<u>SCORES: User Guide</u>

Constance Crozier ~ August 2020

This document contains documentation and example code designed to accompany the SCORES repository.

0 Getting Started

The following packages are necessary prerequisites: numpy, matplotlib, csv, copy, datetime, scipy.optimise, pyDOE, os, mpl_toolkits.basemap (for maps only).

The code in the repository does not contain the raw data necessary to run the model, but it does contain saved results from the UK which allow some analysis to be done. In order to get the full use of the model, or to study systems outside of the UK, hourly weather observations need to be added to the data folder.

**something about how to get the NASA data

0.1 AggEV and Optimisation Module Update, MAY 2022

The model was significantly expanded in May 2022 to include the modelling of EVs and the ability to optimise system operation and sizing via a linear programme described in the LinearProgramFormulation.pdf provided alongside this USERGUIDE. Here is a summary of the main additions within each of these two categories, and the necessary pre-requisites to use them.

Modelling EVs

EVs are modelled by aggregating EVs with similar driving patterns into fleets via the AggregatedEVModel class. These fleet objects are then collected together into a MultipleAggregatedEVs class in an analogous way to storage. Each fleet is split into 2 virtual batteries, one for those with V2G chargers and one for those with Unidirectional chargers. The virtual batteries parameters are time varying depending on the number of EVs plugged in, which is a pattern specified by the user.

Aggregated EV objects can be simulated in operation causally via ordered charging and discharge using the new *MultipleStorageAssets*.**causal_system_operation()** method. This method assumes no forecast of EV plugin behaviour or renewable output. This can be compared to the optimal operation given perfect foresight of both these things using the *MultipleStorageAssets*.**non_causal_system_operation().** The aggregated EV objects are fully compatible with the Lin_Prog_Model() class also.

MultipleStorageAssets.**non_causal_system_operation()** uses Lin_Prog_Model class within it, thus the user needs the pyomo package to be working to use it. More on this below.

Optimisation Module

This module focusses on the Lin_Prog_Model class. The user can specify different types of storage amd renewable generation available to be built, then this module will output the cost optimal combination in order to meet demand with a user specified percentage of demand. The user can also specify different EV fleets present on the system, from which the optimiser will optimally choose how many V2G chargers and how many unidirectional chargers to build for each fleet. Co-optimising this alongside storage and generation is essential to fully explore the marginal value of V2G over Unidirectional chargers.

The optimal sizing problem is formed as a continuous linear programme described in LinearProgramFormulation.pdf. This model is formed at a high level using pyomo, which then feeds the problem to a low – level solver (e.g. MOSEK, Gurobi…) to perform the low level algorithmic solving. The class also contains various methods for visualising and outputting the results of the optimisation.

Pyomo

Pyomo is required to run the optimisation module. It can be downloaded using pip or conda. It needs to be linked to a solver to perform the low level optimisations. Pyomo supports most solvers that can solve continuous linear program, a list of the free or commercial options is available here: https://yalmip.github.io/allsolvers/

An extensive list of the pyomo compatible solvers can be found in their documentation listed below. The SCORES Github contains a simple pyomo example called Pyomo_Example.py to help the user understand pyomo and to verify that their installation is working correctly. Extensive pyomo help is available online, some good resources are:

http://www.pyomo.org/installation

https://www.osti.gov/servlets/purl/1376827

https://pyomo.readthedocs.io/en/stable/solving_pyomo_models.html#supported-solvers

The user will have to change the line (approx. 557) in opt_con_class.py to reflect their chosen solver. Here I have used mosek:

```
opt = pyo.SolverFactory('mosek')
```

1 Generation Models

Each generation model object describes a form of power generation, which has an associated hourly power output.

1.1 Classes

1.1.1 Base Class

GenerationModel(sites, year_min, year_max, months, fixed_cost, variable_cost, name, data_path, save_path)

Note that there are two options for initialising a generation mode: in select cases you can load a previous run of the model (providing it has been stored in the save path), otherwise you will need to run the model using raw weather data. For this reason data_path is technically an optional parameter, but if the simulation has not been previously stored, then it is required.

Parameter	Type	Description
sites	*Array-like*	*List of chosen site numbers, or the string 'all'*
year_min	*int*	*Lowest year to be included in the simulation*
year_max	*int*	*Highest year to be included in the simulation*
months	*Array-like*	*List of months to be included in the simulation (1-12)*
fixed_cost	*float*	*Cost incurred per MW-year of installation in GBP*
variable_cost	*int*	*Cost incurred per MWh of generation in GBP*
name	*str*	*Name of generator - used for graph plotting*
data_path	*str*	*Path to folder where the weather data is stored*
save_path	*str*	*Path to folder where model output will be stored if desired*

1.1.2 Offshore Wind

OffshoreWindModel(sites='all', year_min=2013, year_max=2019, months=list(range(1, 13)), fixed_cost=240000, variable_cost=3, tilt=5, air_density=1.23, rotor_diameter=190, rated_rotor_rpm=10, rated_wind_speed=11.5, v_cut_in=4, v_cut_out=30, n_turbine=None, turbine_size=10, data_path='', save_path='stored_model_runs/', save=True)

Parameter	Type	Description
tilt	*float*	*Blade tilt in degrees*
air_density	*float*	*Density of air in kg/m3*
rotor_diameter	*float*	*Rotor diameter in m*
rated_rotor_rpm	*float*	*Rated rotation speed in rpm*
rated_wind_speed	*float*	*Rated wind speed in m/s*

v_cut_in	*float*	*Cut in wind speed in m/s*
v_cut_out	*float*	*Cut out wind speed in m/s*
n_turbine	*Array-like*	*Relative number of turbines installed at each site, defaults to an even distribution across sites*
turbine_size	*float*	*Size of each turbine in MW*
save	*boo*	*Determines whether to save the results of the run*

1.1.4 Onshore Wind

OnshoreWindModel(sites='all', year_min=2013, year_max=2019, months=list(range(1, 13)), fixed_cost=120000, variable_cost=6, tilt=5, air_density=1.23, rotor_diameter=120, rated_rotor_rpm=13, rated_wind_speed=12.5, v_cut_in=3, v_cut_out=25, n_turbine=None, turbine_size=3.6, hub_height=90, data_path='', save_path='stored_model_runs/', save=True)

Parameter	Type	Description
tilt	*float*	*Blade tilt in degrees*
air_density	*float*	*Density of air in kg/m3*
rotor_diameter	*float*	*Rotor diameter in m*
rated_rotor_rpm	*float*	*Rated rotation speed in rpm*
rated_wind_speed	*float*	*Rated wind speed in m/s*
v_cut_in	*float*	*Cut in wind speed in m/s*
v_cut_out	*float*	*Cut out wind speed in m/s*
n_turbine	*Array-like*	*Relative number of turbines installed at each site, defaults to an even distribution across sites*
turbine_size	*float*	*Size of each turbine in MW*
hub_height	*float*	*Height of turbine hub - needed to adjust the wind speed data.*
save	*boo*	*Determines whether to save the results of the run*

1.1.4 Solar PV

SolarModel(sites='all', year_min=2013, year_max=2019, months=list(range(1, 13)), fixed_cost=42000, variable_cost=0, orient=0, tilt=22, efficiency=0.17, performance_ratio=0.85, plant_capacity=1, area_factor=5.84, data_path='', save_path='stored_model_runs/', save=True)

Parameter	Type	Description
orient	*float*	*Surface azimuth angle in degrees*
tilt	*float*	*Panel tilt in degrees*
efficiency	*float*	*Panel efficiency (0-1)*
performance_ratio	*float*	*Panel performance ratio - determined analytically (0-1)*
plant_capacity	*float*	*Installed capacity in MW*
area_factor	*float*	*Panel area per installed kW in m2/kW*
save	*boo*	*Determines whether to save the results of the run*

1.1.5 Tidal

1.2 Functions

1.2.1 Running the model

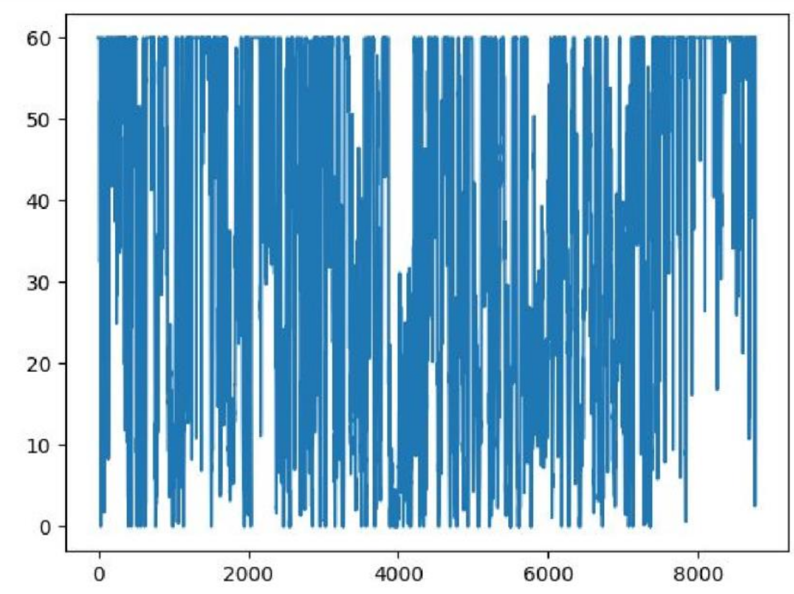
run()

This will populate the object's power_out parameter, either by loading a result from save_path, or by running the relevant model using the data located in data_path. This is called during initialisation of an object if a stored model run is not available.

Example: The following plots the output of six 10 MW offshore wind turbines for 2015, one each at sites 1 and 2, and four at site 3.

```
from generation import OffshoreWindModel import
matplotlib.pyplot as plt
 gen = OffshoreWindModel(sites=[1,2,3], year_min=2015, year_max=2015, turbine_size=10,
n_turbine=[1,1,4], data_path='data/offshore/')
 plt.plot(gen.power_out)
 plt.show()
```

[out]:



1.2.2 Load factor calculation

get_load_factor()

This will return the load factor in percent (0-100) of the predicted output power vector.

Example: The following calculates the aggregate load factor of solar stations uniformly distributed across the available locations between 2013-19.

```
from generation import SolarModel
 gen = SolarModel(sites='all',year_min=2013, year_max=2019,
data_path='data/solar/')
 print(gen.get_load_factor())
```

[out]:

    10.791703612581438

1.2.4 Scaling the amount of installed generation

scale_output(installed_capacity)

This will return an array of the hourly average output over a 24 hour period.

