_vec_kW
 - ElectricalLoad, [50](#)
- main
 - test_Combustion.cpp, [165](#)
 - test_Controller.cpp, [193](#)
 - test_Diesel.cpp, [167](#)
 - test_ElectricalLoad.cpp, [195](#)
 - test_Lilon.cpp, [191](#)
 - test_Model.cpp, [197](#)
 - test_Production.cpp, [188](#)
 - test_Renewable.cpp, [172](#)
 - test_Resources.cpp, [204](#)
 - test_Solar.cpp, [174](#)
 - test_Storage.cpp, [192](#)
 - test_Tidal.cpp, [177](#)
 - test_Wave.cpp, [181](#)
 - test_Wind.cpp, [185](#)
- max_load_kW
 - ElectricalLoad, [50](#)
- mean_load_kW
 - ElectricalLoad, [50](#)
- min_load_kW
 - ElectricalLoad, [50](#)
- minimum_load_ratio
 - Diesel, [41](#)
 - DieselInputs, [45](#)
- minimum_runtime_hrs
 - Diesel, [41](#)
 - DieselInputs, [45](#)
- missed_load_vec_kW
 - Controller, [32](#)
- Model, [55](#)
 - __checkInputs, [58](#)
 - __computeEconomics, [58](#)
 - __computeFuelAndEmissions, [58](#)
 - __computeLevellizedCostOfEnergy, [59](#)
 - __computeNetPresentCost, [59](#)
 - ~Model, [57](#)
 - addDiesel, [60](#)
 - addResource, [60](#)
 - addSolar, [61](#)

- addTidal, 61
- addWave, 62
- addWind, 62
- clear, 62
- combustion_ptr_vec, 64
- controller, 64
- electrical_load, 64
- levellized_cost_of_energy_kWh, 64
- Model, 57
- net_present_cost, 64
- renewable_ptr_vec, 65
- reset, 63
- resources, 65
- run, 63
- storage_ptr_vec, 65
- total_dispatch_discharge_kWh, 65
- total_emissions, 65
- total_fuel_consumed_L, 65
- ModelInputs, 66
 - control_mode, 66
 - path_2_electrical_load_time_series, 66
- N_COMBUSTION_TYPES
 - Combustion.h, 144
- N_CONTROL_MODES
 - Controller.h, 140
- n_points
 - ElectricalLoad, 51
 - Production, 75
- N_RENEWABLE_TYPES
 - Renewable.h, 147
- n_replacements
 - Production, 76
- n_starts
 - Production, 76
- N_TIDAL_POWER_PRODUCTION_MODELS
 - Tidal.h, 149
- N_WAVE_POWER_PRODUCTION_MODELS
 - Wave.h, 150
- N_WIND_POWER_PRODUCTION_MODELS
 - Wind.h, 152
- n_years
 - ElectricalLoad, 51
- net_load_vec_kW
 - Controller, 32
- net_present_cost
 - Model, 64
 - Production, 76
- nominal_discount_annual
 - ProductionInputs, 79
- nominal_inflation_annual
 - ProductionInputs, 79
- NOx_emissions_intensity_kgL
 - Combustion, 16
 - DieselInputs, 45
- NOx_emissions_vec_kg
 - Combustion, 16
- NOx_kg
 - Emissions, 52
- operation_maintenance_cost_kWh
 - DieselInputs, 45
 - Production, 76
 - SolarInputs, 104
 - TidalInputs, 116
 - WaveInputs, 128
 - WindInputs, 138
- operation_maintenance_cost_vec
 - Production, 76
- path_2_electrical_load_time_series
 - ElectricalLoad, 51
 - ModelInputs, 66
- path_map_1D
 - Resources, 96
- path_map_2D
 - Resources, 96
- PM_emissions_intensity_kgL
 - Combustion, 16
 - DieselInputs, 46
- PM_emissions_vec_kg
 - Combustion, 16
- PM_kg
 - Emissions, 53
- power_model
 - Tidal, 114
 - TidalInputs, 116
 - Wave, 126
 - WaveInputs, 128
 - Wind, 136
 - WindInputs, 138
- print_flag
 - Production, 76
 - ProductionInputs, 79
- printGold
 - testing_utils.cpp, 212
 - testing_utils.h, 218
- printGreen
 - testing_utils.cpp, 212
 - testing_utils.h, 219
- printRed
 - testing_utils.cpp, 212
 - testing_utils.h, 219
