```
import pyomo.environ as pyo #optimization tool
import numpy as np #data processing
import pandas as pd #data procdssing/ strucure
import itertools
#==================================================== 'mg' parameters
====================================================================
====
mg = {
'RDG': [{'id': 'sp', 'microgrid_id': 'mg1', 'class': 'Solar Panel'}],
'CDG': [{'class': 'FC',
'id': 'fc',
'microgrid_id': 'mg1',
'cost_model_type': 'Quadratic',
'cost_model_parameters': [0.004, 0.066, 0.7],
'P_max': 0.01,
'P_min': 0.0,
'R_max': 0.0}],
'Load': [{'id': 'l1', 'microgrid_id': 'mg1', 'λ': 0.5, 'α_NCL': 0}],
'ESS': [{'id': 'bat1',
'microgrid_id': 'mg1',
'class': 'Lithium Battery',
'γch': 0.3, #cost of charging
'γdis': 0.3, #cost of discharging
'Pch_min': 0.0,
'Pch_max': 203.13,
48'Pdis_min': 0.0,
'Pdis_max': 308.9,
'SOC_min': 0.2,
'SOC_max': 1,
'E_cap': 1240.0, ##################################
'ηch': 0.95,
'ηdis': 0.95}],
'Microgrid': [{'id': 'mg1', 'T': 48, 'Δt': 30, 'L_PWF': 10}],
'Grid': [{'id': 'ug1',
```

```
'Pb_min': 0.0,
'Pb_max': 10000000.0,
'Ps_min': 0.0,
'Ps_max': 10000000.0}],
'system': {'userId': '1'}}
#====================================== MILP Mixed Integer Linear
Programming
====================================================================
=
def build_MILP_model(mg:dict = mg, SOC_init: list = [0.2], price_buy: list = [1], price_sell:
list = [0],
P_RDG: list = [0], P_LD: list = [0], ESS_recover: bool = False, E_org: list = 1):
for (i,ess) in enumerate(mg['ESS']):
# Cater for minor offset
assert(ess['SOC_min'] - 1e-3 <= SOC_init[i] <= ess['SOC_max'] + 1e-3), 'SOC_init out of
bound.'#1e-3 to cater for minor errors
# Reshape the lists
49price_buy_1d = list(itertools.chain(*price_buy)) #flatten data to [1,2,3,4,5,6,7]
price_sell_1d = list(itertools.chain(*price_sell))
P_RDG_1d = list(itertools.chain(*P_RDG))
P_LD_1d = list(itertools.chain(*P_LD))
# Validate number - Utility Grid
assert(len(price_buy) == len(price_sell)), 'Number of price data does not match.'
# assert all(x > 0 for x in price_buy)
assert all(x >= 0 for x in price_buy_1d), 'Buying price is invalid.' #Buying price is invalid
# Validate number - RDG(Renewable Distributed Generation): PV
assert all(x >= 0.0 for x in P_RDG_1d), 'RDG power supply is negative.' #invalid
# Validate number - Load
assert all(x >= 0.0 for x in P_LD_1d), 'Load demand is negative.' #invalid
# Validate time horizon
T = np.array(price_buy).shape[1] #counting number of columns
#check if the number of columns of comparing data are equal
assert(T == np.array(price_sell).shape[1]), 'Time horizon of buying/selling price does not
match.'
```

```
assert(T == np.array(P_RDG).shape[1]), 'Time horizon of RDG data does not match.'
assert(T == np.array(P_LD).shape[1]), 'Time horizon of Load data does not match.'
# Validate TOML input
50n_grid = len(mg["Grid"])
n_load = len(mg["Load"])
n_RDG = len(mg["RDG"])
n_bat = len(mg['ESS'])
assert(len(price_buy) == n_grid), 'Grid number does not match with the TOML file.' #check
for
length of price buy and grid
assert(len(P_LD) == n_load), 'Load number does not match with the TOML file.' #check for
length
of active load power and load
assert(len(P_RDG) == n_RDG), 'RDG number does not match with the TOML file.' #check
for
length of active renewable power and RDG
if ESS_recover:
if len(E_org) != 0: #check for initial ess soc level
assert(len(E_org) == len(SOC_init)), 'You required final state SOC, but ESS number
does not match.'