Parameter	Type	Description
installed_capacity	*float*	*Aggregated installed capacity in MW*

1.2.4 Getting the average daily output curve

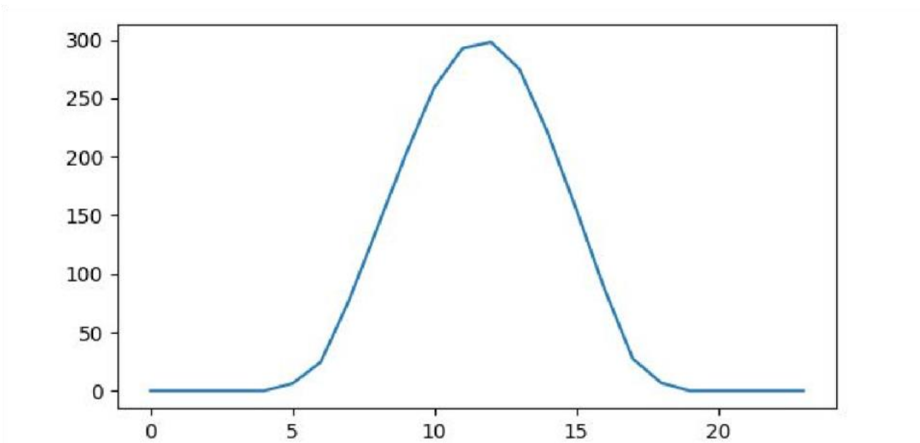
get_dirunal_profile()

This will return an array containing the hourly average output over a 24 hour period.

Example: Plotting the average daily output of 800 MW of solar, evenly distributed across all sites, loading from a saved run of 2013-2019.

```
from generation import SolarModel import
matplotlib.pyplot as plt
 gen = SolarModel()
 gen.scale_output(800)
 plt.plot(gen.get_diurnal_profile())
 plt.show()
```

[out]:



2 Individual Storage Models

2.1 Classes

2.1.1 Base Class

StorageModel(eff_in, eff_out, self_dis, variable_cost, fixed_cost, max_c_rate, max_d_rate, name, capacity=1)

Parameter	Type	Description
eff_in	*float*	*Charging efficiency in % (0-100)*

eff_out	*float*	*Discharging efficiency in % (0-100)*
self_dis	*float*	*Rate of self discharge in %/month (0-100)*
fixed_cost	*float*	*Cost incurred per MWh-year of installed capacity in GBP*
variable_cost	*float*	*Cost incurred per MWh of storage throughput in GBP*
name	*str*	*Name of storage - used for graph plotting*
max_c_rate	*float*	*Maximum charging rate in %/hr (0-100)*
max_d_rate	*float*	*Maximum discharging rate in %/hr (0-100)*
capacity	*float*	*Installed storage capacity in MWh*

2.1.2 Li-Ion Battery Storage

BatteryStorageModel(eff_in=95, eff_out=95, self_dis=2, variable_cost=0, fixed_cost=16000, max_c_rate=100, max_d_rate=100, capacity=1)

2.1.3 Hydrogen Storage

HydrogenStorageModel(eff_in=67, eff_out=56, self_dis=0, variable_cost=42.5, fixed_cost=120, max_c_rate=0.032, max_d_rate=0.15, capacity=1)

2.1.4 Pumped Thermal Storage

ThermalStorageModel(eff_in=80, eff_out=47, self_dis=9.66,variable_cost=331.6, fixed_cost=773.5, max_c_rate=8.56, max_d_rate=6.82, capacity=1)

2.2 Functions

2.2.1 Running a charging simulation

charge_sim(surplus, t_res=1, return_output=False, start_up_time=0)

This simulates the opportunistic operation of the storage to remove the negative values in the input array 'surplus'. It returns the percentage of times when a negative surplus is avoided after operation of the storage and, if requested, the output vector after the storage has been used.

Parameter	Type	Description
surplus	*Array like*	*The generation net demand which is to be smoothed in MW*

t_res	*float*	*The time resolution in hours of the surplus provided*
return_output	*boo*	*Whether to also return the smoothed output vector*
start_up_time	*int*	*The number of first time intervals to be ignored in reliability calculation*

Example: get the reliability with which an 3.6 MW onshore wind turbine at site 20 with a 10 MWh battery can reach a minimum output of 1 MW.

```
from generation import OnshoreWindModel from storage
import BatteryStorageModel import numpy as np
 gen = OnshoreWindModel(turbine_size=3.6, year_min=2013, year_max=2013, sites=[20],
data_path='data/wind/')
 wind_power = np.array(gen.power_out) surplus =
wind_power - np.array([1]*(365*24))
 stor = BatteryStorageModel(capacity=10)
 print(stor.charge_sim(surplus))
```

[out]:

77.52283105022832

2.2.2 Analysing the storage throughput

analyse_usage()

After running a simulation this will return the energy put into storage, the energy extracted from storage, and the total energy curtailed (positive surplus that was not put into storage)

Example: analysis following the previous example.

```
print(stor.analyse_usage())
```

[out]:

[721.7000558527664, 640.0424948515829, 8774.16639791325]

2.2.3 Sizing a storage system

size_storage(surplus, reliability, initial_capacity=0, req_res=1e3, t_res=1, max_capacity=1e8, start_up_time=0)

This uses bisection out the capacity of storage required to achieve the specified level of reliability. Note that if the max_capacity is too oversized, then the self-discharge rate can mean that increasing storage will actually decrease reliability, so it is important to use a maximum capacity that is not unrealistically large.

Parameter	Type	Description
surplus	*Array like*	*The generation net demand which is to be smoothed in MW*
t_res	*float*	*The time resolution in hours of the surplus provided*
initial_capacity	*float*	*The smallest amount of storage to try, if this achieves greater than the stated reliability then an error is raised.*
req_res	*float*	*The precision to which the required storage needs to be achieved*
max_capacity	*float*	*The maximum amount of storage to try, if this is insufficient to achieve the required reliability np.inf will be returned.*
start_up_time	*int*	*The number of first time intervals to be ignored in reliability calculation*

Example: for the example above work out the amount of storage required to achieve 90% reliability

```
print(stor.size_storage(surplus, 90, max_storage=1000, req_res=1e-3))
```

```
[out]:
        37.12797164916992
```

2.2.4 Calculating the cost of the storage system

get_cost()

Following a charging simulation, this will return the cost in GBP per year of operating the storage system

Example: Calculate the cost of the storage in the 90% reliable system above

```
print(stor.get_cost())
```

```
[out]:
        594039.9169921875
```

2.2.5 Plotting Storage Timeseries After Optimisation

This function can be used after the system has been optimised (for operation alone or operation + sizing). It will plot the storage assets state of charge and charging decisions in a timeseries.

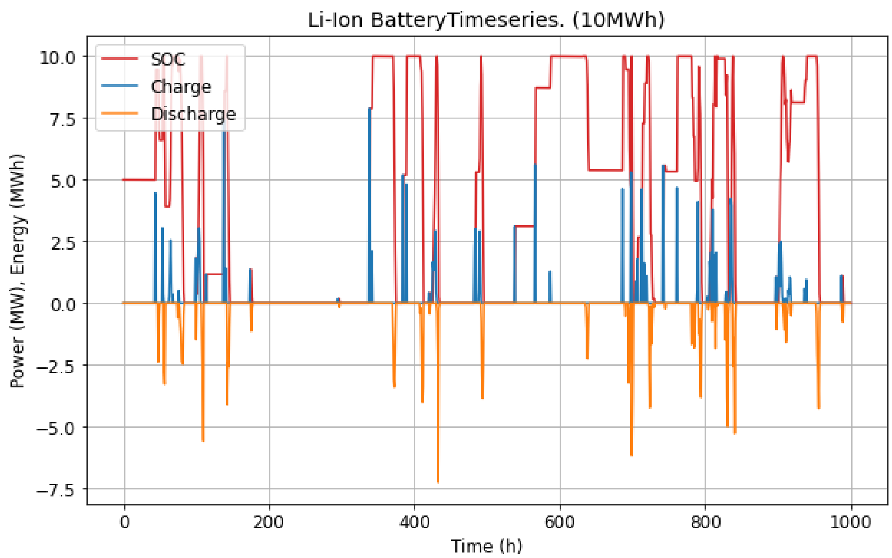
plot_timeseries(start,end):

- **start**: (int) start time of plot (h)
- **end:** (int) end time of plot (h)

Example: plot the first 100h of 10 MWh battery operation after an optimisation

BatStor.plot_timeseries(start = 0, end =1000)

[out]:



3 Multiple Storage Models

3.1 Base Class

MultipleStorageAssets(assets, c_order=None, d_order=None)

Parameter	Type	Description
assets	*Array like*	*A list of StorageModel objects*
c_order	*Array like*	*The priority order for charging under 'ordered' strategy as reference to index in the assets list e.g. [0,1]. If none, assumed to be reverse of the order provided in assets*
d_order	*Array like*	*The priority order for discharging. If none, assumed to be in the order provided in assets list.*

3.2 Functions

3.2.1 Running a charging simulation

charge_sim(surplus,t_res=1, return_output=False, start_up_time=0, strategy='ordered', return_di_av=False)

This simulates the opportunistic operation of the multiple storage assets to remove the negative values in the input array 'surplus'. It returns the percentage of times when a negative surplus is avoided after operation of the storage and, if requested, the output vector after the storage has been used or the daily average charge and discharge profiles.

Parameter	Type	Description
surplus	*Array like*	*The generation net demand which is to be smoothed in MW*
t_res	*float*	*The time resolution in hours of the surplus provided*
return_output	*boo*	*Whether to also return the smoothed output vector*
start_up_time	*int*	*The number of first time intervals to be ignored in reliability calculation*
strategy	*str*	*The strategy to use to operate multiple storage assets. Options: 'ordered'*
return_di_av	*boo*	*Whether to return the daily average charge/discharging profiles of each asset*

Example: get the reliability with which an 3.6 MW onshore wind turbine at site 20 with a 1 MWh battery and 10 MWh of hydrogen can reach a minimum output of 1 MW.

```
from generation import OnshoreWindModel
from storage import BatteryStorageModel, HydrogenStorageModel,
MultipleStorageAssets import numpy as np
 gen = OnshoreWindModel(turbine_size=3.6, year_min=2013, year_max=2013, sites=[20],
data_path='data/wind/')
 wind_power = np.array(gen.power_out)
 surplus = wind_power - np.array([1]*(365*24))

stor = MultipleStorageAssets([BatteryStorageModel(capacity=1),
                              HydrogenStorageModel(capacity=10)])
 print(stor.charge_sim(surplus))

[out]:
        67.51141552511416
```

3.2.2 Analysing the storage throughput

analyse_usage()

This will return one array per storage medium, containing the energy put into and taken out of each asset. The total energy curtailed will also be returned as float.

Example: analysis following the previous example.

```
print(stor.analyse_usage())
```

```
[out]:
        [[123.46064881468301, 16.659119543870013],
         [110.31644049533065, 6.033549637298847], 9362.25070352625]
```

3.2.3 Sizing a storage system

size_storage(surplus, reliability, initial_capacity=0, req_res=1e3, t_res=1, max_capacity=1e8, start_up_time=0)

This will size the total storage capacity required to reach a certain reliability, while keeping the relative sizes of the individual assets constant.

