Solving Optimization Problems in the Electricity System –Using PuLP, docplex and Pyomo package

Ruoshui Li[†]

October, 2020

Optimization problems are often involved in electricity system design. In this documentation, we will introduce `PuLP`, `docplex` and `Pyomo` packages in Python to model and solve three types of typical optimization problem in power system studies.

1 Solving UCED model in a 3-bus system using PuLP

PuLP is an LP modeler written in Python, which can solve most of the linear programming problems including integer programming. It works better on small-scale power system problems with relatively lower variables, and can effectively produce solutions and conduct sensitivity analysis. The PuLP package can be installed directly through command line (for PC users) using `pip install pulp`.

Problem description

In a three-bus power electricity system, grid operators need to dispatch available generators from all three nodes in either day-ahead market or real-time market to meet customer electricity load. The dispatch model usually aims at producing sufficient power output with the least possible generation costs. Therefore, in addition to meeting regional electricity load, generators at each bus may also need to transmit electricity to other buses in order to minimize power supply costs of the entire system.

We now look at a specific three-bus power system with 20 fossil fuel power plants. Respective nameplate capacity, fuel cost, heat rate and other parameter data are all available. We also know that a set of transmission lines between three buses remain effective, though with a certain transmission capacity limitation. We now use PuLP to create an optimization model, if successfully solved, we could obtain: 1). production schedule of 20 generators in a specific time period; 2). amount of

[†]Email ruoshui.li@duke.edu. The first two questions raised in this documentation contain complex data input, therefore are not entirely displayed in the model in consideration of algorithm legibility. You are more than welcome to contact me through email if you're interested in implementing the model with complete data.

electricity transmitted on the grid; and 3). power cost of the marginal generator at each bus, i.e. locational marginal price.

Building the model

We first use `import pulp` to set up the package environment。

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pulp import *
```

Next, read generator data set and prepare fundamental system parameters for later model reference. In order to make codes more readable and to improve the algorithm's performance, the entire model is operated in a vector form.

```
raw_data=pd.read_excel('/raw.xlsx', index_col=0)

# System parameters
Gen_list=list(raw_data.index)
Bus_list=['1','2','3']
Line_list=[('1','2'),('2','3'),('1','3')]
fuel_price={'NG':3.06, 'Coal': 2.16, 'Oil': 9.83} # $/mmBtu
bus1_gen=list((raw_data.loc[raw_data['Bus']==1]).index)
bus2_gen=list((raw_data.loc[raw_data['Bus']==2]).index)
bus3_gen=list((raw_data.loc[raw_data['Bus']==3]).index)
bus_dict={'1': bus1_gen, '2': bus2_gen, '3': bus3_gen}

# Transmission line parameters
line_capacity_dict = dict(zip(Line_list, [1100, 2000, 2000]))
line_impedance = dict(zip(Line_list, [0.2,0.2,-0.2]))

# Percentage of total electricity load from each bus
load_dict={'1': 0.15, '2': 0.5, '3': 0.35}
```

Now it's time to build our simplified model:

```
# Create a new minimization LP problem
prob = LpProblem('Constrained UCED', LpMinimize)

# Decision variables
# 1. Power output of each dispatched generator
gens= LpVariable.dicts('generation',Gen_list)
# 2. Amount of electricity flow on the grid
flows= LpVariable.dicts('flow', Line_list)
```

Here, `LpProblem()` is used to create a new LP problem and simultaneously assign a name (e.g. "Constrained UCED"). Specifying `LpMinimize` or `LpMaximize` enables the modeler to indicate the type of optimization problem. We use `LpVariable()` to define decision variables in a problem. The first argument takes the assigned variable name, while the second one is the key value for indexing. Other arguments can follow if necessary, for instance, applying `LpVariable(lowBound, upBond, cat)` limits the lower/up bound and domain of the variable, respectively. When "cat" is omitted, the model will take continuous variable as a default value, otherwise, modelers can set it as "Integer" or "Binary". Now we declare constraints of this LpProblem and define an objective function:

```
# Constraints
# Supply meets demand
prob += Load - lpSum(gens[g] for g in Gen_list) >= 0, 'Supply meet demand'

# Generator power output within maximum/minimum capacity
for g in Gen_list:
    prob += gens[g] <= maxgen_dict[g], 'Max generation for ' + g
    prob += gens[g] >= mingen_dict[g], 'Min generation for ' + g