- Production, 67
 - __checkInputs, 70
 - __computeRealDiscountAnnual, 71
 - __handleReplacement, 72
 - ~Production, 70
 - capacity_kW, 74
 - capital_cost, 74
 - capital_cost_vec, 74
 - commit, 72
 - computeEconomics, 73
 - curtailment_vec_kW, 74
 - dispatch_vec_kW, 75
 - is_running, 75
 - is_running_vec, 75
 - is_sunk, 75
 - levellized_cost_of_energy_kWh, 75

- n_points, 75
- n_replacements, 76
- n_starts, 76
- net_present_cost, 76
- operation_maintenance_cost_kWh, 76
- operation_maintenance_cost_vec, 76
- print_flag, 76
- Production, 69
- production_vec_kW, 77
- real_discount_annual, 77
- replace_running_hrs, 77
- running_hours, 77
- storage_vec_kW, 77
- total_dispatch_kWh, 77
- type_str, 78
- production_inputs
 - CombustionInputs, 18
 - RenewableInputs, 86
- production_vec_kW
 - Production, 77
- ProductionInputs, 78
 - capacity_kW, 79
 - is_sunk, 79
 - nominal_discount_annual, 79
 - nominal_inflation_annual, 79
 - print_flag, 79
 - replace_running_hrs, 79
- PYBIND11_MODULE
 - PYBIND11_PGM.cpp, 156
- PYBIND11_PGM.cpp
 - PYBIND11_MODULE, 156
- pybindings/PYBIND11_PGM.cpp, 155
- readLoadData
 - ElectricalLoad, 49
- real_discount_annual
 - Production, 77
- Renewable, 80
 - __checkInputs, 82
 - __handleStartStop, 83
 - ~Renewable, 82
 - commit, 83
 - computeEconomics, 84
 - computeProductionkW, 84
 - Renewable, 81, 82
 - resource_key, 85
 - type, 85
- Renewable.h
 - N_RENEWABLE_TYPES, 147
 - RenewableType, 146
 - SOLAR, 147
 - TIDAL, 147
 - WAVE, 147
 - WIND, 147
- renewable_inputs
 - SolarInputs, 104
 - TidalInputs, 116
 - WaveInputs, 129
 - WindInputs, 138
- renewable_ptr_vec
 - Model, 65
- RenewableInputs, 85
 - production_inputs, 86
- RenewableType
 - Renewable.h, 146
- replace_running_hrs
 - DieselInputs, 46
 - Production, 77
 - ProductionInputs, 79
- requestProductionkW
 - Combustion, 14
 - Diesel, 40
- reset
 - Model, 63
- resource_key
 - Renewable, 85
 - SolarInputs, 104
 - TidalInputs, 116
 - WaveInputs, 129
 - WindInputs, 138
- resource_map_1D
 - Resources, 96
- resource_map_2D
 - Resources, 96
- Resources, 86
 - __checkResourceKey1D, 88
 - __checkResourceKey2D, 89
 - __checkTimePoint, 89
 - __readSolarResource, 90
 - __readTidalResource, 91
 - __readWaveResource, 92
 - __readWindResource, 93
 - __throwLengthError, 94
 - ~Resources, 87
 - addResource, 94
 - clear, 95
 - path_map_1D, 96
 - path_map_2D, 96
 - resource_map_1D, 96
 - resource_map_2D, 96
 - Resources, 87
- resources
 - Model, 65
- run
 - Model, 63
- running_hours
 - Production, 77
- SOLAR
 - Renewable.h, 147
- Solar, 97
 - __checkInputs, 99
 - __getGenericCapitalCost, 100
 - __getGenericOpMaintCost, 100
 - ~Solar, 99
 - commit, 100
 - computeProductionkW, 101
 - derating, 102

- Solar, [98](#), [99](#)
- SolarInputs, [102](#)
 - capital_cost, [103](#)
 - derating, [104](#)
 - operation_maintenance_cost_kWh, [104](#)
 - renewable_inputs, [104](#)
 - resource_key, [104](#)
- source/Controller.cpp, [157](#)
- source/ElectricalLoad.cpp, [158](#)
- source/Model.cpp, [158](#)
- source/Production/Combustion/Combustion.cpp, [159](#)
