OPEN Documentation

Release 1.1.0

Energy and Power Group, University of Oxford

Jul 16, 2020

Contents

OPEN provides a toolset for modelling, simulation and optimisation of smart local energy systems. The platform combines distributed energy resource modelling (e.g. for PV generation sources, battery energy storage systems, electric vehicles), energy market modelling, power flow simulation and multi-period optimisation for scheduling flexible energy resources.

The OPEN code is available on GitHub here.

CHAPTER 1

Overview

Oxford University's Energy and Power Group's Open Platform for Energy Networks (OPEN) provides a python toolset for modelling, simulation and optimisation of smart local energy systems. The framework combines distributed energy resource modelling (e.g. for PV generation sources, battery energy storage systems, electric vehicles), energy market modelling, power flow simulation and multi-period optimisation for scheduling flexible energy resources.

OPEN and the methods used are presented in detail in the following publication:

T. Morstyn, K. Collett, A. Vijay, M. Deakin, S. Wheeler, S. M. Bhagavathy, F. Fele and M. D. McCulloch; *"An Open-Source Platform for Developing Smart Local Energy System Applications"*; University of Oxford Working Paper, 2019

1.1 Installation

Download OPEN source code here.

If using conda, we suggest creating a new virtual environment from the requirements.txt file. First, add the following channels to your conda distribution if not already present:

```
conda config --add channels invenia
conda config --add channels picos
conda config --add channels conda_forge
```

To create the new virtual environment, run:

```
conda create --name <env_name> --file requirements.txt python=3.6
```

1.2 Getting started

The simplest way to start is to duplicate one of the case study main.py files: - OxEMF_EV_case_study_v6.py - Main_building_casestudy.py

1.3 Dependencies

- Pandas
- scipy
- cvxopt
- scikit-learn
- Pandapower
- Numpy
- Picos = 1.1.2
- Matplotlib
- numba
- requests

1.4 Platform Structure

OPEN is implemented in Python using an object orientated programming approach, with the aim of providing modularity, code reuse and extensibility. Fig. 1 shows a universal modelling language (UML) class diagram of OPEN. OPEN has four important base classes: Asset, Network, Market and EnergySystem.

Fig. 1 - UML class diagram of OPEN, showing the main classes, attributes and methods.

OPEN includes two energy management system (EMS) methods for controllable Asset objects: (i) one for multi-period optimisation with a simple 'copper plate' network model, and (ii) the other for multi-period optimisation with a linear multi-phase distribution network model which includes voltage and current flow constraints. OPEN has simulation methods for: (i) open-loop optimisation, where the EMS method is run ahead of operation to obtain controllable Asset references over the EMS time series; and (ii) for model predictive control (MPC), where the EMS method is implemented with a receding horizon so that the flexible Asset references are updated at each step of the EMS time-series. Fig. 2 shows a high-level program flow diagram for an example MPC OPEN application.

Fig. 2 - High-level program flow for an MPC OPEN application.

1.4.1 Energy System

In OPEN, a smart local energy system application is built around an EnergySystem object.

The EnergySystem Class has two types of methods i) energy management system (EMS) methods which implement algorithms to calculate Asset control references, and ii) simulation methods which call an EMS method to obtain control references for Asset objects, update the state of Asset objects by calling their update control method and update the state of the Network by calling its power flow method. An EnergySystem has two separate time series, one for the EMS, and the other for simulation.

The EnergySystem class can be extended by defining new EMS methods. For example, new EMS methods could be used to implement more advanced non-convex optimisation strategies, or market-based scheduling with distributed optimisation and P2P negotiation. The requirement for interoperability is that the Asset references are returned by the EMS method as a dictionary that can be read by simulation methods which call it.

1.4.2 Assets

An Asset object define DERs and loads. Attributes include network location, phase connection and real and reactive output power profiles over the simulation time-series. Flexible Asset classes have an update control method, which is called by EnergySystem simulation methods with control references to update the output power profiles and state variables. The update control method also implements constraints which limit the implementation of references. OPEN includes the following Asset subclasses: NondispatchableAsset for uncontrollable loads and generation sources, StorageAsset for storage systems and BuildingAsset for buildings with flexible heating ventilation and air conditioning (HVAC).

New Asset subclasses can be defined which inherit the attributes from other Asset classes, but may have additional attributes and different update control method implementations.

1.4.3 Markets

A Market class defines an upstream market which the EnergySystem is connected to. Attributes include the network location, prices of imports and exports over the simulation time-series, the demand charge paid on the maximum demand over the simulation time-series and import and export power limits.

