基于 Python PuLP、docplex 包、及 Pyomo 包实现电力系统最优化问题

李若水[†]

2020 年 4 月

电力系统中常涉及最优化问题（Optimization Problem），在这篇文档中，我们将引入 Python 的 `PuLP`，`docplex`，及 `Pyomo` 函数包来解决三类电力系统中的典型最优化问题。

1 基于 PuLP 的三线系统 UCED 模型

PuLP 是基于 Python 进行优化的函数库，能够解包括整数规划在内的绝大多数线性规划问题。它可以很好的解决电力系统中变量不多，规模偏小的最优化问题。PuLP 函数包可以直接通过命令行 `pip install pulp` 安装。

问题总述

在一个三线系统（3-bus power system）中，电网工作人员需要对三个节点处的所有机组进行调度，以满足整个系统中的电力需求。调度过程通常希望以最小的发电成本达到最高的电力输出。因此，除了满足区域性负载量的需求，各节点处的发电机还可能需要向其它节点输电，以达到整个系统的供电成本最小化。

已知某三线系统中共有 20 座化石燃料电厂，额定容量、燃料成本等参数已知，三个节点之间的输电线长期有效，但存在一定的传输限额。现基于 PuLP 函数包构建最优化模型，以得到 20 座发电机的生产安排，并观察比较各节点处边际发电机的单位发电成本。

建模

首先通过 `import pulp` 引入函数包。

[†]邮箱: ruoshui.li@duke.edu。文档中选取的前两个问题都含有较复杂的参数，为了算法的易读性，并未全部在模型中写出。如您有兴趣了解更多，欢迎邮件与我联系。

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

接下来读入发电机数据并设置一些系统基本参数（仅在此列出一部分）。为使代码更具可读性，性能更加客观，整个模型以向量化操作。

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

# 三线系统参数
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}

# 输电线参数
line_capacity_dict = dict(zip(Line_list, [1100, 2000, 2000]))
line_impedance = dict(zip(Line_list, [0.2,0.2,-0.2]))

# 已知总负载量，通过每个节点百分比得出各点电力需求量
load_dict={'1': 0.15, '2': 0.5, '3': 0.35}
```

现在开始构建简单的模型：

```
# 新建一个最小化问题实例
prob = LpProblem('Constrained UCED', LpMinimize)

# 决策变量 (Decision variables)
# 1. 各机组产量
gens= LpVariable.dicts('generation',Gen_list)
# 2. 各输电线上传输的电量
flows= LpVariable.dicts('flow', Line_list)
```

通过 LpProblem() 可以新建一个 LP 问题实例，同时设置一个问题名（例如'Constrained UCED'）。指定 LpMinimize 或 LpMaximize 可以求目标函数的最小或最大值。LpVariable() 可以用来构造 LP 问题中的变量，第一个参数是变量名，第二个则是索引各变量的唯一关键字。此处还省略了其他一些可定义的参数，如：LpVariable(lowBound, upBond, cat) 可分别定义决策变量的上下阈值及变量类型。cat 省略时默认为连续型变量，可特别指定为 Integer 或 Binary。接下来向 LpProblem 中加入约束变量，并定义目标函数：

```
# 约束变量 (Constraints)
# 电力供给满足系统需求
prob += Load - lpSum(gens[g] for g in Gen_list) >= 0, 'Supply meet demand'

# 机组发电量处在容量范围以内
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

# 节点之间输电量应小于输电线限额
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]

# 每个节点发电量，输出电量，及输入电量达平衡状态
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

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

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

在 LP 问题中加入约束条件即是向构建的 Lp 实例（prob()）中加入线性不等式的过程。每次的

`prob+ =()` 都对最小化问题作出了限制，通过在语句最后加上含引号的内容，可以为不同的约束条件指定名称，以便在展示结果时快速找到不同变量值。`LpSum()` 可以用来计算序列的点积，是 Pulp 中内置的函数，因此运行速度比普通 `sum()` 更快。目标函数往往是最后一个加入 LP 实例中的不等式，在此例中计算了被调度的机组发电成本之和，这也是我们想要最小化的变量。最后对模型进行求解并展示结果：