Parameter	Type	Description
surplus	*Array like*	*The generation net demand which is to be smoothed in MW*
t_res	*float*	*The time resolution in hours of the surplus provided*
initial_capacity	*float*	*The smallest amount of storage to try, if this achieves greater than the stated reliability then an error is raised.*
req_res	*float*	*The precision to which the required storage needs to be achieved*
max_capacity	*float*	*The maximum amount of storage to try, if this is insufficient to achieve the required reliability np.inf will be returned.*
start_up_time	*int*	*The number of first time intervals to be ignored in reliability calculation*

Example: for the example above work out the amount of storage required to achieve 90% reliability

```
print(stor.size_storage(surplus, 90, max_storage=1e5, req_res=1e-3))
```

```
[out]:
        370.10887498781085
```

3.2.4 Calculating the cost of the storage system

get_cost()

Following a charging simulation, this will return the cost in GBP per year of operating the storage system

Example: Calculate the cost of the storage in the 90% reliable system above

print(stor.get_cost())

[out]:

589993.8299611415

3.2.5 Running Causal System Operation Including EVs

For a given energy surplus, charges the batteries and aggregated EV fleets in specified order. The V2G fleets have a chosen state of charge level below which they will not discharge (V2G_discharge_threshold). All storage units and aggregate EV fleets begin the simulation at the specified fraction of full charge (initial_SOC):

causal_system_operation(demand, power, c_order, d_order, start, end, Mult_aggEV, plot_timeseries, V2G_discharge_threshold, initial_SOC):

- **demand:** array <floats> this is +ve values, a timeseries of the system passive demand (i.e. that not from EVs) (MW)
- **power:** array <float> generation profile of the renewables (MW), must be the same length as the demand
- **Mult_aggEV:** (MultipleAggregatedEVs) different fleets of EVs with defined chargertype ratios!
- **c_order:** list <int>, of order of the charge with c_order[0] being charged first, c_order[1] charged second etc..., the numbering refers to:
    - 0:(n_stor_assets-1) refers to the storage units in order
    - n_stor_assets:(n_stor_assets + 2*n_aggEV_fleets -1) for EV fleets, where the number refer to the virtual batteries representing: V2G_fleet0, smart_fleet0, V2G_fleet1, smart_fleet1...
- **plot_timeseries:** (bool), if true will plot the storage state of charge and charge/discharge decisions, as well as the surplus before and after adjustment from the storage devices.
- **start/end**: <datetime> the start and end time of the simulation. These are needed to construct the correct EV connectivity timeseries.
- **V2G_discharge_threshold:** (float), The kWh limit for the EV batteries, below whcih V2G will not discharge. The state of charge can still drop below this due to driving energy, but V2G will not discharge when the SOC is less than this value.
- **initial_SOC**: <float>, value between 0:1, determines the start SOC of the EVs and batteries (i.e. 0.5 corresponds to them starting 50% fully charged)

== returns ==

- **dataframe <Causal Reliability, EV_Reliability>:**
    - **Causal Reliability** is the % total demand (EV demand + passive demand) that is met by renewable energy

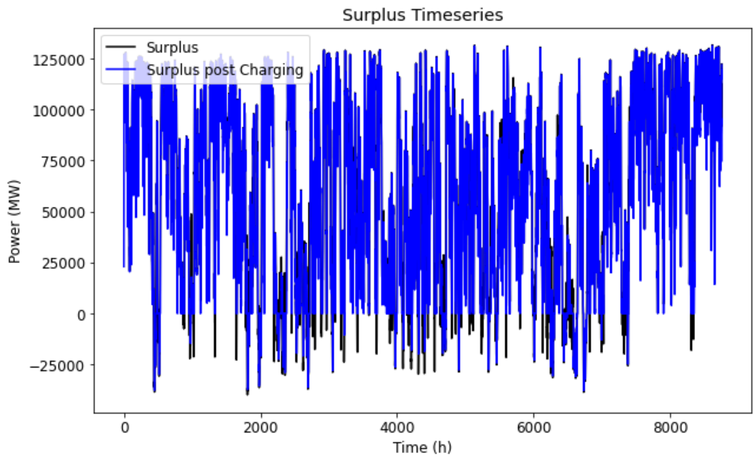
- **EV_Reliability** is the % of driving energy met by renewable energy. Given in order [Fleet0 V2g, Fleet0 Unidirectional, Fleet1 V2G, ...] For V2G this can be -ve, as when the EVs are plugged back in they can be discharged to zero again, thus they will need to be charged to 90% from zero rather than from about 30% as for the Unidirectional. Thus the energy needed from fossil fuels is larger that the driving energy.

Example: Operate the system causally (saved as simulation_ex.py on Github)

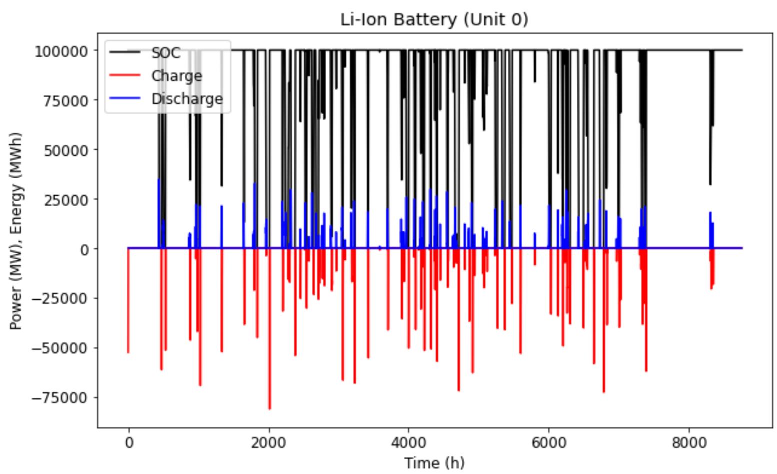
```
from generation import (OffshoreWindModel,SolarModel)
import aggregatedEVs as aggEV
from opt_con_class import (System_LinProg_Model)
import numpy as np
from storage import (BatteryStorageModel, HydrogenStorageModel,
              MultipleStorageAssets)
from fns import get_GB_demand
from datetime import datetime

ymin = 2015
ymax = 2015

#Define the generators
osw_master = OffshoreWindModel(year_min=ymin, year_max=ymax,
                  sites=[119,174,178,209,364], data_path='data/150m/')

#System has 150GW of Wind
power = np.asarray(osw_master.power_out)
power = power/max(power) * 150000

#Define a Fleet of 100000 EVs, half have V2G Chargers
Dom1 = aggEV.AggregatedEVModel(eff_in=95, eff_out=95, chargertype=[0.5,0.5],
chargercost=np.array([2000/20,800/20,50/20]),
                     max_c_rate=10, max_d_rate=10, min_SOC=0, max_SOC=36,
                     number=10000000,initial_number = 0.9,
                    Ein = 20, Eout = 36,
                    Nin = np.array([0,0,0,0,0,0,0,0,0.1,0,0,0,0,0,0.1,0.1,0.1,0.1,0,0,0,0,0]),
                    Nout = np.array([0,0,0,0,0,0,0,0.2,0.2,0,0,0,0,0,0,0.1,0,0,0,0,0,0,0,0]),
                    Nin_weekend = np.array([0,0,0,0,0,0,0,0.0,0.0,0,0,0,0,0,0,0,0.0,0,0,0,0,0,0,0]),
                    Nout_weekend = np.array([0,0,0,0,0,0,0,0.0,0.0,0,0,0,0,0,0,0,0.0,0,0,0,0,0,0,0]),
                    name = 'Domestic1')
```

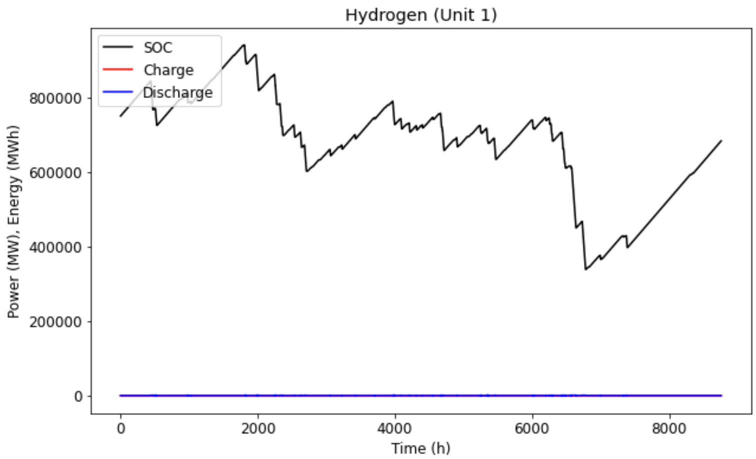
```
#Define Multiple Fleet Object
MultsFleets = aggEV.MultipleAggregatedEVs([Dom1])

#Define Demand
demand = np.asarray(get_GB_demand(ymin,ymax,list(range(1,13)),False,False))


#Storage Units, 100GWh Batteries, 1 TWh Hydrogen
B = BatteryStorageModel(capacity = 100000)
H = HydrogenStorageModel(capacity = 1000000)

Mult_Stor = MultipleStorageAssets([B,H])

### Causal ####
x1 = Mult_Stor.causal_system_operation(demand,power,[2,3,0,1],[0,1,3,2],MultsFleets, start = datetime(ymin,1,1,0),end =datetime(ymax+1,1,1,0),plot_timeseries = True,V2G_discharge_threshold = 20.0,initial_SOC=[0.5,0.75,0.6,1])
print(x1)
```

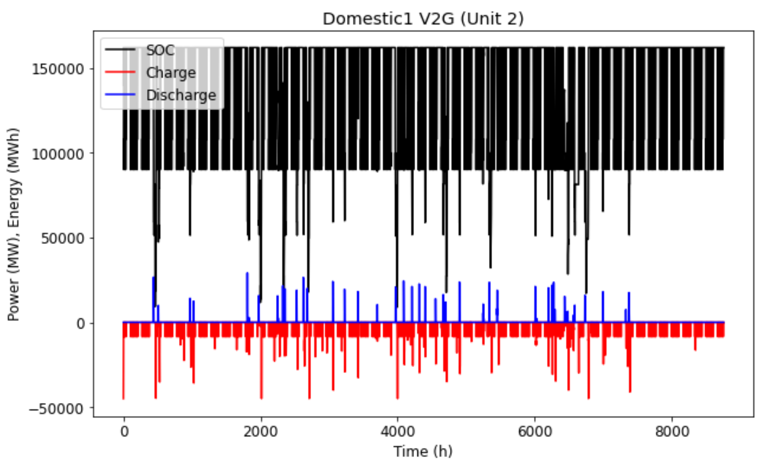
[out]:



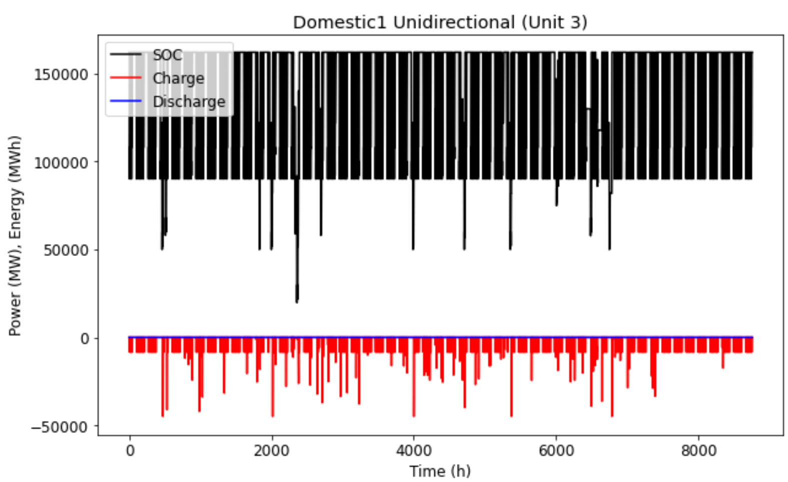
The -ve surplus (blue lines below 0) must be met with fossil fuels, thus contribute to reduced reliability.