The market class has a method which calculates the total revenue associated with a particular set of real and reactive power profiles over the simulation time-series.

1.4.4 Networks

OPEN offers two options for network modelling. For balanced power flow analysis, the PandapowerNet class from the open-source python package pandapower can be used. For unbalanced multi-phase power flow analysis, OPEN offers the Network_3ph class.

The PandapowerNet class offers methods for balanced nonlinear power flow using a Netwon-Raphson solution method, and balanced linear power flow based on the DC approximation. OPEN's Network_3ph class offers nonlinear multi-phase power flow using the Z-Bus method, as well as linear multi-phase power flow using the fixed-point linearisation. Wye and delta connected constant power loads/sources, constant impedance loads and capacitor banks can be modelled. Lines are modelled as π -equivalent circuits. Transformers with any combination of wye, wye-grounded or delta primary and secondary connections can also be modelled. Features that are planned to be added in future include voltage regulators and constant current loads.

1.5 License

For academic and professional use, please provide attribution to the papers describing OPEN.[1]

1.6 References

[1] T. Morstyn, K. Collett, A. Vijay, M. Deakin, S. Wheeler, S. M. Bhagavathy, F. Fele and M. D. McCulloch; *"An Open-Source Platform for Developing Smart Local Energy System Applications"*; University of Oxford Working Paper, 2019

CHAPTER 2

OPEN API Reference

2.1 Assets

OPEN Asset module

Asset objects define distributed energy resources (DERs) and loads. Attributes include network location, phase connection and real and reactive output power profiles over the simulation time-series. Flexible Asset classes have an update control method, which is called by EnergySystem simulation methods with control references to update the output power profiles and state variables. The update control method also implements constraints which limit the implementation of references. OPEN includes the following Asset subclasses: NondispatchableAsset for uncontrollable loads and generation sources, StorageAsset for storage systems and BuildingAsset for buildings with flexible heating, ventilation and air conditioning (HVAC).

class System.Assets.**Asset**(*bus_id, dt, T, phases=[0, 1, 2]*)

An energy resource located at a particular bus in the network

Parameters

- **bus_id** (*float*) – id number of the bus in the network
- **dt** (*float*) – time interval duration
- **T** (*int*) – number of time intervals
- **phases** (*list, optional, default [0,1,2]*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a

Returns

Return type *Asset*

class System.Assets.**Asset_3ph**(*bus_id*, *phases*, *dt*, *T*)

An energy resource located at a particular bus in the 3 phase network

Parameters

- **bus_id** (*float*) – id number of the bus in the network

- **phases** (*list*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **dt** (*float*) – time interval duration
- **T** (*int*) – number of time intervals

Returns

Return type *Asset*

class System.Assets.**BuildingAsset**(*Tmax*, *Tmin*, *Hmax*, *Cmax*, *T0*, *C*, *R*, *CoP_heating*, *CoP_cooling*, *Ta*, *bus_id*, *dt*, *T*, *dt_ems*, *T_ems*)

A building asset (use for flexibility from building HVAC)

Parameters

- **Tmax** (*float*) – Maximum temperature inside the building (Degree C)
- **= float** (*Tmin*) – Minimum temperature inside the building (Degree C)
- **Hmax** (*float*) – Maximum power consumed by electrical heating (kW)
- **Cmax** (*float*) – Maximum power consumed by electrical cooling (kW)
- **deltat** (*float*) – Time interval after which system is allowed to change decisions (h)
- **T0** (*float*) – Initial temperature inside the buidling (Degree C)
- **C** (*float*) – Thermal capacitance of building (kWh/Degree C)
- **R** (*float*) – Thermal resistance of building to outside environment(Degree C/kW)
- **CoP_heating** (*float*) – Coefficient of performance of the heat pump (N/A)
- **CoP_cooling** (*float*) – Coefficient of performance of the chiller (N/A)
- **Ta** (*numpy.ndarray*) – Ambient temperature (Degree C)
- **alpha** (*float*) – Coefficient of previous temperature in the temperature dynamics equation (N/A)
- **beta** (*float*) – Coefficient of power consumed to heat/cool the building in the temperature dynamics equation (Degree C/kW)
- **gamma** (*float*) – Coefficient of ambient temperature in the temperature dynamics equation (N/A)
- **Pnet** (*numpy.ndarray*) – Input real power (kW)
- **Qnet** (*numpy.ndarray*) – Input reactive power (kVAR)

Returns

Return type *Asset*

update_control(*Pnet*)