# Electricity flow not exceed transmission line capacity
for l in Line_list:
    prob += flows[l] <= line_capacity_dict[l], \
        'Max transmission capacity for line ' + l[0] + l[1]
    prob += flows[l] >= -line_capacity_dict[l], \
        'Inverse max transmission capacity for line ' + l[0] + l[1]
```

```
# Bus balance
for b in Bus_list:
    prob += lpSum(gens[g] for g in bus_dict[b]) \
        - lpSum(v for k,v in flows.items() if k[0]==b) \
        + lpSum(v for k,v in flows.items() if k[1]==b) \
        >= Load * load_dict[b] , 'Bus balance for '+ b

# Meet KVL law
prob += lpSum(flows[l]*line_impedance[l] for l in Line_list) == 0, \
        'KVL along the loop'

# Objective function
prob += lpSum(gens[g] * mc_dict[g] for g in Gen_list)
```

Declaring constraints is actually a process to add linear expressions to the LP instance (i.e. `prob`). Each `prob+ =()` places a limit on the minimization problem. By adding strings enclosed in quotation marks at the end of each sentence, a brief explanation can also be provided such that we can locate the expression value quickly when displaying results. Objective function can either be added to the LP problem before/after constraints. In our example, the expression will calculate total cost of dispatched generators, which is the variable we want to minimize. Worth noting, `LpSum()` is a built-in function of PuLP, using it to calculate inner product of sequences can significantly raise the running speed, which does good to computing resources conservation. Finally, we solve the model and display solutions:

```
prob.solve()
print("The dispatch solution of ", Load, "is: ", LpStatus[prob.status])

# Print value of objective function
value(prob.objective)

# Print decision variable values
for v in prob.variables():
    print(v.name, "=", v.varValue)
```

So far, we can obtain the optimal schedule of each generator and electricity flow between each bus. For more practical analysis, shadow price of each constraint, which indicates change in objective function value as one unit of right-hand-side constraint is released or tightened, can be calculated (see below). In our case, the locational marginal price at each bus will be the shadow price of three

"Bus balance" constraints.

```
shp = pd.DataFrame([{'Constraint_name':name, 'shadow_price':c.pi,
                     'slack':c.slack} for name,c in prob.constraints.items()])
print(shp)
```

2 Conduct Capacity Planning in an ISO using docplex

Compared to PuLP, Cplex is a more widely used LP solver, which can be used to solve various large-scale linear programming problems. You may access the Cplex solver in two ways: one is to download and install Cplex studio IDE from IMB's official website and build models using its own syntax (the IDE is equivalent to R studio for R language, and Spider or Pycharm interface for Python); while the other way is to refer to Cplex packages written in other programming languages through an API interface. Here, I choose to use the Python-based docplex package to solve my LP problem. Nevertheless, this method still needs Cplex studio to be installed beforehand. As downloading the studio, installation assistant will forward an option whether installing other API interfaces or not, just click the Python one, and then run `pip install docplex` in the command line to install the `docplex` package. Detailed installation instructions are omitted fom this documentation, you can simply follow the prompts of IBM's download assistant.

Problem Description

In one of the deregulated states in America, a hypothetical ISO (Independent System Operator) is planning to gradually increase the proportion of renewable electricity in local power system to accelerate clean energy transition. In proposing the capacity plan for the next 10 years, three targets are expected to achieve: 1). meet annual average and summer peak load of local residents; 2). reduce carbon emissions from system generators by over 75%; and 3). minimize total cost for constructing new capacities and annual electricity dispatching.

Currently, the ISO is operating 20 fossil fuel power plants (including 17 gas-fired and 3 coal-fired) and 20 utility-scale solar power plants. One additional utility-scale solar plant, residential solar panels, one concentration solar power facility, and one wind farm are under consideration in the expanding plan. Generator parameters of existing power plants and construction cost of new capacity have all been collected from the ISO websites and previous publications.

Building the model

The first step is always importing the package. docplex contains two main modules: docplex.mp (for Mathematical programming) and docplex.cp (for Constraint programming). The former one is more

commonly used to address integer or mixed-integer linear programming problems. Therefore, we call the mp module to build our model.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import docplex.mp
from docplex.mp.model import Model
```