The V2G discharge threshold means that the V2G won't discharge when the average EV has an SOC under 20 kWh.



Causal Reliability	Fleet 0 V2G	Fleet 0 Uni
97.657723	100.0	100.0

The function returns this dataframe. Note that 100% EV driving energy is met with renewable energy, and 97.65% of (passive+EV demand) is met with renewables overall.

3.2.6 Running Non-Causal System Operation Including EVs

This will find the optimal charging strategy for the batteries and aggregate EVs for a given surplus. It assumes full knowledge of the generation and demand over the entire simulation, thus is non_causal (i.e. has perfect forecasts). It does this by forming an linear programme optimisation with fixed storage and charger capacities, and then optimising for the given demand and generation profiles that are input. Fossil fuel usage is allowed but is very heavily penalised in the cost function, thus it is only used as an absolute last resort. The non_causal reliability is output (1- fossil fuel use / (EV demand + passive demand)) * 100%. Can also plot the EV and storage charging routines.

non_causal_system_operation(demand, power, start, end, Mult_aggEV,
plot_timeseries, InitialSOC, form_model):

== description ==

This function non-causally operate the storage and EVs over the given year. To save time on repeated operations, the model can be specified weather it needs to be rebuilt or not.

== parameters ==

- **demand:** array <floats> this is +ve values, a timeseries of the system passive demand (i.e. that not from EVs) (MW)
- **power:** array <float> generation profile of the renewables (MW), must be the same length as the demand
- **Mult_aggEV:** (MultipleAggregatedEVs) different fleets of EVs with defined chargertype ratios!
- **plot_timeseries**: (bool), if true will plot the storage SOCs and charge/discharge, as well as the surplus before and after adjustement.
- **start/end**: <datetime> the start and end time of the simulation. These are needed to construct the correct EV connectivity timeseries.
- **initial_SOC**: <float>, value between 0:1, determines the start SOC of the EVs and batteries (i.e. 0.5 corresponds to them starting 50% fully charged)
- **form_model**: (bool), when true the function will form the entire model, when false it will use the model previously created (this saves time during repeated simulations, as the model is only formed once)

== returns ==

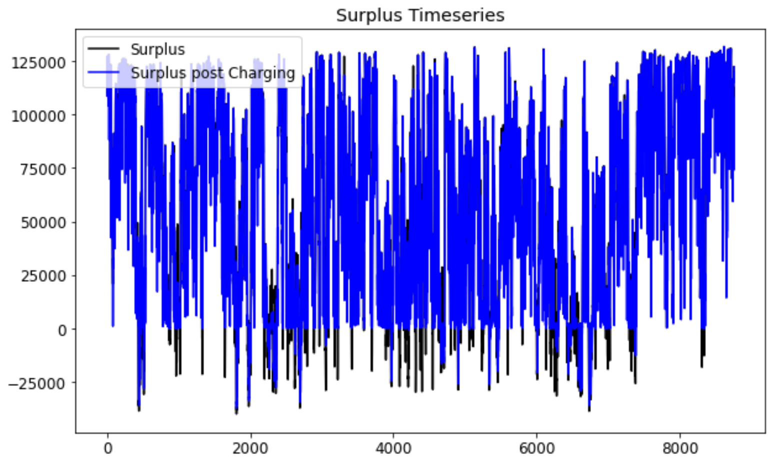
- **Non Causal Reliability** <float>: Non Causal Reliability is the % total demand (EV demand + passive demand) that is met by renewable energy. Unlike Non Causal Operation, EV reliability is always 100% as these are hard constraints within the optimisation. This may come at the cost of decreased total Causal reliability however.

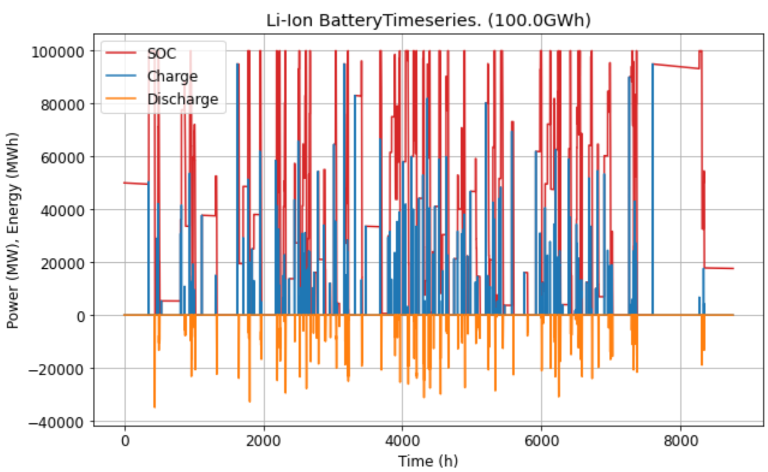
Example: Operate the system non-causally (carrying on from the Causal system operation example above)

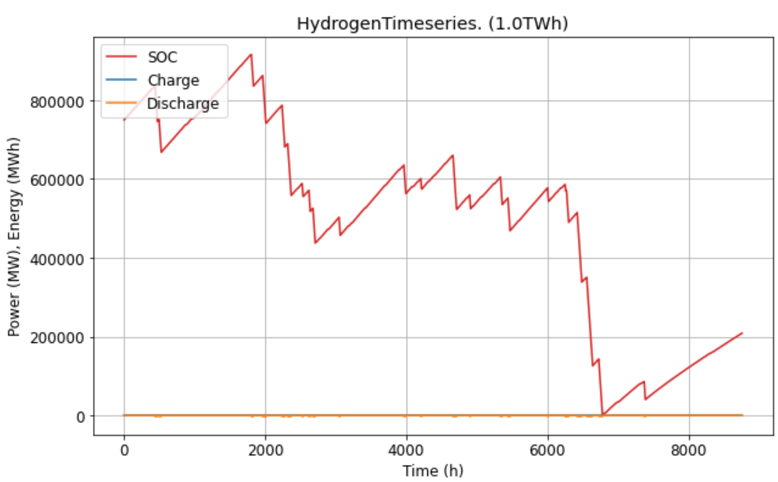
```
# #### Non Causal ####
```

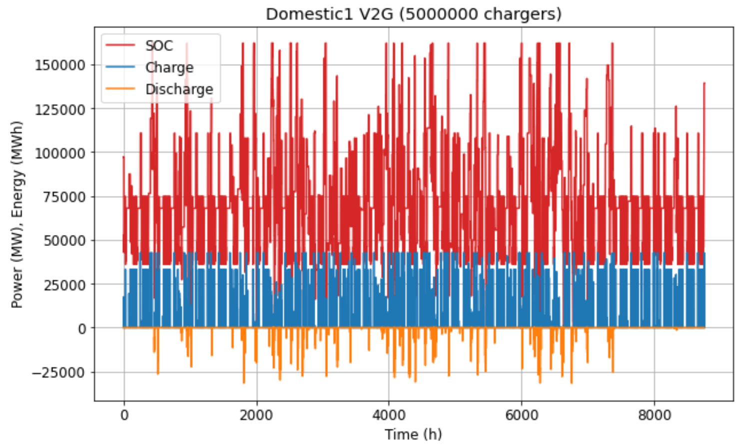
```
x2 = Mult_Stor.non_causal_system_operation(demand,power,MultsFleets,start = datetime(ymin,1,1,0), end =datetime(ymax+1,1,1,0), plot_timeseries = True,InitialSOC=[0.5,0.75,0.6,1])
print(x2)
```

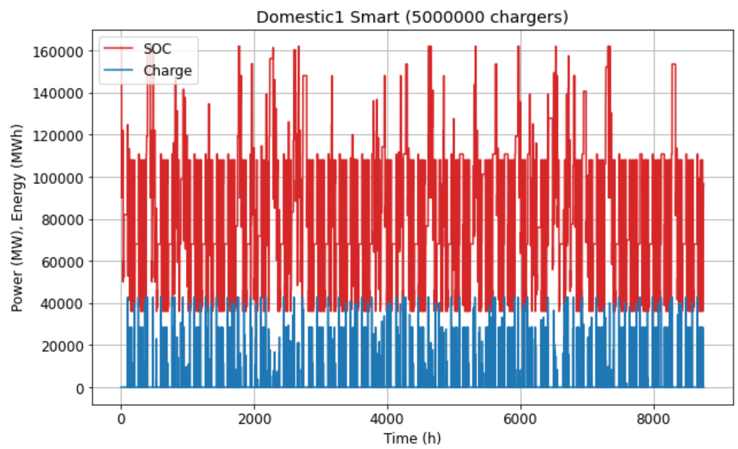
[out]:











The output reliability is: **98.30%.** The non-causal reliability will always be at least as high as the causal because it has the benefit of perfect forecasts when planning the charging behaviour.

4 Electricity System

This module combines a set of generation models with a multiple storage asset model to simulate a systems ability to meet an electricity demand profile.

4.1 Classes

4.1.1 Base Class

ElectricitySystem(*gen_list, stor_list, demand, t_res=1, reliability=99, start_up_time=0, strategy='ordered'*)

Parameter	Type	Description
gen_list	*Array-like*	*List of generation model objects*
stor_list	*MultipleStorageAssets*	*MultipleStorageAssets object*

demand	*Array-like*	*Demand to be met in MW*
t_res	*float*	*Time resolution (hours)*
reliability	*float*	*Percentage of demand to be met (0-100)*
start_up_time	*int*	*Number of first time intervals to ignore in reliability*
strategy	*str*	*Storage operation strategy - 'ordered'*

4.1.2 GB electricity system

ElectricitySystemGB(*gen_list, stor_list, t_res=1, reliability=99, start_up_time=40*24*4, strategy='ordered', electrify_heat=False, evs=False, months=list(range(1,14)), year_min=2014, year_max=2019)*

Parameter	Type	Description
months	*Array-like*	*List of months to be included in the simulation (1-12)*
year_min	*int*	*Lowest year to be included in the simulation*
year_max	*int*	*Highest year to be included in the simulation*
electrify_heat	*boo*	*Whether to include electrified heating demand*
evs	*boo*	*Whether to include domestic EV charging*

4.2 Functions

4.2.1 System cost calculations

cost(x)

Returns the whole system cost in £bn /yr.