Update the power consumed by the HVAC at time interval t

Parameters **Pnet** (*numpy.ndarray*) – input powers over the time series (kW)

class System.Assets.**NondispatchableAsset**(*Pnet, Qnet, bus_id, dt, T, phases=[0, 1, 2], Pnet_pred=None, Qnet_pred=None*)

A 3 phase nondispatchable asset class (use for inflexible loads, PVsources etc)

Parameters

- **Pnet** (*float*) – uncontrolled real input powers over the time series

- **Qnet** (*float*) – uncontrolled reactive input powers over the time series (kVar)
- **bus_id** (*float*) – id number of the bus in the network
- **dt** (*float*) – time interval duration
- **T** (*int*) – number of time intervals
- **phases** (*list, optional, default [0,1,2]*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **Pnet_pred** (*float or None*) – predicted real input powers over the time series (kW)
- **Qnet_pred** (*float or None*) – predicted reactive input powers over the time series (kVar)

Returns

Return type *Asset*

class System.Assets.**NondispatchableAsset_3ph**(*Pnet*, *Qnet*, *bus_id*, *phases*, *dt*, *T*, *Pnet_pred=None*, *Qnet_pred=None*)

A 3 phase nondispatchable asset class (use for inflexible loads, PVsources etc)

Parameters

- **Pnet** (*float*) – uncontrolled real input powers over the time series
- **Qnet** (*float*) – uncontrolled reactive input powers over the time series (kVar)
- **bus_id** (*float*) – id number of the bus in the network
- **phases** (*list*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **dt** (*float*) – time interval duration
- **T** (*int*) – number of time intervals
- **Pnet_pred** (*float*) – predicted real input powers over the time series (kW)
- **Qnet_pred** (*float*) – predicted reactive input powers over the time series (kVar)

Returns

Return type *Asset*

class System.Assets.**StorageAsset**(*Emax, Emin, Pmax, Pmin, E0, ET, bus_id, dt, T, dt_ems, T_ems, phases=[0, 1, 2], Pmax_abs=None, c_deg_lin=None, eff=1, eff_opt=1*)

A storage asset (use for batteries, EVs etc.)

Parameters

- **Emax** (*numpy.ndarray*) – maximum energy levels over the time series (kWh)
- **Emin** (*numpy.ndarray*) – minimum energy levels over the time series (kWh)
- **Pmax** (*numpy.ndarray*) – maximum input powers over the time series (kW)
- **Pmin** (*numpy.ndarray*) – minimum input powers over the time series (kW)
- **E0** (*float*) – initial energy level (kWh)
- **ET** (*float*) – required terminal energy level (kWh)
- **bus_id** (*float*) – id number of the bus in the network
- **dt** (*float*) – time interval duration (s)

- **T** (*int*) – number of time intervals
- **dt_ems** (*float*) – time interval duration (energy management system time horizon) (s)
- **T_ems** (*int*) – number of time intervals (energy management system time horizon)
- **phases** (*list, optional, default [0,1,2]*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **Pmax_abs** (*float*) – max power level (kW)
- **c_deg_lin** (*float*) – battery degradation rate with energy throughput (£/kWh)
- **eff** (*float, default 1*) – charging efficiency (between 0 and 1)
- **eff_opt** (*float, default 1*) – charging efficiency to be used in optimiser (between 0 and 1).
- **Pnet** (*numpy.ndarray*) – Input real power over the simulation time series (kW)
- **Qnet** (*numpy.ndarray*) – Input reactive power over the simulation time series(kVAR)

Returns

Return type *Asset*

update_control(*Pnet*)

Update the storage system power and energy profile

Parameters **Pnet** (*float*) – input powers over the time series (kW)

update_control_t(*Pnet_t*, *t*)

Update the storage system power and energy at time interval t

Parameters

- **Pnet_t** (*float*) – input powers over the time series (kW)
- **t** (*int*) – time interval

class System.Assets.**StorageAsset_3ph**(*Emax*, *Emin*, *Pmax*, *Pmin*, *E0*, *ET*, *bus_id*, *phases*, *dt*, *T*, *dt_ems*, *T_ems*, *Pmax_abs=None*, *c_deg_lin=None*, *eff=1*, *eff_opt=1*)

An 3 phase storage asset (use for batteries, EVs etc.)