Next, read the raw data files and set up fundamental parameters (e.g. time horizon, fossil fuel prices, and construction cost) for the model. Similarly, in order to increase code legibility and algorithm efficiency, parameters are operated in a matrix form. Build on current business-as-usual electricity data, we assume a rate of 0.9% per year increase in customer load. We also simulated the growth of summer peak load in future 10 years for model robustness. Detailed data manipulation process before modeling is omitted here.

```
# Read raw data
Gen_info=pd.read_excel('/data.xlsx',sheet_name='Geninfo', index_col=0)
Capex=pd.read_excel('/data.xlsx',sheet_name='Capex($kw)', index_col=0)
OMcost=pd.read_excel('/data.xlsx',sheet_name='OMcost($MWh)', index_col=0)
tech_dict=(Gen_info.loc[:,'Fuel']).to_dict()
omcost_dict=OMcost.to_dict() # $/MWh
capex_dict=Capex.to_dict() # $/MWh
# Primary parameters used in the model
times=np.arange(2020,2031) # 2020 to 2030
hours=np.arange(0,24)
gens=list(Gen_info.index)
fuel_price={'NG':3.06, 'COAL': 2.16, 'OIL':9.83,
            'UtiliPV':0,'ResPV':0, 'Wind':0, 'CSP':0} # /mmBtu
```

Now let's start creating our Cplex solver.

```
# Create a new mp model
md1 = Model(name='capacity planning')

# Decision variables
gens_exist=list(Gen_info.loc[Gen_info['already online']==1].index)
gens_new=list(Gen_info.loc[Gen_info['already online']==0].index)

# 1. Power output from each generator in a year
generate = md1.continuous_var_matrix(keys1=times, keys2=gens)

# 2. Newly built capaity (either wind or solar) in one year
buildCapacity = md1.continuous_var_matrix(keys1=times, keys2=gens_new)

# 3. Whether or not one existing generator is dispatched in a year
online = md1.binary_var_matrix(keys1=times, keys2=gens_exist)
```

`Model` component can create a new docplex instance and assign names for it. In this model, we define three groups of decision variables. `continuous_var_matrix` indicates a continuous variable created by a matrix factory. Similar to PuLP, two `keys` argument define the index set for each variable, i.e. time and generation code in this model. Other variable types include "binary", "integer" and "boolean", each has a different scope of possible values. In addition, standard factory also supports creating list (1D) and cube (3D). Notable, for a list variable, index set should be defined with a unique `key=` instead of `keys=`.

```
# Objective function
newCapex=md1.sum(buildCapacity[t, g]*capex_dict[t][tech_dict[g]]*1000\
                 for g in gens_new for t in times)
translinecost=md1.sum(buildCapacity[t, g]*newline_cost*newline_distance[g]\
                 for g in gens if tech_dict[g]=='Wind'\
                 or tech_dict[g]=='CSP' for t in times)
dispatchcost=md1.sum(generate[t,g] * (omcost_dict[t][tech_dict[g]] \
        + fuelcost_dict[g]) for g in gens for t in times)

md1.minimize((newCapex + translinecost + dispatchcost)*(1/(1.089**11)))
```