Parameter	Type	Description
$x[:n_{gen}]$	*Array-like*	*Installed capacity of each generator in GW (using order of gen_list)*
$x[n_{gen}:]$	*Array-like*	*Capacity of first n-1 storage assets relative to total installed. Must sum to less than 1. If only one storage asset then it will be empty.*

Example: The following returns the cost of running the GB system with existing demand using 60 GW each of offshore, onshore, and solar, alongside a 10/90 mix of batteries and hydrogen storage.

```
from generation import OffshoreWindModel, OnshoreWindModel, SolarModel from storage import
BatteryStorageModel, HydrogenStorageModel from system import ElectricitySystemGB
 gen = [OffshoreWindModel(), OnshoreWindModel(), SolarModel()]
 stor = [BatteryStorageModel(), HydrogenStorageModel()]
 es = ElectricitySystemGB(gen, stor, reliability=99)
 print(es.cost([60,60,60,0.1]))
```

[out]:

    42.958280920024505

4.2.2 System operation analysis

analyse(x, filename='log/system_analysis.txt')

Runs a whole system simulation and writes a text file in the log with the system cost breakdown, the amounts and utilisation of storage installed, and total energy curtailment.

Parameter	Type	Description
x[:n_{gen}]	*Array-like*	*Installed capacity of each generator in GW (using order of gen_list)*
x[n_{gen}:]	*Array-like*	*Capacity of first n-1 storage assets relative to total installed. Must sum to less than 1. If only one storage asset then it will be empty.*
filename	*str*	*File path for analysis to be stored, should end in .txt*

Example: The following analyses the system from the previous example.

```
from generation import OffshoreWindModel, OnshoreWindModel, SolarModel from storage import
BatteryStorageModel, HydrogenStorageModel from system import ElectricitySystemGB

gen = [OffshoreWindModel(), OnshoreWindModel(), SolarModel()]
 stor = [BatteryStorageModel(), HydrogenStorageModel()]
 es = ElectricitySystemGB(gen, stor, reliability=99)
 es.analyse([60,60,60,0.1],filename='log/analysis.txt')
```

[out]:

System cost: £42.958280920024505 bn/yr

-------------------- INSTALLED GENERATION

Offshore Wind: 60 GW
Onshore Wind: 60 GW
Solar: 60 GW

>>TOTAL: 180 GW

------------------ INSTALLED STORAGE

Li-Ion Battery: 0.4882112529278146 TWh
Hydrogen: 4.494901276450441 TWh

>>TOTAL: 4.882112529278146 TWh

-------------------- STORAGE UTILISATION

>> Li-Ion Battery <<

1.426515280796148 TWh/yr in (grid side)
1.275665601444428 TWh/yr out (grid side)
4.458956654147789 cycles per year

>> Hydrogen <<

0.2718659270924416 TWh/yr in (grid side)
0.07082591178249478 TWh/yr out (grid side)
0.04619874541792025 cycles per year

------------------- ENERGY UTILISATION

Total Demand: 288.49464577846245 TWh/yr
Total Supply: 580.6900640898285 TWh/yr
Curtailment: 41.52144687548876 TWh/yr

--------------- COST
BREAKDOWN

Offshore Wind: £15.44999218041244 bn/yr

Onshore Wind: £8.420025411401847 bn/yr
Solar: £2.52 bn/yr
Li-Ion Battery: £6.211480046845044 bn/yr
Hydrogen: £0.4568844814641887 bn/yr

4.2.3 Visualisation of daily load profiles

get_dirunal_profile(gen_cap, stor_cap)

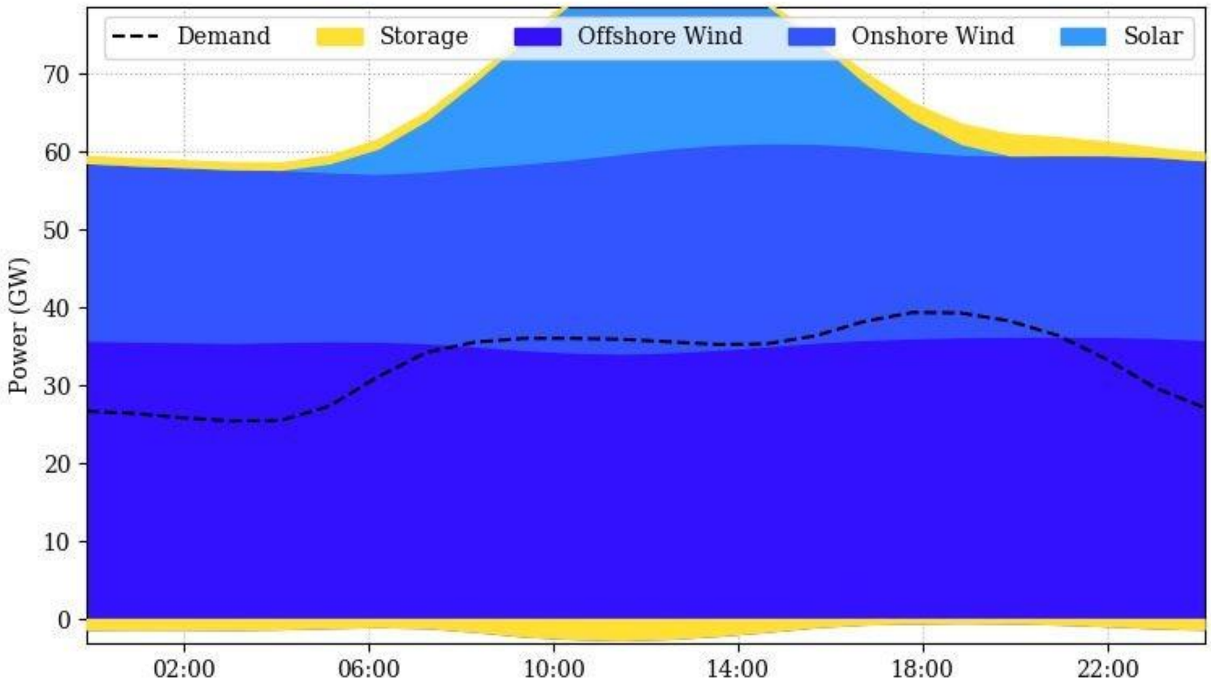
Plots the average daily supply and demand breakdown, alongside the average daily usage profile for each storage asset.

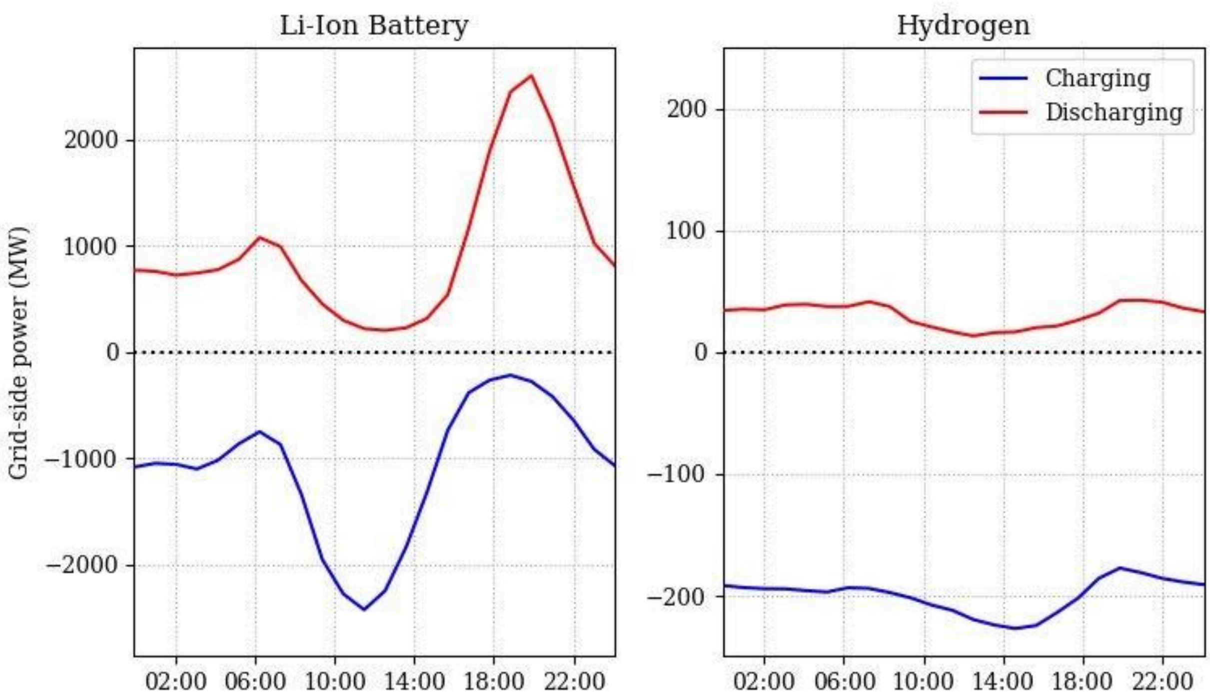
Parameter	Type	Description
gen_cap	*Array-like*	*Installed capacity of each generator in GW (using order of gen_list)*
stor_cap	*Array-like*	*Capacity of first n-1 storage assets relative to total installed. Must sum to less than 1. If only one storage asset then it will be empty.*

Example: The following plots the usage profile of the system from the previous example.

```
from generation import OffshoreWindModel, OnshoreWindModel, SolarModel from storage import BatteryStorageModel, HydrogenStorageModel from system import ElectricitySystemGB

gen = [OffshoreWindModel(), OnshoreWindModel(), SolarModel()]
 stor = [BatteryStorageModel(), HydrogenStorageModel()]
 es = ElectricitySystemGB(gen, stor, reliability=99)
 es.get_diurnal_profile([60,60,60],[0.1])
```

[out]:





5 Load factor estimator

This module uses the results from the generation models to estimate the load factor of a generator at a specific location, by interpolating the available data points.

5.1 Base Class

LoadFactorEstimator(gen_type, data_loc=None)

Parameter	Type	Description
gen_type	*str*	*Code to determine the type of generator. 'w' for onshore wind, 'osw' for offshore wind, and 's' for solar.*
data_loc	*str*	*Location of folder containing the raw weather data*

5.2 Functions

5.2.1 Determine load factors at all available sites

calculate_load_factors()

This will estimate the load factor of a generator at each of the sites provided, and store the results in the stored_model_runs folder.

5.2.2 Estimate the load factor at a particular point

estimate(lat, lon, max_dist=1, num_pts=3)

This will estimate the load factor of a generator at a specified location, by interpolating the estimated load factors at the site locations. A weighted average of the closest n points will be performed, providing the points are within the specified maximum distance.

Parameter	Type	Description
lat	*float*	*Location latitude*
lon	*float*	*Location longitude*
max_dist	*float*	*The largest straight line distance in degrees that a point will be used for interpolation*
num_pts	*int*	*The number of closest points that will be used for the weighted average*

Example: Calculate the load factor of a solar farm in Greenwich Park

```
from maps import LoadFactorEstimator
 lfe = LoadFactorEstimator('s')
```

```
print(lfe.estimate(51.48,0.00))
```

[out]:

```
12.933988121729667
```

6 Load factor maps

This module uses the results from the load factor estimation to plot maps showing the geographic variation in predicted load factor.