Parameters

- **Emax** (*numpy.ndarray*) – maximum energy levels over the time series (kWh)
- **Emin** (*numpy.ndarray*) – minimum energy levels over the time series (kWh)
- **Pmax** (*numpy.ndarray*) – maximum input powers over the time series (kW)
- **Pmin** (*numpy.ndarray*) – minimum input powers over the time series (kW)
- **E0** (*float*) – initial energy level (kWh)
- **ET** (*float*) – required terminal energy level (kWh)
- **bus_id** (*float*) – id number of the bus in the network
- **phases** (*list, optional, default [0,1,2]*) – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **dt** (*float*) – time interval duration (s)
- **T** (*int*) – number of time intervals
- **dt_ems** (*float*) – time interval duration (energy management system time horizon) (s)

- **T_ems** (*int*) – number of time intervals (energy management system time horizon)
- **phases** – [0, 1, 2] indicates 3 phase connection Wye: [0, 1] indicates an a,b connection Delta: [0] indicates a-b, [1] b-c, [2] c-a
- **Pmax_abs** (*float*) – max power level (kW)
- **c_deg_lin** (*float*) – battery degradation rate with energy throughput (£/kWh)
- **eff** (*float, default 1*) – charging efficiency (between 0 and 1).
- **eff_opt** (*float, default 1*) – charging efficiency to be used in optimiser (between 0 and 1).

Returns

Return type *Asset*

update_control(*Pnet*)
Update the storage system power and energy profile

Parameters **Pnet** (*numpy.ndarray*) – input powers over the time series (kW)

update_control_t(*Pnet_t*, *t*)
Update the storage system power and energy at time interval t

Parameters

- **Pnet_t** (*numpy.ndarray*) – input powers over the time series (kW)
- **t** (*int*) – time interval

2.2 Energy System

OPEN Energy System Module.

The EnergySystem Class has two types of methods i) energy management system (EMS) methods which implement algorithms to calculate Asset control references, and ii) simulation methods which call an EMS method to obtain control references for Asset objects, update the state of Asset objects by calling their updatecontrol method and update the state of the Network by calling its power flow method. An EnergySystem has two separate time series, one for the EMS, and the other for simulation.

OPEN includes two EMS methods for controllable Asset objects: (i) one for multi-period optimisation with a simple 'copper plate' network model, and (ii) one for multi-period optimisation with a linear multi-phase distribution network model which includes voltage and current flow constraints.

Open has simulation methods for: (i) open-loop optimisation, where the EMS method is run ahead of operation to obtain controllable Asset references over the EMS time-series; and (ii) for MPC, where the EMS method is implemented with a receding horizon so that the flexible Asset references are updated at each step of the EMS time series.

class System.EnergySystem.**EnergySystem**(*storage_assets*, *nondispatch_assets*, *network*, *market*, *dt*, *T*, *dt_ems*, *T_ems*, *building_assets=[]*)
Base Energy Sysem Class

Parameters

- **storage_assets** (*list of objects*) – Containing details of each storage asset
- **building_assets** (*list of objects*) – Containsing details of each building asset
- **nondispatch_assets** (*list of objects*) – Containsing details of each nondispatchable asset

- **network** (*object*) – Object containing information about the network
- **market** (*object*) – Object containing information about the market
- **dt_ems** (*float*) – EMS time interval duration (hours)
- **T_ems** (*int*) – Number of EMS time intervals
- **dt** (*float*) – time interval duration (hours)
- **T** (*int*) – number of time intervals

Returns

Return type *EnergySystem*

EMS_3ph_linear_t0(*t0*, *i_unconstrained_lines=[]*, *v_unconstrained_buses=[]*)

Energy management system optimization assuming 3 phase linear network model for Model Predictive Control interval t0

Parameters

- **self** (*EnergySystem object*) – Object containing information on assets, market, network and time resolution.
- **t0** (*int*) – Interval in Model Predictive Control. If open loop, t0 = 0
- **i_unconstrained_lines** (*list*) – List of network lines which have unconstrained current
- **v_unconstrained_buses** (*list*) – List of buses at which the voltage is not constrained

Returns

Output –

The following numpy.ndarrays are present depending upon asset mix: P_ES_val : Charge/discharge power for storage assets (kW) P_import_val : Power imported from central grid (kW) P_export_val : Power exported to central grid (kW) P_demand_val : System power demand at energy management time

resolution (kW)

PF_networks_lin [Network 3ph list of objects, one for each] optimisation interval, storing the linear power flow model used to formulate netowrk constraints

Return type dictionary

EMS_copper_plate()

Energy management system optimization assuming all assets connected to a single node.

Parameters **self** (*EnergySystem object*) – Object containing information on assets, market, network and time resolution.