#========================================================= MILP model
building
=================================================================
model = pyo.ConcreteModel() #optimization solver
model.T = pyo.Param(initialize = T) #T= number of columns in price_buy
model.Δt = pyo.Param(initialize = mg["Microgrid"][0]["Δt"] / 60) # minute --> hour
L_PWF = mg["Microgrid"][0]["L_PWF"] #copy data from microgrid dictionary to L_PWF
51# Utility Grid
Pb_max = [x["Pb_max"] for x in mg["Grid"]]
Pb_min = [x["Pb_min"] for x in mg["Grid"]]
Ps_max = [x["Ps_max"] for x in mg["Grid"]]
Ps_min = [x["Ps_min"] for x in mg["Grid"]]
model.n_grid = pyo.RangeSet(1, n_grid) #creating indices ranging from 1 to 'n_grid'
```

```
model.t = pyo.RangeSet(1, T) #creating indices ranging from 1 to 'T' T= number of columns
in
price_buy
# Utility Grid Parameters
model.Pb = pyo.Var(model.n_grid, model.t, initialize = 0)
model.Ps = pyo.Var(model.n_grid, model.t, initialize = 0)
model.P_ex = pyo.Var(model.n_grid, model.t)
model.ζb = pyo.Var(model.n_grid, model.t, domain = pyo.Boolean, initialize = 0)
model.ζs = pyo.Var(model.n_grid, model.t, domain = pyo.Boolean, initialize = 0)
# Utility Grid Contraints
# compute values of P_ex based on Pb (battery power), Ps (Solar power) for each grid 'i' and
time
'j'
def Pex_rule(model, i, j):
return model.P_ex[i,j] == model.Pb[i,j] - model.Ps[i,j] # exchange = buy-sell
model.Pex_constraint = pyo.Constraint(model.n_grid, model.t, rule = Pex_rule)
52def buy_sell_rule(model, i, j):
return model.ζb[i,j] + model.ζs[i,j] <= 1 # to let either buying or selling take place at a time
model.buy_sell_constraint = pyo.Constraint(model.n_grid, model.t, rule = buy_sell_rule)
def buying_lb_rule(model, i, j):
return model.ζb[i,j] * Pb_min[i-1] <= model.Pb[i,j] # to make sure min buy price is less or
equal to buy price
def buying_ub_rule(model, i, j):
return model.ζb[i,j] * Pb_max[i-1] >= model.Pb[i,j] # to make sure max buy price is greater
than euqal to buy price
def selling_lb_rule(model, i, j):
return model.ζs[i,j] * Ps_min[i-1] <= model.Ps[i,j]
def selling_ub_rule(model, i, j):
return model.ζs[i,j] * Ps_max[i-1] >= model.Ps[i,j]
model.buying_lb_constraint = pyo.Constraint(model.n_grid, model.t, rule = buying_lb_rule)
model.buying_ub_constraint = pyo.Constraint(model.n_grid, model.t, rule = buying_ub_rule)
model.selling_lb_constraint = pyo.Constraint(model.n_grid, model.t, rule = selling_lb_rule)
model.selling_ub_constraint = pyo.Constraint(model.n_grid, model.t, rule = selling_ub_rule)
# RDG
```

```
model.n_RDG = pyo.RangeSet(1, n_RDG)
model.P_RDG = pyo.Param(model.n_RDG, model.t, initialize = {(i,j): P_RDG[i-1][j-1] for i
in
model.n_RDG for j in model.t}) # store power data for renewable
53# ESS (Energy Storage System)
γch = [x["γch"] for x in mg["ESS"]] # store charging data
γdis = [x["γdis"] for x in mg["ESS"]] # store dis-charging data
model.n_bat = pyo.RangeSet(1, n_bat) #
Pch_min = [x["Pch_min"] for x in mg["ESS"]]
Pch_max = [x["Pch_max"] for x in mg["ESS"]]
Pdis_min = [x["Pdis_min"] for x in mg["ESS"]]
Pdis_max = [x["Pdis_max"] for x in mg["ESS"]]
model.Pch = pyo.Var(model.n_bat, model.t)
model.Pdis = pyo.Var(model.n_bat, model.t)
model.P_bat = pyo.Var(model.n_bat, model.t)
model.ζch = pyo.Var(model.n_bat, model.t, domain = pyo.Boolean)
model.ζdis = pyo.Var(model.n_bat, model.t, domain = pyo.Boolean)
# ESS Contraints
def Pbat_rule(model, i, j):
return model.P_bat[i,j] == model.Pdis[i,j] - model.Pch[i,j] # battery power = dischaging -
charging
model.Pbat_constraint = pyo.Constraint(model.n_bat, model.t, rule = Pbat_rule)
54def charge_discharge_rule(model, i, j):
return model.ζch[i,j] + model.ζdis[i,j] <= 1 # to ensure either charging or discharging takes
place at a time
model.charge_discharge_constraint = pyo.Constraint(model.n_bat, model.t, rule =
charge_discharge_rule)
def charging_lb_rule(model, i, j):