The objective function, i.e. total cost, contains three parts: construction cost of new capacity, additional transmission line costs of potential wind farm or CSP plant, and annual dispatch costs in 10 years. Now we enter all the constraints.

```
# Supply meet annual average electricity demand
# (also assume that 30% of coal or gas capacity need to
# be reserved for the sake of grid reliability)
for t in times:
    md1.add_constraint(md1.sum(generate[t,g] for g in gens) \
                    >= annual_dict[t][0] * (1+0.108))
    md1.add_constraint(md1.sum(generate[t][g] for g in gens \
                    if tech_dict[g]=='NG' or tech_dict[g]=='COAL' \
                or tech_dict[g]=='OIL') >= (0.3*annual_dict[t][0])*(1+0.108)

# Meet summer peak load (including peak load + 10.8% reserve margin)
for t in times:
    existing_gen=md1.sum(avai_dict[g]*online[t][g] for g in gens_exist)
    new_gen=md1.sum(md1.sum(buildCapacity[T][g] for T in np.arange(2020,t) \
            for g in gens_new)*cf_dict_peak[tech_dict[g]] for g in gens_new)
    md1.add_constraint(existing_gen + new_gen >= load_dict[t][0]*(1+0.108))

# Annual new capacity construction not exceed maximum allowance
md1.add_constraints(md1.sum(buildCapacity[t][g] for t in times) <=\
                     maxgen_dict[g] for g in gens_new)

# Power output from existing generators within maximum/minimum capacity
for t in times:
    md1.add_constraints(generate[t][g] <= maxgen_dict[g]*24*365*online[t][g]\
                    *cf_dict_annual[tech_dict[g]] for g in gens_exist)
    md1.add_constraints(generate[t][g] >= mingen_dict[g]*24*365*online[t][g]\
                    *cf_dict_annual[tech_dict[g]] for g in gens_exist)

# Power output from newly-built capacity also within available range
for t in times[1:]:
    md1.add_constraints(generate[t][g] <= md1.sum(buildCapacity[T][g] \
                    for T in np.arange(2020,t) \
                    for g in gens_new)*24*365*cf_dict_annual[tech_dict[g]])
    md1.add_constraints(generate[t][g] >=0)
for t in times[:1]:
    md1.add_constraints(generate[t][g] ==0)

# Meet carbon reduction target in the last of 10 years(<25% of benchmark)
for t in times[-1:]:
    md1.add_constraint(md1.sum(generate[t][g]*CO2_dict[g] for g in gens)\
                    <= 0.25* 61153730298)
```

As more constraints and variables entered the docplex instance, we can see how vectorized operations outperform. Notice that the process of entering constraints is prone to errors: when using `add_constraint`, if the statement contains multiple components of the same form with varied index, we need to switch to `add_constraints`, otherwise docplex will not recognize this group of constraints. In writing such syntax, the model also automatically prints out each expression under the statement. This benefits the debugging process in that we can manually check validity of each component. However, this meanwhile makes the model tedious. Therefore, we can change `add_constraint` to `add_constraint_` in order to temporarily prevent model printing.
We now solve the docplex model:

```
md1.solve()
print(md1.get_solve_status())
# Display results:
solution = md1.solve()
solution.display()
```

If a feasible solution is successfully found, `get_solve_status()` will display a `JobSolveStatus.OPTIMAL_SOLUTION`, otherwise null value will be returned. We then can do preliminary examinations to see whether there are obvious errors. In our case, the model provides: optimal installation schedule of new wind or solar capacity and recommended retirement year for existing coal or gas-fired power plants (this can be concluded if one plant is never dispatched after a certain year).
Of course, all sorts of problems can arise during the modeling process. We may need to interrupt the model, print the constraints and examine different expressions constantly. When seeing clear violation of constraints or unexpected values, direction of inequality or whether the expression contains an equal relationship also need to be carefully assessed. Here are some sentences comes with docplex package for examining conflicted constraints:

```
from docplex.mp.relaxer import Relaxer
from docplex.mp.conflict_refiner import ConflictRefiner,\
    TConflictConstraint, VarUbConstraintWrapper, VarLbConstraintWrapper
# md1 is the name of model which reports an error
cr=ConflictRefiner()
conflicts=cr.refine_conflict(md1)
for conflict in conflicts:
    ct = conflict.element
    label = conflict.name
    print("Conflict involving constraint: \%s" % label)
    print(" \tfor: \%s" % ct)
```

3 Minimize residential Solar+Storage system cost in Pyomo

Pyomo is another Python-based, open-source optimization modeling language with a diverse set of optimization capabilities including linear, non-linear, and quadratic LP. To install, just enter `pip install pyomo` in the command line. However, different from the first two packages, Pyomo itself does not include any optimization solvers. Therefore, we will need to install third-party solvers to solve optimization problems built with Pyomo. There are several options, e.g. cbc, gurobi, cplex and glpk. Since I already have Cplex executor installed on my PC, we will use Cplex to solve our third example. Other third-party solvers, if called, also need to be installed beforehand.

Problem description

Households now increasingly install rooftop PV systems to power their houses as cost of solar energy substantially decrease. Solar plus Storage is an effective way to further increase utilization rate of renewable energy and draw down household electricity fees. When combined with a storage system, household load will first consume power output from solar panels to the greatest extent, then charge the battery with remaining unused energy. As the sunshine wane, the battery discharges to power electrical appliances using stored clean energy. Only when the battery cannot meet power demand from the household, distribution grids will supplement the rest of needs. Our model, in this case, aims at scheduling time and amount of battery charge/discharge, cutting power need from utilities through distribution lines and therefore minimizing household's electricity bill.