Returns

Output –

The following numpy.ndarrays are present depending upon asset mix: P_ES_val : Charge/discharge power for storage assets (kW) P_BLDG_val :Builfing power consumption (kW) P_import_val :Power imported from central grid (kW) P_export_val :Power exported to central grid (kW) P_demand_val :System power demand at energy management time

resolution

Return type dictionary

EMS_copper_plate_t0(*t0*)
Setup and run a basic energy optimisation (single copper plate network model) for MPC interval t0

EMS_copper_plate_t0_c1deg(*t0*)
setup and run a basic energy optimisation (single copper plate network model) for MPC interval t0

simulate_network()
Run the Energy Management System in open loop and simulate a pandapower network.

Parameters **self** (*EnergySystem object*) – Object containing information on assets, market, network and time resolution.

Returns

Output –

The following numpy.ndarrays are present depending upon asset mix: buses_Vpu : Voltage magnitude at bus (V) buses_Vang : Voltage angle at bus (rad) buses_Pnet : Real power at bus (kW) buses_Qnet : Reactive power at bus (kVAR) Pnet_market : Real power seen by the market (kW) Qnet_market : Reactive power seen by the market (kVAR) P_ES_ems : Charge/discharge power for storage assets at energy

management time resolution (kW)

P_BLDG_ems :Builfing power consumption at energy management time resolution (kW)

P_import_ems :Power imported from central grid at energy management time resolution (kW)

P_export_ems :Power exported to central grid at energy management time resolution(kW)

P_demand_ems :System power demand at energy management time resolution (kW)

Return type dictionary

simulate_network_3phPF(*ems_type='3ph'*, *i_unconstrained_lines=[]*, *v_unconstrained_buses=[]*)
Run the Energy Management System in open loop and simulate an IEEE 13 bus network either copper plate or 3ph

Parameters

- **self** (*EnergySystem object*) – Object containing information on assets, market, network and time resolution.
- **ems_type** (*string*) – Identifies whether the system is copper plate or 3ph. Default 3ph
- **i_unconstrained_lines** (*list*) – List of network lines which have unconstrained current
- **v_unconstrained_buses** (*list*) – List of buses at which the voltage is not constrained

Returns

Output –

PF_network_res [Network power flow results stored as a list of] objects

P_ES_ems [Charge/discharge power for storage assets at energy] management time resolution (kW)

P_import_ems :Power imported from central grid at energy management time resolution (kW)

P_export_ems :Power exported to central grid at energy management time resolution(kW)

P_demand_ems :System power demand at energy management time resolution (kW)

Return type dictionary

simulate_network_3phPF_lean(*ems_type='3ph'*)
run the EMS in open loop and simulate a 3-phase AC network

simulate_network_mpc_3phPF(*ems_type='3ph'*, *i_unconstrained_lines=[]*, *v_unconstrained_buses=[]*)
Run the Energy Management System using Model Predictive Control (MPC) and simulate an IEEE 13 bus network either copper plate or 3ph

Parameters

- **self** (*EnergySystem object*) – Object containing information on assets, market, network and time resolution.
- **ems_type** (*string*) – Identifies whether the system is copper plate or 3ph. Default 3ph
- **i_unconstrained_lines** (*list*) – List of network lines which have unconstrained current
- **v_unconstrained_buses** (*list*) – List of buses at which the voltage is not constrained

Returns

Output –

PF_network_res [Network power flow results stored as a list of] objects

P_ES_ems [Charge/discharge power for storage assets at energy] management time resolution (kW)

P_import_ems :Power imported from central grid at energy management time resolution (kW)

P_export_ems :Power exported to central grid at energy management time resolution(kW)

P_demand_ems :System power demand at energy management time resolution (kW)

Return type dictionary

2.3 Markets

OPEN Markets module

A Market class defines an upstream market which the EnergySystem is connected to. Attributes include the network location, prices of imports and exports over the simulation time-series, the demand charge paid on the maximum demand over the simulation time-series and import and export power limits.

The market class has a method which calculates the total revenue associated with a particular set of real and reactive power profiles over the simulation time-series.

class System.Markets.**Market**(*bus_id*, *prices_export*, *prices_import*, *demand_charge*, *Pmax*, *Pmin*, *dt_market*, *T_market*, *FR_window=None*, *FR_capacity=None*, *FR_SOC_max=0.6*, *FR_SOC_min=0.4*, *FR_price=0.005*, *stochastic_date=None*, *daily_connection_charge=0.13*)

A market class to handle prices and other market associated parameters.