return model.ζch[i,j] * Pch_min[i-1] <= model.Pch[i,j] #upper bound for charging
def charging_ub_rule(model, i, j):
return model.ζch[i,j] * Pch_max[i-1] >= model.Pch[i,j] #lower boung for charging
def discharging_lb_rule(model, i, j):
return model.ζdis[i,j] * Pdis_min[i-1] <= model.Pdis[i,j]
```

```
def discharging_ub_rule(model, i, j):
return model.ζdis[i,j] * Pdis_max[i-1] >= model.Pdis[i,j]
model.charging_lb_constraint = pyo.Constraint(model.n_bat, model.t, rule =
charging_lb_rule)
model.charging_ub_constraint = pyo.Constraint(model.n_bat, model.t, rule =
charging_ub_rule)
model.discharging_lb_constraint = pyo.Constraint(model.n_bat, model.t, rule =
discharging_lb_rule)
model.discharging_ub_constraint = pyo.Constraint(model.n_bat, model.t, rule =
discharging_ub_rule)
# SOC Constraint (State Of Charge)
55ηch = [x["ηch"] for x in mg["ESS"]]
ηdis = [x["ηdis"] for x in mg["ESS"]]
E_cap = [x["E_cap"] for x in mg["ESS"]]
SOC_min = [x["SOC_min"] for x in mg["ESS"]]
SOC_max = [x["SOC_max"] for x in mg["ESS"]]
E0 = {i: E_cap[i-1] * SOC_init[i-1] for i in model.n_bat} # energy level = energy capacity *
state
of charge
model.E0 = pyo.Param(model.n_bat, initialize = E0) # energy level
model.E = pyo.Var(model.n_bat, model.t) # decision variable
model.SOC = pyo.Var(model.n_bat, model.t) # state of charge
# Regulation of SOC
def SOC_update_rule(model, i, j):
if j == 1:
#return model.E[i, 1] == model.E0[i] + model.Pch[i, 1] * model.Δt * ηch[i-1] -
model.Pdis[i, 1] * model.Δt / ηdis[i-1]
return model.E[i, 1] == 1240
##########################################################
#For the first time step, the energy stored in the ESS is calculated based on the initial energy,
the power charged, and the power discharged during the first time step.
else:
return model.E[i, j] == model.E[i, j-1] + model.Pch[i, j] * model.Δt * ηch[i-1] -
model.Pdis[i, j] * model.Δt / ηdis[i-1]
```

```
56#For subsequent time steps, the energy stored in the ESS is updated based on the energy
stored in the previous time step
model.SOC_update_constraint = pyo.Constraint(model.n_bat, model.t,
rule=SOC_update_rule)
def SOC_lb_rule(model, i, j):
return E_cap[i-1] * SOC_min[i-1] <= model.E[i,j] #prevent SOC from dropping below E,
capacity *min = Energy stored
def SOC_ub_rule(model, i, j):
return E_cap[i-1] * SOC_max[i-1] >= model.E[i,j] #prevent SOC from going above E, cap *
max = Energy stored
model.SOC_lb_constraint = pyo.Constraint(model.n_bat, model.t, rule=SOC_lb_rule)
model.SOC_ub_constraint = pyo.Constraint(model.n_bat, model.t, rule=SOC_ub_rule)
def SOC_calculation_rule(model, i, j):
return model.SOC[i, j] == model.E[i, j] / E_cap[i-1] #SOC = Energy stored/Capacity
model.SOC_calculate = pyo.Constraint(model.n_bat, model.t, expr = SOC_calculation_rule)
def ESS_last_rule(model, i):
if len(E_org) == 0: #if E_org is empty
return model.E[i, T] >= model.E0[i]
#ensure energy stored in ESS at the final step is at least as much as the initial energy sotred
else:
return model.E[i, T] >= E_org[i-1]
57if ESS_recover:
model.ESS_last_constraint = pyo.Constraint(model.n_bat, rule = ESS_last_rule)
# Load
λ_shed = [x["λ"] for x in mg["Load"]]
model.n_load = pyo.RangeSet(1, n_load)
model.P_LD = pyo.Param(model.n_load, model.t, initialize = {(i,j): P_LD[i-1][j-1] for i in
model.n_load for j in model.t}) # power demand
# Load shedding
α_NCL = [x["α_NCL"] for x in mg["Load"]]
model.P_shed = pyo.Var(model.n_load, model.t)
def load_shedding_lb_rule(model, i, j):
return 0 <= model.P_shed[i, j] # ensure load shedding doesn't fall below 0
```

```
def load_shedding_ub_rule(model, i, j):
return α_NCL[i-1] * P_LD[i-1][j-1] >= model.P_shed[i, j] # ensure load shedding is lesser
than some % of the load demand
model.load_shedding_lb_constraint = pyo.Constraint(model.n_load, model.t, rule =
load_shedding_lb_rule)