6.1 Classes

6.1.1 Base class

LoadFactorMap(load_factor_estimator, lat_min, lat_max, lon_min, lon_max, lat_num, lon_num, quality, is_land)

Parameter	Type	Description
load_factor_estimator	*LoadFactorEstimator*	*Object to fill in the estimates at each point*
lat_min	*float*	*Minimum latitude to show on map*
lon_min	*float*	*Minimum longitude to show on map*
lat_max	*float*	*Maximum latitude to show on map*
lon_max	*float*	*Maximum longitude to show on map*
quality	*str*	*'h' for high resolution, 'l' for low resolution*
is_land	*boo*	*Whether the shaded area is on land or off land*

6.1.2 Offshore wind map

OffshoreWindMap(lat_min=48.2, lat_max=61.2, lon_min=-10.0, lon_max=4.0, lat_num=400, lon_num=300, quality='h', data_loc=None)

Parameter	Type	Description
lat_num	*int*	*Number of x points on the shaded mesh*
lon_num	*int*	*Number of y points on the shaded mesh*
data_loc	*str*	*Path to weather data - required if load factors have not previously been saved*

6.1.3 Onshore wind map

OnshoreWindMap(lat_min=49.9, lat_max=59.0, lon_min=-7.5, lon_max=2.0, lat_num=400, lon_num=300, quality='h', turbine_size=3.6, data_loc=None)

Parameter	Type	Description
lat_num	*int*	*Number of x points on the shaded mesh*
lon_num	*int*	*Number of y points on the shaded mesh*
turbine_size	*float*	*Rated capacity of individual turbine in MW*
data_loc	*str*	*Path to weather data - required if load factors have not previously been saved*

6.1.4 Solar map

SolarMap(lat_min=49.9, lat_max=59.0, lon_min=-7.5, lon_max=2.0, lat_num=400, lon_num=300, quality='h', turbine_size=3.6, data_loc=None)

Parameter	Type	Description
lat_num	*int*	*Number of x points on the shaded mesh*
lon_num	*int*	*Number of y points on the shaded mesh*
data_loc	*str*	*Path to weather data - required if load factors have not previously been saved*

6.2 Functions

6.2.1 Draw a map

draw_map(show=True, savepath='', cmap=None, vmax=None, vmin=None)

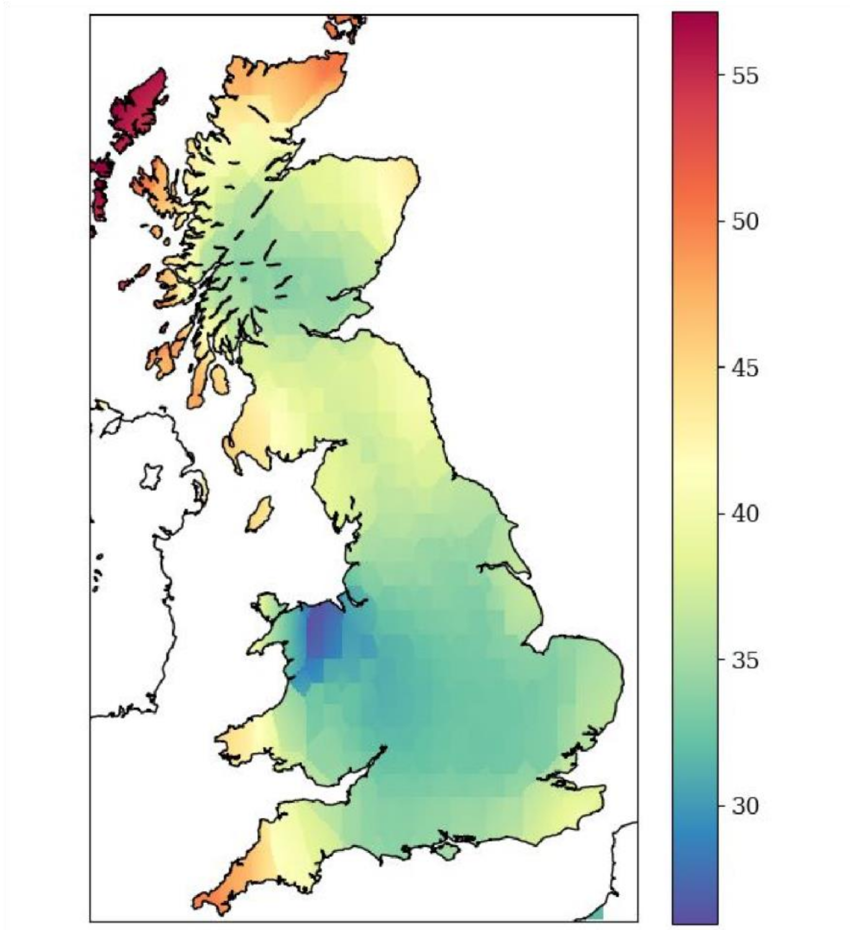
Parameter	Type	Description
show	*boo*	*Whether to show the result*
savepath	*str*	*If desired, location to save the result*
cmap	*matplotlib.cm*	*Color map to use for shading*
vmax	*float*	*Value to cap the colour map at (default is set to the max value)*
vmin	*float*	*Minimum value to cap colour map at (default is the min value)*

Example: Plot a map of onshore wind load factor

from maps import OnshoreWindMap

```
mp = OnshoreWindMap()
mp.draw_map()
```

[out]:



7 Aggregated EV Fleet Models

Brief intro to explain how the EVs are modelled, with a few diagrams.

7.1 Classes

7.1.1 Individual Fleet Class

AggregatedEVModel(eff_in, eff_out chargercost, max_c_rate, max_d_rate, min_SOC, max_SOC, number, initial_number, Ein, Eout, Nin, Nout, name, limits = [] , chargertype=[]):

Parameter	Type	Description
eff_in	*float*	*Charging efficiency in % (0-100)*
eff_out	*float*	*Discharging efficiency in % (0-100)*
Chargercost	*Array <float>*	*Cost of individual charger divided by years of service (£/yr). [V2G charger cost, Smart Unidirectional cost]*
max_c_rate	*float*	*the maximum charging rate (kW per Charger) from the grid side. (Assumed the same for smart and V2G)*
max_d_rate	*float*	*the maximum V2G discharging rate (kW per Charger) from the grid side. (Smart chargers cannot discharge)*
Min_SOC	*float*	*Min state of charge of individual EV (kWh).*
Max_SOC	*float*	*Max state of charge of individual EV (kWh).*
Number	*float*	*Number of EVs in fleet (all need a charger).*
Initial_number	*float*	*Proportion of chargers with EVs attached at the start of the simulation (0-1), (split evenly between charger types)*
Ein	*float*	*Energy stored within EVs when they plugin (kWh)*
Eout	*float*	*Energy of disconnected EVs (KWh). For the moment this must equal Max_SOC.*
Nin	*Array< float>*	*Normalised timeseries of EV connections during the week. This must have 24 entries corresponding to each hour of the day. (e.g. Nin[4] = 0.1 for number = 1000, indicates that 100 EVs plugin at 4am everyweekday). Sum(Nin) == sum(Nout)*
Nout	*Array<float>*	*Normalised timeseries of EV disconnections. Must have 24 entries corresponding to each hour of the day. (e.g. Nout[17] = 0.05 for number = 1000, indicates that 50 EVs disconnect at 5pm every week day).*

Nin_weekend	*Array< float>*	*Normalised timeseries of EV connections during the week. This must have 24 entries corresponding to each hour of the day. (e.g. Nin[4] = 0.1 for number = 1000, indicates that 100 EVs plugin at 4am every weekend day). Sum(Nin_weekend) == sum(Nout_weekend)*
Nout_weekend	*Array<float>*	*Normalised timeseries of EV disconnections. Must have 24 entries corresponding to each hour of the day. (e.g. Nout[17] = 0.05 for number = 1000, indicates that 50 EVs disconnect at 5pm every weekend day).*
Name	*str*	*Name of the fleet (e.g. Domestic, Work, Commercial..). Used for labelling plots*
chargertype	*Array <float>*	*Ratio of charger types, this is mostly used to store the outputs from optimisations. Chargertype[0] – V2G, chargertype[1] – Smart. Chagertype = [0.4,0.6] would indicate 40% of Evs have V2G charger, 60% have a Smart charger. Alternatively, specifying is necessary before running any simulations via the MultipleStorageClass.*

To clarify what is meant by grid side. If max_c_rate = 10kW, and eff_in = 50%, then when charging at max rate, 10kWh will be removed from the grid, and the SOC of the EV battery will increase by 5kWh.

7.1.2 DomesticFleet

7.1.3 WorkFleet

7.2 Functions

7.2.1 Plot Timeseries of Operation

Must be run after an optimisation, this will plot the state of charge and (dis)charge decisions over the time period of interest.

plot_timeseries(start,end,withSOClimits)

start: (int) start time of plot. (default = 0)

end: (int) end time of plot. (default is end of timehorizon)

withSOClimits: (bool) when true will plot the SOC limits imposed on the aggregate battery caused by EV driving patterns.

See Multiple Aggregated EV Fleet Model for further example.

8 Multiple Aggregated EV Fleet Models

8.1 Classes

8.1.1 Multiple Fleet Class

MultipleAggregatedEVs(assets)

Parameter	Type	Description
assets	*Array<AggregatedEVModel>*	*List of aggregated EV Model Objects*

8.2 Functions

The main functions for the Multiple_Aggregated_EV_Fleet class are contained within other modules, listed here:

- System_LinProg_Model class, where once a fleet is specified within the system, the optimiser can be used to find the optimal charge/discharge decisions of the aggregated fleet batteries. In doing so it will also decide on whether it is cost optimal to build Smart chargers, or pay more and build vehicle to grid chargers, allowing access to the increased balancing capabilities they facilitate.
- MultipleStorageAssets.causal_system_operation(): this method will causally simulate the system operation. It treats the aggregated EV batteries in a similar way to the batteries, where they have ranked charge and ranked discharge orders. They charge when renewables > demand, and discharge otherwise.
- MultipleStorageAssets.non_causal_system_operation(): this method will non causally simulate the system operation.