Parameters

- **bus_id** (*int*) – id number of the bus in the network
- **prices_export** (*numpy.ndarray*) – price paid for exports (£/kWh)
- **prices_import** (*numpy.ndarray*) – price charged for imports (£/kWh)
- **demand_charge** (*float*) – charge for the maximum demand over the time series (£/kWh)
- **Pmax** (*float*) – maximum import power over the time series (kW)
- **Pmin** (*float*) – minimum import over the time series (kW)
- **dt_market** (*float*) – time interval duration (minutes)
- **T_market** (*int*) – number of time intervals
- **FR_window** (*int*) – binary value over time series to indicate when frequency response has been offered (0,1)
- **FR_capacity** (*float*) – capacity of frequency response offered (kW)
- **FR_SOC_max** (*float*) – max SOC at which frequency response can still be fulfilled if needed
- **FR_SOC_min** (*float*) – min SOC at which frequency response can still be fulfilled if needed
- **FR_price** (*float*) – price per kW capacity per hour avaiable (£/kW.h)

Returns

Return type *Market*

calculate_revenue(*P_import_tot*, *dt*)

Calculate revenue according to simulation results

Parameters

- **P_import_tot** (*float*) – Total import power to the site over the time series (kW)
- **dt** (*float*) – simulation time interval duration (minutes)
- **c_deg_lin** (*float*) – cost of battery degradation associated with each kWh throughput (£/kWh)

Returns **revenue** – Total revenue generated during simulation

Return type float

2.4 Networks

OPEN 3 phase networks module

OPEN offers two options for network modelling. For balanced power flow analysis, the PandapowerNet class from the open-source python package pandapower can be used. For unbalanced multi-phase power flow analysis, OPEN offers the Network_3ph class.

class System.Network_3ph_pf.**Network_3ph**

A 3-phase electric power network. Default to an unloaded IEEE 13 Bus Test Feeder.

Parameters

- **bus_df** (*pandas.DataFrame*) – bus information, columns: ['name','number','load_type','connect', 'Pa','Pb','Pc','Qa','Qb','Qc'], load_type: 'S' (slack),'PQ','Z' or 'I' (only S and PQ are currently implemented), connect: 'Y' (wye) or 'D' (delta), 'Px' in (kW), 'Qx' in (kVAr)
- **capacitor_df** (*pandas.DataFrame*) – capacitor information, columns ['name','number','bus','kVln', 'connect','Qa','Qb','Qc'], connect: 'Y' (wye) or 'D' (delta), 'kVln': line-to-line base voltage, 'Qx' in (kVAr)
- **di_iter** (*np.ndarray*) – change in the sum of abs phase currents at each Z-Bus iteration
- **dv_iter** (*np.ndarray*) – change in the sum of abs phsae voltages at each Z-Bus iteration
- **i_abs_max** (*numpy.ndarray*) – max abs line phase currents [line, phase] (|A|)
- **i_PQ_iter** (*numpy.ndarray*) – current injected at each phase at each Z-Bus iteration [iter,phase] (A)
- **i_PQ_res** (*numpy.ndarray*) – power flow result, current injected at each phase (excl. slack) (A)
- **i_net_res** (*numpy.ndarray*) – power flow result, current injected at each phase (A)
- **i_slack_res** (*numpy.ndarray*) – power flow result, current injected at slack bus phases (A)
- **Jabs_dPQdel_list** (*list of numpy.ndarray*) – linear line abs current model, [P_delta,Q_delta] coeff. matrix list
- **Jabs_dPQwye_list** (*list of numpy.ndarray*) – linear line abs current model, [P_wye,Q_wye] coeff. matrix list
- **Jabs_I0_list** (*list of numpy.ndarray*) – linear line abs current model, constant vector list
- **J_dPQdel_list** (*list of numpy.ndarray*) – linear line current model, [P_delta,Q_delta] coeff. matrix list
- **J_dPQwye_list** (*list of numpy.ndarray*) – linear line current model, [P_wye,Q_wye] coeff. matrix list
- **J_I0_list** (*list of numpy.ndarray*) – linear line current model, constant vector list
- **K_del** (*numpy.ndarray*) – linear abs voltage model, [P_delta,Q_delta] coeff. matrix
- **K_wye** (*numpy.ndarray*) – linear abs voltage model, [P_wye,Q_wye] coeff. matrix
- **K0** (*numpy.ndarray*) – linear abs voltage model, constant vector
- **line_config_df** (*pandas.DataFrame*) – information on line configurations, columns: ['name','Zaa',

'Zbb','Zcc','Zab','Zac','Zbc','Baa','Bbb','Bcc','Bab','Bac','Bbc'], 'Zxx' in (Ohms/length unit), 'Bxx' in (S/length unit) [base voltage of Vslack]