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

# 输出目标函数值
value(prob.objective)

# 输出各决策变量值
for v in prob.variables():
    print(v.name, "=", v.varValue)
```

由此，若模型给出 optimal 的唯一解，便可得到各机组发电量及个点之间输电量。实际研究中，调度者往往还会关心各节点处的边际发电成本，便可以进一步计算约束条件的阴影价格（Shadow price）。由于边际发电成本反映的是电力负荷每增加一单位多出的发电成本，因此即等于三个”Bus balance”约束变量的阴影价格：

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

2 基于 docplex 进行区域容量规划（Capacity Planning）

相较于 Pulp，Cplex 是一种使用更广泛的模型求解器（Solver），可以用来求解各种大规模的线性规划问题。两种途径可以使用 Cplex 模型：一是从 IBM 的官网上下载安装 Cplex studio IDE 使用（相当于 R 语言的 Rstudio，Python 的 Spyder 或 Pycharm 界面）；二是使用编程语言的 API 接口调用函数包。由于我有较丰富的 Python 编程语言经历，因此选择使用基于 Python 的 docplex 函数包调用 cplex 的求解器来进行模型求解。但此法仍需先安装 Cplex studio。Studio 下载过程中安装助手会提示是否安装其他 API 接口，在此选择 Python 编程语言安装，再通过命令行命令 `pip install docplex` 安装 `docplex` 函数包。具体安装过程在此文档中略去，根据 IBM 官网下载界面的提示进行操作即可。

问题总述

以美国电力系统为背景，为加速能源转型，某 ISO（Independent System Operator）希望逐步增加新能源发电量在系统中的占比。针对未来 10 年当地电力系统的容量规划，目标有三：满足当地居民的年度平均及夏季峰值用电量；达到整个系统减排 75% 的目标；以及最小化新建容量的建设费用与机组调度成本（dispatch cost）之和。

已知区域内现存 20 座化石燃料发电站（其中 17 座燃气，3 座燃煤）和 20 片太阳能光伏电厂。现规划扩展住宅区太阳能容量（Residential solar），新增一座公用规模太阳能发电厂（Utility PV）；聚光太阳能热电厂（Concentrated Solar Power plant）和一座风电厂。现存电厂发电参数及新增容量建设成本已从 ISO 官方网站及报告中收集到。

建模

首先引入函数包。docplex 主要包含两个模块：docplex.mp 和 docplex.cp。前者用于处理常见的整数规划、线性规划、混合整数线性规划问题等。我们引入 mp 模块进行建模。

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

接下来读入数据并设置一些基本参数，如时间，化石燃料价格，成本参数等。同样，为使代码更具可读性，参数仍以矩阵形式体现。根据已有的用电数据，假设每年负载量以 0.9% 的速度增加，我们还模拟了未来 10 年内夏季高峰用电量及年度平均用电量，此处略去。

```
# 读入数据
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
# 基本参数
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
```

下面正式建立 Cplex 求解器。模型中的参数需在代码准备阶段完成设置。

```
# 新建模型
md1 = Model(name='capacity planning')

# 决策变量 (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. 每座电厂在某一年的发电量
generate = md1.continuous_var_matrix(keys1=times, keys2=gens)

# 2. 光、风等规划能源在某一年的新建容量
buildCapacity = md1.continuous_var_matrix(keys1=times, keys2=gens_new)

# 3. 现存电厂在某年是否被调度发电 (哑变量)
online = md1.binary_var_matrix(keys1=times, keys2=gens_exist)
```

`Model` 可以直接新建 docplex 模型并对其赋名。在这个模型中，我们一共设置了三个决策变量。`continuous_var_matrix` 代表这是一个连续型变量，且变量容器是 matrix（矩阵）类。类似于 Pulp 函数包，两个 `keys` 代表矩阵索引的关键字，此处为时间年份和发电站代号。其他变量类型还包括 binary，integer 和 boolean，变量容器还支持 list（一维）和 cube（三维）。注意，变量容器类型为 list 时，设置唯一索引关键词用 `key=` 即可。

```
# 目标函数 (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)))
```

目标函数中将总成本拆解为了三部分：新建风光资源的建造成本；新建风电场或 CSP 涉及的额外输电线成本；及 10 年内电网调度机组花费。接下来向模型中加入约束变量：

```python
# 满足年度平均电力需求（且假设电网中需要留存 30% 煤电、气电保证稳定性）
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)

# 满足夏季峰值（包含 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))

# 新建容量总量不得超过最高允许值，但允许分多次建设
md1.add_constraints(md1.sum(buildCapacity[t][g] for t in times) <=\
                    maxgen_dict[g] for g in gens_new)