Example: Create Multiple Fleet Object, Optimise system to find optimal operation and types of charger to build. (saved as agg_EV_example.py on Github)

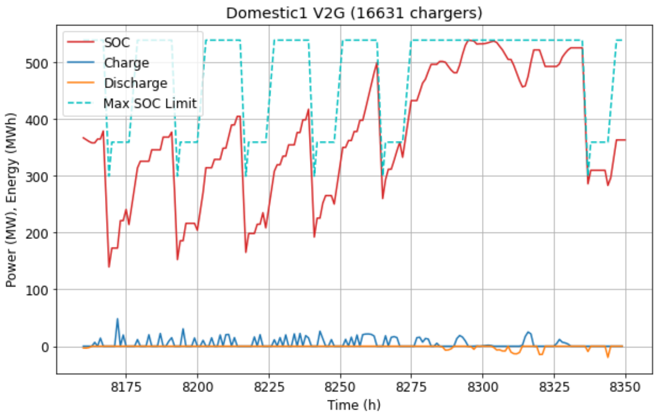
```
from generation import (OffshoreWindModel,SolarModel)
import aggregatedEVs as aggEV
from opt_con_class import (System_LinProg_Model)
from storage import (MultipleStorageAssets)
import numpy as np
import datetime as dt
from fns import get_GB_demand

ymin = 2015
ymax = 2015

#Define the generators
osw_master = OffshoreWindModel(year_min=ymin, year_max=ymax,
                 sites=[119,174,178,209,364], data_path='data/150m/')
```

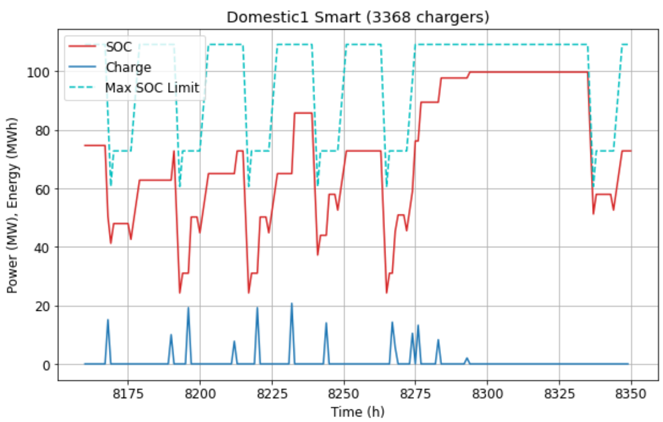
```
s = SolarModel(year_min=ymin, year_max=ymax, sites=[17,23,24],
                data_path='data/solar/')
generators = [s,osw_master]

#Define a Fleet of EVs
Dom1 = aggEV.AggregatedEVModel(eff_in=95, eff_out=95, chargertype=[0.5,0.5],
chargercost=np.array([2000/20,800/20,50/20]),
                    max_c_rate=10, max_d_rate=10, min_SOC=0, max_SOC=36,
                    number=20000,initial_number = 0.9,
                    Ein = 20, Eout = 36,
                    Nin = np.array([0,0,0,0,0,0,0,0,0.1,0,0,0,0,0,0.1,0.1,0.1,0.1,0,0,0,0,0]),
                    Nout = np.array([0,0,0,0,0,0,0,0.2,0.2,0,0,0,0,0,0,0.1,0,0,0,0,0,0,0,0]),
                    Nin_weekend = np.array([0,0,0,0,0,0,0,0.0,0.0,0,0,0,0,0,0,0.0,0,0,0,0,0,0,0,0]),
                    Nout_weekend = np.array([0,0,0,0,0,0,0,0.0,0.0,0,0,0,0,0,0,0.0,0,0,0,0,0,0,0,0]),
                    name = 'Domestic1')

#Define Multiple Fleet Object
MultsFleets = aggEV.MultipleAggregatedEVs([Dom1])

#Define Demand, normalised to max 15MW
demand = np.asarray(get_GB_demand(ymin,ymax,list(range(1,13)),False,False))
demand = - demand/max(demand) * 30.0

#Form Model and solve allowing 2% of demand from fossil fuels
x = System_LinProg_Model(surplus = demand,fossilLimit = 0.02,Mult_Stor = MultipleStorageAssets([]),
Mult_aggEV = MultsFleets, gen_list=generators)
x.Form_Model(start_EV = dt.datetime(ymin,1,1,0),end_EV = dt.datetime(ymax+1,1,1,0))
x.Run_Sizing()

#Save Output DataFrame to csv
x.df_capital.to_csv('log/SomeFossil.csv', index=False)

#Plot A Week in December
MultsFleets.assets[0].plot_timeseries(start = 8160, end =8350,withSOClimits=True)
x.PlotSurplus(start = 8160, end =8350)
```

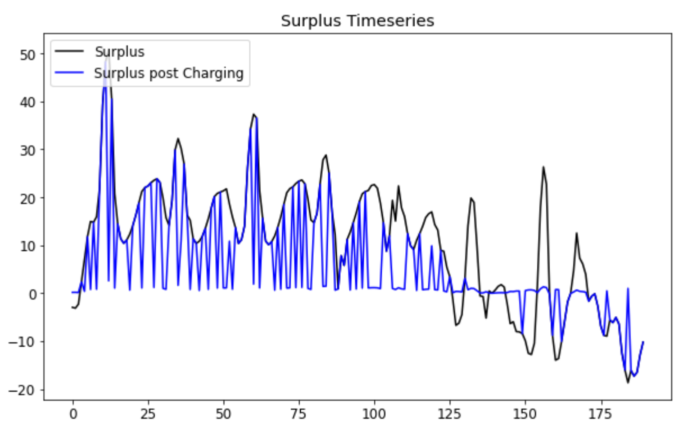
[out]:

Total Demand (GWh)	Total Fossil Fuel (GWh)	Total Curtailement (GWh)	Gen 0 Cap (GW)	Gen 1 Cap (GW)	Fleet 0 V2G	Fleet 0 Uni
165	3	150	0.11	0.03	16631	3368

This is the CSV. 110MW Solar Built, 30MW offshore Wind, 16631 V2G Chargers Built, 3368 Unidirectional Chargers built.


Domestic1 V2G (16631 chargers)
SOC
Charge
Discharge
Max SOC Limit
Power (MW), Energy (MWh)
Time (h)
0
100
200
300
400
500
8175
8200
8225
8250
8275
8300
8325
8350


Domestic1 Smart (3368 chargers)
SOC
Charge
Max SOC Limit
Power (MW), Energy (MWh)
Time (h)
0
20
40
60
80
100
8175
8200
8225
8250
8275
8300
8325
8350


Surplus Timeseries
Surplus
Surplus post Charging
−20
−10
0
10
20
30
40
50
0
25
50
75
100
125
150
175

8 Linear Program Model

8.1 Classes

8.1.1 System_LinProg_Model

The actual linear programme model is an object of this class, formed and then run by separate methods (described below). The class initialisation method mostly just checks that the input parameters are of the correct form.

System_LinProg_Model(surplus,fossilLimit,Mult_Stor,Mult_aggEV,gen_list=[],YearRange=[]):

Parameter	Type	Description
surplus	np.array <floats>	If optimising generation this is just demand (which is input with -ve values!). If not optimising generation, then enter full surplus with an empty gen_list.
fossilLimit	float	fraction of demand (i.e. -ve surplus) that can come from fossil fuels (expected values between 0:0.05....)
Mult_Stor	*MultipleStorageAssets()*	MUST be a multiple storage object, even if it has a length of zero!
Mult_aggEV	*MultipleAggregatedEVs()*	Must be multiple fleet object. (even if of length zero!)
gen_list	List < *GenerationModel* >	list of the potential renewable generators to build
YearRange	List <int>	[MinYear,MaxYear] of renewables.

8.2 Functions

8.1.1 Form_Model

This creates the model object of a linear programme. Useful because Pyomo programmes take a long time to form initially. Once formed though they can be 'solved' repeatedly whilst only changing specific parameters, this reduces construction time massively.

Form_Model(start_EV=-1,end_EV=-1, SizingThenOperation = False, includeleapdays=True, StartSOCEqualsEndSOC=True, InitialSOC = [-1])

Parameter	Type	Description
start_EV	Datetime()	This is used to setup the EV plugin timeseries, making sure that weekdays and weekends are properly aligned. Must be set correctly to first hour of simulation (usually midnight on Jan 1st ymin). When no EVs are being optimised it can remain at -1, otherwise an error will output.
end_EV	Datetime()	This is used to setup the EV plugin timeseries, making sure that weekdays and weekends are properly aligned. Must be set correctly to first hour of

		simulation (usually midnight on Jan 1st ymax+1). When no EVs are being optimised it can remain at -1, otherwise an error will output.
SizingThenOperation	Bool	Make this True when the intention is to use the model for repeated system sizing and operational simulation (using Run_Sizing_Then_Op() method). This means that the timehorizon will be one year less than the year range states and that the leap years will be removed. **Run_Sizing_Then_Op()** not fully functional yet so recommended just to keep this =False.
includeleapdays	Bool	When set to False the model will ignore leap days. Can be useful if want to run repeated sims with different years data without having to reform the model to include a leap year.
StartSOCEqualsEndSOC	Bool	When set to True the state of charge of all the storage and aggEV units at the first and last timestep will be equal. This insures no energy loss during simulation.
InitialSOC	List <float>	Fraction of the storage devices (inc EVs) energy capacity that is full at the start of the simulation. The default is to leave the initial SOC unconstrained. Can be used with or without the above Boolean = True. **2 entry methods**: 1. [Single value] – This will make all storage devices start at the same SOC fraction. 2. [Stor0, Stor1,…, V2G_0,Uni_0,V2G_0,…] If range of Values given allows the specifying of the start SOC fraction of each storage device individually. NB each fleet is split into 2 virtual batteries, one for V2G units and 1 for Unidirectional, thus need one entry here for each.

8.1.2 Form_Model

Solve the linear programme specified by Form_Model(). Results recorded by updating df_capital and df_costs. The operational timeseries for generators and storage are saved to their respective objects, to then be used with plotting functions.

Run_Sizing()

This method returns nothing, but updates some of the LinProgModel attributes, namely **df_capital and df_costs**. These are two dataframes that present in an easy to read and manipulate format the sizing decisions (in MW) and the respective costs.

8.1.3 PlotSurplus

This plots the generation-demand profiles pre and post charging actions have been applied to it. This is complemented by the plot_timeseries methods attached to the AggregatedFleet and Storage classes.

PlotSurplus(start = 0, end = -1)

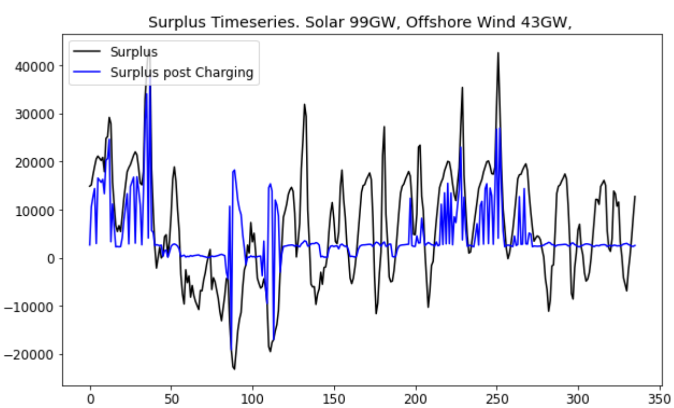
Parameter	Type	Description
start	int	First timestep user wishes to plot surpluses from. Default is the start

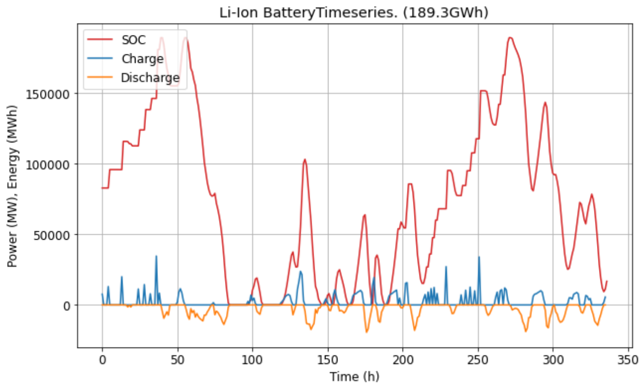
end	int	Last timestep user wishes to plot surpluses up to. Default value of -1 results in plotting until end of simulation.