- **line_df** (*pandas.DataFrame*) – line information: ['busA','busB','Zaa','Zbb','Zcc','Zab','Zac','Zbc', 'Baa','Bbb','Bcc','Bab','Bac','Bbc'], 'Zxx' in (Ohms) 'Bxx' in (S) [base voltage of Vslack]
- **line_info_df** – matches lines to their configurations: ['busA','busB','config', 'length','Z_conv_factor','B_conv_factor'], Z,B_conv_factors used to match 'lenth' units to line_config_df 'Zxx' and 'Bxx' values
- **list_Yseries** (*list of numpy.ndarray*) – list of series admittance matrices for the lines
- **list_Yshunt** (*list of numpy.ndarray*) – list of shunt admittance matrices for the lines
- **M_del** (*numpy.ndarray*) – linear voltage model, [P_delta,Q_delta] coeff. matrix
- **M_wye** (*numpy.ndarray*) – linear voltage model, [P_wye,Q_wye] coeff. matrix
- **M0** (*numpy.ndarray*) – linear abs voltage model, constant vector
- **N_buses** (*int*) – number of buses
- **N_capacitors** (*int*) – number of capacitors
- **N_iter** (*int*) – number of Z-Bus power flow solver iterations
- **N_phases** (*int*) – number of phases
- **N_lines** (*int*) – number of lines
- **N_transformers** (*int*) – number of transformers
- **res_bus_df** (*pandas.DataFrame*) – power flow result, bus information, columns: ['name','number', 'load_type','connect','Sa','Sb','Sc','Va','Vb','Vc','Ia','Ib','Ic'] load_type: 'S' (slack),'PQ','Z' or 'I' (only S and PQ are currently implemented), connect: 'Y' (wye) or 'D' (delta), 'Sx' in (kVA), 'Vx' in (V), 'Ix' in (A)
- **res_lines_df** (*pandas.DataFrame*) – power flow result, line information, columns: ['busA','busB','Sa','Sb', 'Sc','Ia','Ib','Ic','VAa','VAb','VAc','VBa','VBb','VBc'], 'Sx' in (VA), 'Ix' in (A), bus A voltages 'VAx' in (V), bus B voltages 'VBx' in (V).
- **S_del_lin0** (*numpy.ndarray*) – linear model, delta apparent power load (kVA)
- **S_PQloads_del_res** (*numpy.ndarray*) – power flow result, delta apparent power load (kVA)
- **S_PQloads_wye_res** (*numpy.ndarray*) – power flow result, wye apparent power load (kVA)
- **S_net_res** (*pandas.DataFrame*) – power flow result, apparent power load at each phase (VA)
- **S_wye_lin0** – linear model, delta apparent power load (kVA) Y :
- **transformer_df** (*pandas.DataFrame*) – transformer information, columns: ['busA','busB','typeA','typeB', 'Zseries','Zshunt'], typex: 'wye-g', 'wye' or 'delta'
- **v_abs_min** (*numpy.ndarray*) – min abs bus voltages [bus, phase] (|V|)
- **v_abs_max** (*numpy.ndarray*) – max abs bus voltages [bus, phase] (|V|)
- **v_iter** (*numpy.ndarray*) – bus phase voltages at each Z-Bus iteration [iteration, phase]

- **v_lin_abs_res** (*numpy.ndarray*) – linear model result, bus phase abs voltage (excl. slack) (|V|)
- **v_lin_res** (*numpy.ndarray*) – linear model result, bus phase voltage (excl. slack) (V)
- **v_net_lin_abs_res** (*numpy.ndarray*) – linear model result, bus phase abs voltage (|V|)
- **v_net_lin_res** (*numpy.ndarray*) – linear model result, bus phase voltage (V)
- **v_net_lin0** (*numpy.ndarray*) – linear model, nominal bus phase voltages (V)
- **v_net_res** (*numpy.ndarray*) – power flow result, bus phase voltages (V)
- **vs** (*numpy.ndarray*) – slack bus phase voltages (V)
- **Vslack** (*float*) – slack bus line-to-line voltage magnitude (|V|)
- **Vslack_ph** (*float*) – slack bus line-to-phase voltage magnitude (|V|)
- **v_res** (*numpy.ndarray*) – power flow result, bus phase voltages (excl. slack) (V)
- **Y** (*numpy.ndarray*) – admittance matrix (excl. slack) (S)
- **Ynet** (*numpy.ndarray*) – admittance matrix (S)
- **Y_non_singular** – admittance matrix with 1e-20*I added (excl. slack) (S) [base voltage of Vslack]
- **Yns** (*numpy.ndarray*) – admittance matrix partition [Yss, Ysn; Yns, Y] (S) [base voltage of Vslack] voltage of Vslack
- **Ysn** (*numpy.ndarray*) – admittance matrix partition [Yss, Ysn; Yns, Y] (S) [base voltage of Vslack]
- **Yss** (*numpy.ndarray*) – admittance matrix partition [Yss, Ysn; Yns, Y] (S) [base voltage of Vslack]
- **Z** – impedance matrix (Ohm) [base voltage of Vslack]
- **Znet** – impedance matrix (excl. slack) (Ohm) [base voltage of Vslack]