# 现存电厂发电量需要在其容量范围之内
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)

# 新建电厂发电量也需要在每年新建容量范围内
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)

# 最后一年满足减排目标 (<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)
```

约束变量的设置需要特别注意,涉及较多变量时便体现出了向量化操作的优势。注意：add_constraint 时，若隐含超过一个不等式，需要改成 add_constraints，不然模型会报错。在这种写法下，模型会自动将每个 constraint 代表的不等式全部输出，这有助于检查公式输入正确与否，利于 debug 过程，但同时也使程序变得冗杂。因此可将 add_constraint 变成 add_constraint_ 暂时阻止程序输出。最后对模型进行求解：

```
md1.solve()
print(md1.get_solve_status())
# 显示结果：
solution = md1.solve()
solution.display()
```

若成功找到唯一的最优解，则 get_solve_status() 会输出 JobSolveStatus.OPTIMAL_SOLUTION，若不成功则可能输出为空。之后便可对模型结果进行初步检查看是否有明显错误。从本问题的模型求解结果我们可以得出：光、风等可再生资源的最优安装年份和容量大小，及现存燃煤、燃气电厂的建议退休年份（自某年之后没有再被调度）。

当然，建模过程中可能会出现各种各样的问题。当模型结果与预期产生差异，可将所有约束变量不等式输出进行检查。不等式的方向，是否包含等于关系也要细致检验。docplex 自带的函数包也可检验产生冲突的约束条件：

```
from docplex.mp.relaxer import Relaxer
from docplex.mp.conflict_refiner import ConflictRefiner,\
    TConflictConstraint, VarUbConstraintWrapper, VarLbConstraintWrapper

# 例如 md1 出现问题时：
cr=ConflictRefiner()
conflicts=cr.refine_conflict(md1)
for conflict in conflicts:
    st = conflict.status
    ct = conflict.element
    label = conflict.name
    label_type = type(conflict.element)

    # 输出相互矛盾的约束变量
    print("Conflict involving constraint: \%s" % label)
    print(" \tfor: \%s" % ct)
```

3 基于 Pyomo 的光伏 + 储能系统成本优化

Pyomo 是另一种在业界较为常用的开源函数包，可以解决线性、非线性、和二次规划等多种问题。与前两个函数包不同的是，Pyomo 是独立于 Solver 之外的建模语言，即用户可使用第三方求解器（如：cbc，gurobi，cplex 和 glpk）对同一模型进行求解，这也使分布式优化成为可能。Pyomo 函数包可以通过命令行 `pip install pyomo` 进行安装，由于我已经在本地安装了 Cplex studio，Pyomo 可以直接调用 cplex 求解器寻找模型最优解。若使用其它自定义求解器，也需提前安装至本地并确定安装路径可被调用。

问题总述

随着光伏发电成本的大幅度降低，更多人使用屋顶光伏系统为家庭供电。光伏与储能的结合则是另一种提高可再生能源利用率，减少家庭用电支出的有效方式。在该模式下，家庭负载会最大程度消耗光伏发电量，并由蓄电池储存剩余电量。当光照变弱时，蓄电池可通过逆变器为用电器提供储存的清洁电力，当电池不能满足负载用电需求时，由电网补充电量供负载使用。如何规划电池的充放电电量和时间以最小化用户需支付的网电费用是此类最优化模型需要解决的问题。

在光伏发电量较高的时段，针对蓄电池充满后剩余发电量的处置在各地有所不同。在本例中，假设剩余的光电将全部反向馈送至电网，且用电公司将以零售电价对用户进行补贴。

建模

首先引入需要的组件。

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pyomo.environ import *
from pyomo.core import *
```

Pyomo 支持两种建模方式：抽象模型（Abstract model）和具体模型（Concrete model）。前者使用变量名代替参数，单独创建数据文件以供索引。因此可用于定义和初始化问题，仅在具体求解时指定数据值。相较之下，具体模型直接定义变量数值并使用在模型中。以下是官方文档中的两类模型示例：

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:

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

取决于优化问题的类型，不同用户可能选择不同的建模方式。在本例中，由于屋顶光伏和蓄电池参数选择较为多样，我选择构建独立于数据以外的抽象模型，以便后期进行敏感性分析，模型创建如下：

```
# 新建模型
model=AbstractModel("MinPVBatCost")
```

接下来使用 Pyomo 的 `Param` 组件声明与屋顶光伏和蓄电池系统设计相关的参数变量。此例中，技术参数包括光伏面板容量、电池充放电效率及充放电状态限值等。在声明时间变量时，`Param` 组件采用了一个的 `within` 参数（argument），以限制其为一个非负的整形变量。所有数据代码将在文档末列出。

```
# 已知的参数变量
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()
```