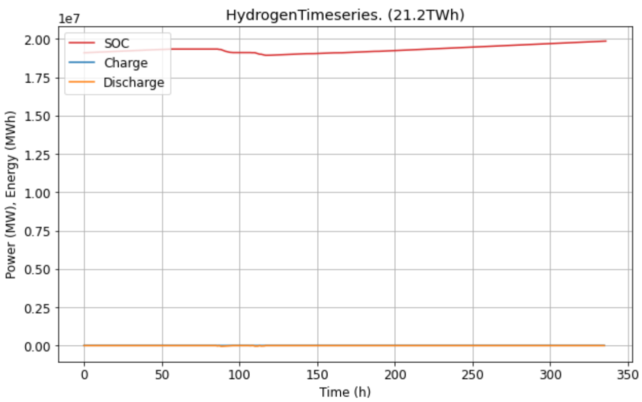
Example – Form simple Linear Programme and then solve. Plot and output the results (available on GitHub as LinProgExample.py):

```
from generation import (OffshoreWindModel,SolarModel)
import aggregatedEVs as aggEV
from opt_con_class import (System_LinProg_Model)
from storage import (BatteryStorageModel, HydrogenStorageModel,
                     MultipleStorageAssets)
import numpy as np
from fns import get_GB_demand

ymin = 2015
ymax = 2015

#Define the generators
osw_master = OffshoreWindModel(year_min=ymin, year_max=ymax,
                               sites=[119,174,178,209,364], data_path='data/150m/')

s = SolarModel(year_min=ymin, year_max=ymax, sites=[17,23,24],
                        data_path='data/solar/')
generators = [s,osw_master]

#Define the Storage

B = BatteryStorageModel()
H = HydrogenStorageModel()
storage = [B,H]


#Define Demand
demand = np.asarray(get_GB_demand(ymin,ymax,list(range(1,13)),False,False))

#Initialise LinProg Model
x = System_LinProg_Model(surplus = -demand,fossilLimit = 0.01,Mult_Stor = MultipleStorageAssets(storage),
                         Mult_aggEV = aggEV.MultipleAggregatedEVs([]), gen_list = generators,YearRange = [ymin,ymax])

#Form the Linear Program Model
x.Form_Model()

#Solve the Linear Program
x.Run_Sizing()

#Plot Results
x.PlotSurplus(0,336)
B.plot_timeseries(0,336)
H.plot_timeseries(0,336)

#Store Results
x.df_capital.to_csv('log/Capital.csv', index=False)
x.df_costs.to_csv('log/Costs.csv', index=False)
```

[out]:







Capital:

Capital:										
Total Demand (GWh)	Total Fossil Fuel (GWh)	Total Curtailement (GWh)	Gen 0 Cap (GW)	Gen 1 Cap (GW)	Stor 0 Cap (GWh)	Stor 1 Cap (GWh)				
290492	2904	58598	99.56	43.9	189.34	21291.95				
Costs:										
Length of Sizing (yr)	Gen 0 Capital (¬£m/yr)	Gen 0 Operation (¬£m/yr)	Gen 1 Capital (¬£m/yr)	Gen 1 Operation (¬£m/yr)	Stor 0 Capital (¬£m/yr)	Stor 0 Operation (¬£m/yr)	Stor 1 Capital (¬£m/yr)	Stor 1 Operation (¬£m/yr)	Total Capital (¬£m/yr)	Total Operation (¬£m/yr)
0	4181	0	10536	721	3029	0	2555	170	20303	889

The strange symbol to the left of £ symbols is a strange printing error yet to be resolved. **IT IS NOT A MINUS SYMBOL,** ignore it.

8.1.Appendix: <u>Using Parameters to speed Sensitivity Analysis</u>

Sensitivity analysis involves repeating optimisations whilst changing certain parameters. When doing this it is recommended to use .Form_Model() only once, and then to update the model parameters between .Run_Sizing(). This reduces run times significantly as model formation can take a long time, where as parameter updating is instantaneous and achieves the same results.

Here are two demonstrative examples to explore how changing the cost of solar generation impacts the optimal system. The first is the 'naive' method of reforming the entire model every time.

Initialise LinProg Model (saved as Variable_Parameters.py on Github):

```python
from generation import (OffshoreWindModel,SolarModel)
import aggregatedEVs as aggEV
from opt_con_class import (System_LinProg_Model)
from storage import (BatteryStorageModel, HydrogenStorageModel,
                     MultipleStorageAssets)
import numpy as np
from fns import get_GB_demand
import pandas as pd

ymin = 2012
ymax = 2016

#Define the generators
osw_master = OffshoreWindModel(year_min=ymin, year_max=ymax,
                              sites=[119,174,178,209,364], data_path='data/150m/')

s = SolarModel(year_min=ymin, year_max=ymax, sites=[17,23,24],
                        data_path='data/solar/')
generators = [s,osw_master]

#Define the Storage
H = HydrogenStorageModel()
B = BatteryStorageModel()
storage = [B,H]


#Define Demand
demand = np.asarray(get_GB_demand(ymin,ymax,list(range(1,13)),False,False))


SolarCost = [40000,60000,80000]
Capital_Record = []

x = System_LinProg_Model(surplus = -demand,fossilLimit = 0.01,Mult_Stor = MultipleStorageAssets(storage),
                         Mult_aggEV = aggEV.MultipleAggregatedEVs([]), gen_list = generators,YearRange = [ymin,ymax])
```

Run Sensitivity on cost of solar power using naïve repeated Use of .Form_Model():

```python
#### Naive Method ####

for i in range(len(SolarCost)):
    s.fixed_cost = SolarCost[i]
    x.Form_Model()
    x.Run_Sizing()
    Capital_Record.append(x.df_capital)

Capital_Record = pd.concat(Capital_Record, ignore_index=True)
Capital_Record['Solar Cost (£/MW/yr)'] = SolarCost
Capital_Record.to_csv('log/SolPrice.csv', index=False)
```

[out]:

```
Forming Optimisation Model...
Model Formation Complete after:  203 s
Finding Optimal System ...
Solved after:  378 s
Forming Optimisation Model...
Model Formation Complete after:  339 s
Finding Optimal System ...
Solved after:  104 s
Forming Optimisation Model...
Model Formation Complete after:  336 s
Finding Optimal System ...
Solved after:  112 s
```

Total time = 24.5 min

Form Model	t = 24.5 min						
Total Demand (GWh)	Total Fossil Fuel (GWh)	Total Curtailement (GWh)	Gen 0 Cap (GW)	Gen 1 Cap (GW)	Stor 0 Cap (GWh)	Stor 1 Cap (GWh)	Solar Cost (¬£/MW/yr)
1501862	15018	313791	104.09	50.09	169.37	28639.1	40000
1501862	15018	271205	85.52	53.25	167.18	29857.37	60000
1501862	15018	260749	68.66	57.72	133.49	34379.05	80000

Run Sensitivity on cost of solar power via updating parameters:

```python
#### Fast Method ####
x.Form_Model()

for i in range(len(SolarCost)):
    x.model.GenCosts[0,0] = SolarCost[i]  #First 0 refers to solar generator
                                         #Second 0 refers to fixed_cost
    x.Run_Sizing()
    Capital_Record.append(x.df_capital)

Capital_Record = pd.concat(Capital_Record, ignore_index=True)
Capital_Record['Solar Cost (£/MW/yr)'] = SolarCost
Capital_Record.to_csv('log/SolPrice_Fast.csv', index=False)

```

[out]:

```
Forming Optimisation Model...
Model Formation Complete after:  182 s
Finding Optimal System ...
Solved after:  397 s
Finding Optimal System ...
Solved after:  92 s
Finding Optimal System ...
Solved after:  127 s
```

Total time = 13.3 min

Update Parameter	t = 13.3 min						
Total Demand (GWh)	Total Fossil Fuel (GWh)	Total Curtailement (GWh)	Gen 0 Cap (GW)	Gen 1 Cap (GW)	Stor 0 Cap (GWh)	Stor 1 Cap (GWh)	Solar Cost (¬£/MW/yr)
1501862	15018	313791	104.09	50.09	169.37	28639.1	40000
1501862	15018	271205	85.52	53.25	167.18	29857.37	60000
1501862	15018	260750	68.66	57.72	133.49	34379.05	80000

Comments:

There is some stochasticity in the solve and form times of the model, but this example shows that by not reforming the model for each case study, the **exact same results are achieved in 52% of the time**. Thus it is always recommended to reset the parameters alone if possible.

Below is an exhaustive list of the parameters that can be adjusted without reforming the model. On Github the script Parameter_Change_Examples.py gives further examples of using these parameters.

Parameter	**Index and Units**	**Description**
Limits		
Gen_Limit_Param_Lower	[len(gen_list)] (MW)	Limits the minimum capacity of generator g to be built.
Gen_Limit_Param_Upper	[len(gen_list)] (MW)	Limits the maximum amount of generator g to be built. If want to fix the generator capacity then make this = lower limit.
Stor_Limit_Param_Lower	[Mult_Stor.n_assets] (MWh)	Limits the minimum capacity of storage type s.
Stor_Limit_Param_Upper	[Mult_Stor.n_assets] (MWh)	Limits the maximum capacity of storage type s.
V2G_Limit_Param_Lower	[Mult_Fleet.n_assets] (number)	Limits the minimum number of V2G chargers for fleet k.
V2G_Limit_Param_Upper	[Mult_Fleet.n_assets] (number)	Limits the max number of V2G chargers for fleet k.

Uni_Limit_Param_Lower	[Mult_Fleet.n_assets] (number)	Limits the min number of unidirectional chargers for fleet k.
Uni_Limit_Param_Upper	[Mult_Fleet.n_assets] (number)	Limits the max number of unidirectional chargers for fleet k.
Costs		
GenCosts	[len(gen_list),2] (£/MW/yr, £/MWh)	If g is the reference number of the generator of interest within gen_list: GenCosts[g,0] – Fixed cost of installation GenCosts[g,1] – Marginal cost of energy production
StorCosts	[Mult_Stor.n_assets,2] (£/MWh/yr, £/MWh)	If i is the reference number of the storage unit of interest within Mult_Stor: StorCosts[i,0] – Fixed cost of installation StorCosts[i,1] – Marginal cost of energy production
chargercost	[Mult_Fleet.n_assets,2] (£/V2G_charger/yr, £/Uni_charger/yr)	chargercost[k,0] – cost of V2G charger fleet k chargercost[k,1] – cost of unidirectional charger fleet k.
Fossil Fuel Use		
foss_lim_param	(float) (MWh)	This is the maximum amount of fossil fuel energy that can be used over the simulation. The usual form of this would be something like = 0.02 * sum(demand)