Returns

Return type *Network_3ph*

clear_loads()
Removes all real and reactive power loads from the network by clearing bus_df.

linear_model_setup(*v_net_lin0*, *S_wye_lin0*, *S_del_lin0*)
Set up a linear model based on A. Bernstein, et al., "Load Flow in Multiphase Distribution Networks: Existence, Uniqueness, Non-Singularity and Linear Models," IEEE Transactions on Power Systems, 2018.

Parameters

- **v_net_lin0** (*numpy.ndarray*) – nominal operating point, bus phase voltages (V)
- **S_wye_lin0** (*numpy.ndarray*) – nominal operating point, apparent wye power loads (kVA)
- **S_del_lin0** (*numpy.ndarray*) – nominal operating point, apparent delta power loads (kVA)

linear_pf()
Solves the linear model based on A. Bernstein, et al., "Load Flow in Multiphase Distribution Networks:

Existence, Uniqueness, Non-Singularity and Linear Models," IEEE Transactions on Power Systems, 2018. First run linear_model_setup().

set_load(*bus_id*, *ph_i*, *Pph*, *Qph*)
Sets the P and Q load on a particular bus and phase

Parameters

- **bus_id** (*int*) – the load bus id
- **ph_i** (*int*) – the load phase (either 0, 1 or 2)
- **Pph** (*float*) – nominal operating point, apparent wye load (kVA)
- **Qph** (*float*) – nominal operating point, apparent delta load (kVA)

set_pf_limits(*v_abs_min_val*, *v_abs_max_val*, *i_abs_max_val*)
Sets the abs bus phase voltage limits and abs line phase current limits

Parameters

- **v_abs_min_val** (*float*) – minimum abs voltage bus phase voltage limit
- **v_abs_max_val** (*float*) – maximum abs voltage bus phase voltage limit
- **i_abs_max_val** (*float*) – maximum abs line phase current limit

setup_network_ieee13()
Set up the network as the unloaded IEEE 13 Bus Test Feeder

update_YandZ()
Update the network admittance and impedance matrices

update_line_pf_results()
Updates the line power flow results dataframe res_lines_df, based on the results of zbus_pf().

v_flat()
Get the vector of 1 p.u. balanced bus phase voltages

Returns

Return type numpy.ndarray

zbus_pf()
Solves the nonlinear power flow problem using the Z-bus method from M. Bazrafshan, N. Gatsis, "Comprehensive Modeling of Three-Phase Distribution Systems via the Bus Admittance Matrix," IEEE Transactions on Power Systems, 2018.

CHAPTER 3

OPEN Examples

For further details on the example case studies, please refer here.[1]

3.1 Electric Vehicle (EV) Smart Charging

The Electric Vehicle Smart Charging case study considers the smart charging of EVs within an unbalanced three-phase distribution network. The case study considers a business park where 80 EVs are charged at 6.6 kW charge points. The objective is to charge all of the vehicles to their maximum energy level prior to departure, at lowest cost.

3.2 Flexible Heating Ventilation Air Conditioning (HVAC) Demand Side Response (DSR)

The building energy management case study focuses on a building with a flexible HVAC unit which is controlled in order to minimise costs, with the constraint that the internal temperature remains between 16^{o}C and 18^{o}C.

[1] T. Morstyn, K. Collett, A. Vijay, M. Deakin, S. Wheeler, S. M. Bhagavathy, F. Fele and M. D. McCulloch; *"An Open-Source Platform for Developing Smart Local Energy System Applications"*; University of Oxford Working Paper, 2019

CHAPTER 4

Authors

OPEN was developed by the Energy and Power Group, Engineering Science, University of Oxford.

Website

- Thomas Morstyn
- Avinash Vijay
- Katherine Collet
- Filiberto Fele
- Matthew Deakin
- Sivapriya Mothilal Bhagavathy
- Scot Wheeler
- Malcolm McCulloch

Python Module Index

S

Index

V

Z