Specifically, when there is excess solar power left even after fully charging the battery, utilities / electricity companies treated them differently. In our model, we assume the excess energy will all be reversely transmitted back to the grid, and the utility will purchase whatever amount from households and pay them in a market retail rate.

Building the model

First import necessary packages.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from pyomo.environ import *
5 from pyomo.core import *
```

Pyomo also has two ways to build a model, unlike docplex's mathematical or constriant programming, Pyomo distinguish between an Abstract model and a Concrete model. The former is defined using only unspecified symbols with a separate data file attached, it thus provides a context for defining

and initialing optimization problems in Pyomo while data be supplied until a solution is wanted. In comparison, Concrete model is directly defined with data values provided at the time of model definition. The following two LP models are examples from Pyomo's official documentation:

Abstract Model:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad \forall i = 1 \ldots m \\ & x_j \geq 0 \qquad\qquad\quad \forall j = 1 \ldots n \end{aligned}$$

Concrete Model:

$$\begin{aligned} \min \quad & 2x_1 + 3x_2 \\ \text{s.t.} \quad & 3x_1 + 4x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Modelers will probably prefer to write different types of optimization model depending on circumstances, the choice is largely a matter of taste. In this case, we choose to create an abstract model so rooftop system and battery parameters can change to reflect their influences on customer's total cost. Here is the model:

```
# Create a new abstract model instance
model=AbstractModel("MinPVBatCost")
```

Next, we use the Pyomo `Param` component to declare technical design parameters, including PV panel capacity, charge/discharge efficiency, and battery status of charge (SOC) in this model. `Param` component can take a variety of arguments, the first sentence has illustrated use of the `within` option to validate a non-negative integer value of h.

```
# Prepare parameters
model.h=Param(within=NonNegativeIntegers)
model.PVcap=Param()
model.Batcap=Param()
model.Chargeeffi=Param()
model.Discheffi=Param()
model.SOC0=Param()
model.SOCmin=Param()
model.SOCmax=Param()
model.Chargemin=Param()
model.Chargemax=Param()
model.Dischmin=Param()
model.Dischmax=Param()
```

Main data input of the model includes: 1). Household electricity load in 24 hours; 2). Solar energy production from rooftop system; and 3). Fixed / floating electricity rate in a day. All the data were obtained from software SAM before building the Pyomo model. Although not required, in an abstract model, it is convenient to define index sets for locating variables / data values. Here, component `RangeSet` is used to declare the sets, which will then be entering `Param` components for indexing. Notable, the range set will be a sequence of integers starting at 1, instead of 0 like normal lists or arrays, and ending at a value specified by parameter `model.h`.

```
# Declare index sets
model.i=RangeSet(model.h)

# Declare indexed parameters
model.PVgen=Param(model.i)
model.houseload=Param(model.i)
model.rate=Param(model.i)
```

Now declaring decision variables.

```
# Decision variables
# Electricity use from grid in each hour
model.Grid=Var(model.i, domain=NonNegativeReals)

# 2. Solar energy used to charge battery
model.Charge=Var(model.i, domain=NonNegativeReals)

# 3. Amount of battery energy release
model.Disch=Var(model.i, domain=NonNegativeReals)

# 4. Electricity flow on grid after battery is fully charged
model.Slack=Var(model.i, domain=NonNegativeReals)

# 5. Batter status indicator
model.SOC=Var(model.i, domain=NonNegativeReals)
```

`Var` component is specifically used to declare decision variables. The first argument is an index set, while the second one implements the requirement that all need to be greater than or equal to zero. As such domain is specified, the information will be passed to the solver when data is provided. In this case, the variables have only one index set (model.i), but multiple sets could be used, e.g. `model.multiindex=Var(model.a, model.b, domain=)`. Variable domain can also

change to RealSet, IntegerSet or Binary. Now it's time to define the objective function.