58model.load_shedding_ub_constraint = pyo.Constraint(model.n_load, model.t, rule =
load_shedding_ub_rule)
# Power balance
def power_balance_rule(model, j):
return sum(model.Pb[i, j] for i in range(1,n_grid+1)) + sum(model.Pdis[i, j] for i in
range(1,n_bat+1)) + sum(model.P_RDG[i, j] for i in range(1,n_RDG+1)) +
sum(model.P_shed[i, j]
for i in range(1,n_load+1)) == sum(model.Ps[i, j] for i in range(1,n_grid+1)) +
sum(model.Pch[i, j] for
i in range(1,n_bat+1)) + sum(model.P_LD[i, j] for i in range(1,n_load+1))
# left side : power injected (power bought + power discharged from ESS + renewable power
generated + shed power)
# ride side : power withdrawn (power sold + power charged for ESS + load power)
model.power_balance_constraint = pyo.Constraint(model.t, rule = power_balance_rule)
ρb = {(i,j): price_buy[i-1][j-1] for i in model.n_grid for j in model.t} #buying power price
ρs = {(i,j): price_sell[i-1][j-1] for i in model.n_grid for j in model.t} #for selling power price
model.ρb = pyo.Param(model.n_grid, model.t, initialize = ρb)
model.ρs = pyo.Param(model.n_grid, model.t, initialize = ρs)
# Objective function setup
def C_ex_objective_expr(model):
return sum(model.Pb[i, j] * model.ρb[i, j] - model.Ps[i, j] * model.ρs[i, j] for i in
range(1,n_grid+1) for j in range(1,T+1))*model.Δt
# calculate the net cost of buying/selling from the grid over the entire time horizon
def C_shed_objective_expr(model):
59return sum(model.P_shed[i, j] * λ_shed[i-1] for i in range(1,n_load+1) for j in
range(1,T+1))*model.Δt
# calculate the net cost of load shedding over the entire time horizon
def C_ESS_objective_expr(model):
```

```
return sum(model.Pch[i, j] * γch[i-1] + model.Pdis[i, j] * γdis[i-1] for i in range(1,n_bat+1)
for j in range(1,T+1))*model.Δt
# calculate the net cost of charing/discharging over the entire time horizon
model.cost = pyo.Var(model.t)
model.C_ex_vector = pyo.Var(model.t)
model.C_ESS_vector = pyo.Var(model.t)
model.C_shed_vector = pyo.Var(model.t)
def C_ex_step_rule(model, j):
model.first_C_ex_expr = sum(model.Pb[i, j] * model.ρb[i, j] - model.Ps[i, j] * model.ρs[i, j]
for i in range(1,n_grid+1))*model.Δt
return model.C_ex_vector[j] == model.first_C_ex_expr
# calculating cash flow (bought price - sold price)
def C_shed_step_rule(model, j):
model.first_C_shed_expr = sum(model.P_shed[i, j] * λ_shed[i-1] for i in
range(1,n_load+1))*model.Δt
return model.C_shed_vector[j] == model.first_C_shed_expr
# calculating cash flow (shed power * shedding cost * time)
def C_ESS_step_rule(model, j):
60model.first_C_ESS_expr = sum(model.Pch[i, j] * γch[i-1] + model.Pdis[i, j] * γdis[i-1] for
i
in range(1,n_bat+1))*model.Δt
return model.C_ESS_vector[j] == model.first_C_ESS_expr
# calculating cost of charging and discharging process
def step_cost_rule(model, j):
return model.cost[j] == model.C_ex_vector[j] + model.C_shed_vector[j] +
model.C_ESS_vector[j]
model.C_ex_step_constraint = pyo.Constraint(model.t, rule = C_ex_step_rule)
model.C_shed_step_constraint = pyo.Constraint(model.t, rule = C_shed_step_rule)
model.C_ESS_step_constraint = pyo.Constraint(model.t, rule = C_ESS_step_rule)
model.step_cost_constraint = pyo.Constraint(model.t, rule = step_cost_rule)
def Objective_rule(model):
model.C_ex_objective_expr = pyo.Expression(expr = C_ex_objective_expr)
model.C_shed_objective_expr = pyo.Expression(expr = C_shed_objective_expr)
model.C_ESS_objective_expr = pyo.Expression(expr = C_ESS_objective_expr)
```

```
model.objective_expr = pyo.Expression(expr = model.C_ex_objective_expr +
model.C_shed_objective_expr + model.C_ESS_objective_expr)
return model.objective_expr
model.obj = pyo.Objective(expr = Objective_rule)
solver = pyo.SolverFactory('glpk')
results = solver.solve(model)
#P_shed_is0 = max(pyo.value(model.P_shed[i,j]) for i in range(1,n_load+1) for j in range(1,
T+1))
# Print the results
print(model.obj())
return model
```