此例中涉及的数据为：1）用户在 24 小时内的用电负荷；2）屋顶光伏系统的产量曲线；及 3）当地每小时的电价，在构建 Pyomo 模型之前均已通过 SAM 软件模拟获得。在抽象模型中，这些作为初始数据可通过时间参数进行索引（index set）。`RangeSet` 组件可用于定义索引集，并作为参数传至 `Param` 组件，指示其为 Param 参数编制索引。有一点值得注意的是，与普通的 array 或 list 不同，`RangeSet` 定义的索引是一个从 1 而非 0 开始，至指定 model.h 数值的整数序列。

```
# 定义索引集
model.i=RangeSet(model.h)

# 定义含 index 的变量
model.PVgen=Param(model.i)
model.houseload=Param(model.i)
model.rate=Param(model.i)
```

现在正式定义决策变量。

```
# 决策变量 (Decision variables)
# 1. 每小时使用的网电量
model.Grid=Var(model.i, domain=NonNegativeReals)

# 2. 每小时向蓄电池充电量
model.Charge=Var(model.i, domain=NonNegativeReals)

# 3. 蓄电池放电量
model.Disch=Var(model.i, domain=NonNegativeReals)

# 4. 电池充满后户用系统每小时向电网反向馈电量
model.Slack=Var(model.i, domain=NonNegativeReals)

# 5. 指示每小时电池状态的变量
model.SOC=Var(model.i, domain=NonNegativeReals)
```

决策变量的定义使用了 Pyomo 的另一个组件，`Var`。五类决策变量中，第一个参数均为索引集，第二个则指定了变量的可行域 – 非负的实数。此例中，`Var` 只包含了一组索引（model.i），但多组索引也可实现，如：`model.multiindex=Var(model.a, model.b, domain=)`。变量类型也可通过 domain 改变为包含负数的 RealSet, IntegerSet 或 Binary。接下来定义目标函数。

```
# 目标函数 (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)
```

在抽象模型中，Pyomo 通常使用 python 的 `def` 语句自定义目标函数表达式，然后使用 `Objective` 组件传入模型，这与前两类 Pulp 或 docplex 函数包直接加入线性表达式的方式有所不同。当 Pyomo 调用该函数时（此例中即为 obj_expression 函数），抽象函数本身会默认作为第一个参数传入（即 m，或 model），根据 sense 的取值判断为最大化或最小化优化问题。当没有明确定义时，`Objective` 默认执行 minimize 优化。与 `lpSum` 类似，Pyomo 有内置的 `summation` 函数，可以根据索引值更快速、高效地计算两组变量的点积。但由于该函数目前不支持含代数运算表达式的点乘，此例中仍使用了 numpy 的 sum 函数。

与目标函数类似，下一步是声明各类约束条件函数，并通过 `Constraint` 组件传入抽象模型中。

```
# 约束变量（constraints）
# 家庭用电、馈电、取电保持平衡
def const1(m,i):
    return m.Grid[i]-m.Charge[i]+m.Disch[i]-m.Slack[i]  \
                    == m.houseload[i] - m.PVgen[i]

# 蓄电池只能由光伏板进行充电
def const2(m,i):
    return m.Charge[i]<=m.PVgen[i]

# 电池放电量不超过该小时家庭用电器负荷量
def const3(m,i):
    return m.Disch[i]<=m.houseload[i]

# 电池单次充、放电量在允许范围内
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

# 由上一期和本期充、放电量计算每小时电池状态，各小时内电池不可过度充、放电
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
```

```
# 将所有定义约束加入抽象模型中
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)
```

至此，从面向对象的角度出发，我们已经构建了一个 Pyomo 抽象模型实例，其内包含多个组件，并可使用 `RangeSet` 声明的关键字集合进行索引。为使用该模型进行求解，现需定义含参数值的数据文件。此例中，我使用了 AMPL 格式编写.dat 文件，部分数据如下。由于我使用的是抽象模型，可变动参数值进行敏感性分析。

```
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;
```

现在调用 cplex 求解器进行模型求解：

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

# 显示结果：
opt.write()
```

若成功发现最优解，`opt.write()` 会输出最优化问题的目标函数值，并显示 Solver Status 为“ok”。由此可以得到电池充、放电，及分布式能源向电网反向馈电的时间规划，在智能家居的帮助下最大化户用光伏 + 储能系统利用率，减少家庭需向电网支付的费用。检验模型可行性时，也可以通过 `pprint` 输出所有决策变量及约束变量的值，如下。

```
# 输出最优化问题信息
instance.pprint()

# 输出所有决策变量值
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)
```