```
# objective function
def obj_expression(m):
    return sum((m.Grid[i]-m.Slack[i])*m.rate[i] for i in range(1,25))
model.obj=Objective(rule=obj_expression, sense=minimize)
```

In abstract models, Pyomo expressions are usually provided to objective declarations via a user-defined-function with a Python `def` statement. This exhibits huge differences compared to PuLP or docplex. As Pyomo models call a function (i.e. obj_expression in this case) to get objective expressions, it automatically passes the model itself as the first argument. This is why the model name is always the first formal argument when declaring functions. Pyomo then decides the direction of problems based on the value of "sense" argument. When there is no specification, `Objective` component will take minimize as default. Similar to `lpSum`, Pyomo also provides a flexible function to do summation. When two arguments with the same index set are provided, the Pyomo built-in function `summation` can efficiently calculate the inner product. Since this only works when there are no algebraic operations included in the function, we still used the `sum` function from Numpy to formulate the objective function.

Declaration of constraints is a similar process. Several functions are defined to generate constraint and passed to the abstract model via `Constraint` component.

```
# constraints
# Household electricity generation, load, grid demand in a balance
def const1(m,i):
    return m.Grid[i]-m.Charge[i]+m.Disch[i]-m.Slack[i]  \
                    == m.houseload[i] - m.PVgen[i]

# Batteries can only be charged using solar energy
def const2(m,i):
    return m.Charge[i]<=m.PVgen[i]

# Batteries won't discharge more electricity than household need
def const3(m,i):
    return m.Disch[i]<=m.houseload[i]
```

```
# Within the range of battery charge/discharge capacity
def const4(m,i):
    return m.Charge[i]<=m.Chargemax
def const5(m,i):
    return m.Charge[i]>=m.Chargemin
def const6(m,i):
    return m.Disch[i]<=m.Dischmax
def const7(m,i):
    return m.Disch[i]>=m.Dischmin

# Battery not over-charge / over-discharge
def const8(m,i):
    if i==1:
        return m.SOC[i]==m.SOC0 + m.Charge[i]*m.Chargeeffi/m.Batcap  \
                    - 1/(m.Batcap*m.Discheffi)*m.Disch[i]
    else:
        return m.SOC[i]==m.SOC[i-1] + m.Charge[i]*m.Chargeeffi/m.Batcap  \
                    - 1/(m.Batcap*m.Discheffi)*m.Disch[i]
def const9(m,i):
    return m.SOC[i]<=m.SOCmax
def const10(m,i):
    return m.SOC[i]>=m.SOCmin

# Neter constraints to the abstract model
model.Const1 = Constraint(model.i,rule=const1)
model.Const2 = Constraint(model.i,rule=const2)
model.Const3 = Constraint(model.i,rule=const3)
model.Const4 = Constraint(model.i,rule=const4)
model.Const5 = Constraint(model.i,rule=const5)
model.Const6 = Constraint(model.i,rule=const6)
model.Const7 = Constraint(model.i,rule=const7)
model.Const8 = Constraint(model.i,rule=const8)
model.Const9 = Constraint(model.i,rule=const9)
model.Const10 = Constraint(model.i,rule=const10)
```

By far, we've created a Pyomo optimization instance of the `AbstractModel` class, which contains a `model.i` set object for indexing many modeling components. Now, in order to solve the model, data

need to be provided for the values of parameters. For this model, I compiled the data code in AMPL ".dat" format, part of which can be seen below. Since we have an abstract model in hand, parameter values can be changed quickly for sensitivity analysis.

```
param h := 24;
param PVcap:= 7.92;
param Batcap:= 4;
param Chargeeffi:= 0.92;
param Discheffi:=0.92;
param SOC0:=0.2;
param SOCmin:= 0.2;
param SOCmax:= 0.8;
param Chargemin:= 0;
param Chargemax:=3;
param Dischmin:=0;
param Dischmax:= 3;
```

Now let's call the cplex solver to solve the question:

```
instance=model.create_instance("/datafile.dat")
opt=SolverFactory('cplex',executable="/cplex.exe").solve(instance)

# Display results:
opt.write()
```

If a group of optimal solution is found, `opt.write()` will print the objective function value and give "Solver Status" an "ok". We therefore can obtain the ideal schedule for charging/discharging battery and transmit electricity back to grids. When examining validity of the model, we can also print out all decision variables and constraints using `pprint()`, see below.

```
# Print optimization info
instance.pprint()
# Print values of all decision variables
for v in instance.component_objects(Var, active=True):
    print ("Variable",v)
    varobject = getattr(instance, str(v))
    for index in varobject:
        print ("   ",index, varobject[index].value)
```