- source/Production/Combustion/Diesel.cpp, [159](#)
- source/Production/Production.cpp, [160](#)
- source/Production/Renewable/Renewable.cpp, [160](#)
- source/Production/Renewable/Solar.cpp, [161](#)
- source/Production/Renewable/Tidal.cpp, [161](#)
- source/Production/Renewable/Wave.cpp, [162](#)
- source/Production/Renewable/Wind.cpp, [162](#)
- source/Resources.cpp, [163](#)
- source/Storage/Lilon.cpp, [163](#)
- source/Storage/Storage.cpp, [164](#)
- SOx_emissions_intensity_kgL
 - Combustion, [17](#)
 - DieselInputs, [46](#)
- SOx_emissions_vec_kg
 - Combustion, [17](#)
- SOx_kg
 - Emissions, [53](#)
- Storage, [105](#)
 - ~Storage, [105](#)
 - Storage, [105](#)
- storage_ptr_vec
 - Model, [65](#)
- storage_vec_kW
 - Production, [77](#)
- test/source/Production/Combustion/test_Combustion.cpp, [164](#)
- test/source/Production/Combustion/test_Diesel.cpp, [166](#)
- test/source/Production/Renewable/test_Renewable.cpp, [172](#)
- test/source/Production/Renewable/test_Solar.cpp, [173](#)
- test/source/Production/Renewable/test_Tidal.cpp, [177](#)
- test/source/Production/Renewable/test_Wave.cpp, [180](#)
- test/source/Production/Renewable/test_Wind.cpp, [184](#)
- test/source/Production/test_Production.cpp, [188](#)
- test/source/Storage/test_Lilon.cpp, [190](#)
- test/source/Storage/test_Storage.cpp, [192](#)
- test/source/test_Controller.cpp, [193](#)
- test/source/test_ElectricalLoad.cpp, [194](#)
- test/source/test_Model.cpp, [197](#)
- test/source/test_Resources.cpp, [204](#)
- test/utls/testing_utils.cpp, [210](#)
- test/utls/testing_utils.h, [216](#)
- test_Combustion.cpp
 - main, [165](#)
- test_Controller.cpp
 - main, [193](#)
- test_Diesel.cpp
 - main, [167](#)
- test_ElectricalLoad.cpp
 - main, [195](#)
- test_Lilon.cpp
 - main, [191](#)
- test_Model.cpp
 - main, [197](#)
- test_Production.cpp
 - main, [188](#)
- test_Renewable.cpp
 - main, [172](#)
- test_Resources.cpp
 - main, [204](#)
- test_Solar.cpp
 - main, [174](#)
- test_Storage.cpp
 - main, [192](#)
- test_Tidal.cpp
 - main, [177](#)
- test_Wave.cpp
 - main, [181](#)
- test_Wind.cpp
 - main, [185](#)
- testFloatEquals
 - testing_utils.cpp, [213](#)
 - testing_utils.h, [219](#)
- testGreaterThan
 - testing_utils.cpp, [213](#)
 - testing_utils.h, [220](#)
- testGreaterThanOrEqualTo
 - testing_utils.cpp, [214](#)
 - testing_utils.h, [221](#)
- testing_utils.cpp
 - expectedErrorNotDetected, [211](#)
 - printGold, [212](#)
 - printGreen, [212](#)
 - printRed, [212](#)
 - testFloatEquals, [213](#)
 - testGreaterThan, [213](#)
 - testGreaterThanOrEqualTo, [214](#)
 - testLessThan, [215](#)
 - testLessThanOrEqualTo, [215](#)
 - testTruth, [216](#)
- testing_utils.h
 - expectedErrorNotDetected, [218](#)
 - FLOAT_TOLERANCE, [218](#)
 - printGold, [218](#)
 - printGreen, [219](#)
 - printRed, [219](#)
 - testFloatEquals, [219](#)
 - testGreaterThan, [220](#)
 - testGreaterThanOrEqualTo, [221](#)
 - testLessThan, [221](#)
 - testLessThanOrEqualTo, [222](#)
 - testTruth, [222](#)
- testLessThan
 - testing_utils.cpp, [215](#)

- testing_utils.h, 221
- testLessThanOrEqualTo
 - testing_utils.cpp, 215
 - testing_utils.h, 222
- testTruth
 - testing_utils.cpp, 216
 - testing_utils.h, 222
- TIDAL
 - Renewable.h, 147
- Tidal, 106
 - __checkInputs, 109
 - __computeCubicProductionkW, 109
 - __computeExponentialProductionkW, 110
 - __computeLookupProductionkW, 111
 - __getGenericCapitalCost, 111
 - __getGenericOpMaintCost, 112
 - ~Tidal, 109
 - commit, 112
 - computeProductionkW, 113
 - design_speed_ms, 114
 - power_model, 114
 - Tidal, 108
- Tidal.h
 - N_TIDAL_POWER_PRODUCTION_MODELS, 149
 - TIDAL_POWER_CUBIC, 149
 - TIDAL_POWER_EXPONENTIAL, 149
 - TIDAL_POWER_LOOKUP, 149
 - TidalPowerProductionModel, 149
- TIDAL_POWER_CUBIC
 - Tidal.h, 149
- TIDAL_POWER_EXPONENTIAL
 - Tidal.h, 149
- TIDAL_POWER_LOOKUP
 - Tidal.h, 149
- TidalInputs, 115
 - capital_cost, 116
 - design_speed_ms, 116
 - operation_maintenance_cost_kWh, 116
 - power_model, 116
 - renewable_inputs, 116
 - resource_key, 116
- TidalPowerProductionModel
 - Tidal.h, 149
- time_since_last_start_hrs
 - Diesel, 41
- time_vec_hrs
 - ElectricalLoad, 51
- total_dispatch_discharge_kWh
 - Model, 65
- total_dispatch_kWh
 - Production, 77
- total_emissions
 - Combustion, 17
 - Model, 65
- total_fuel_consumed_L
 - Combustion, 17
 - Model, 65
- type
 - Combustion, 17
 - Renewable, 85
- type_str
 - Production, 78
- WAVE
 - Renewable.h, 147
- Wave, 117
 - __checkInputs, 120
 - __computeGaussianProductionkW, 120
 - __computeLookupProductionkW, 121
 - __computeParaboloidProductionkW, 121
 - __getGenericCapitalCost, 123
 - __getGenericOpMaintCost, 123
 - ~Wave, 119
 - commit, 124
 - computeProductionkW, 125
 - design_energy_period_s, 126
 - design_significant_wave_height_m, 126
 - power_model, 126
 - Wave, 118, 119
- Wave.h
 - N_WAVE_POWER_PRODUCTION_MODELS, 150
 - WAVE_POWER_GAUSSIAN, 150
 - WAVE_POWER_LOOKUP, 150
 - WAVE_POWER_PARABOLOID, 150
 - WavePowerProductionModel, 150
- WAVE_POWER_GAUSSIAN
 - Wave.h, 150
- WAVE_POWER_LOOKUP
 - Wave.h, 150
- WAVE_POWER_PARABOLOID
 - Wave.h, 150
- WaveInputs, 127
 - capital_cost, 128
 - design_energy_period_s, 128
 - design_significant_wave_height_m, 128
 - operation_maintenance_cost_kWh, 128
 - power_model, 128
 - renewable_inputs, 129
 - resource_key, 129
- WavePowerProductionModel
 - Wave.h, 150
- WIND
 - Renewable.h, 147
- Wind, 129
 - __checkInputs, 132
 - __computeExponentialProductionkW, 132
 - __computeLookupProductionkW, 133
 - __getGenericCapitalCost, 134
 - __getGenericOpMaintCost, 134
 - ~Wind, 132
 - commit, 134
 - computeProductionkW, 135
 - design_speed_ms, 136
 - power_model, 136
 - Wind, 131

Wind.h
 N_WIND_POWER_PRODUCTION_MODELS, [152](#)
 WIND_POWER_EXPONENTIAL, [152](#)
 WIND_POWER_LOOKUP, [152](#)
 WindPowerProductionModel, [152](#)
WIND_POWER_EXPONENTIAL
 Wind.h, [152](#)
WIND_POWER_LOOKUP
 Wind.h, [152](#)
WindInputs, [137](#)
 capital_cost, [138](#)
 design_speed_ms, [138](#)
 operation_maintenance_cost_kWh, [138](#)
 power_model, [138](#)
 renewable_inputs, [138](#)
 resource_key, [138](#)
WindPowerProductionModel
 Wind.h, [152](#)