Documentation for the Peak Demand Management Solution

Shan Dora He
dora.he3@monash.edu

Prof. Mark Wallace,
Dr. Frits de Nijs

August 23, 2021

List of Acronyms

LP linear programming.

PDMP peak demand management problem.

SOC state-of-charge.

List of Symbols

$p^{+}_{b,n}$ the amount of power charged to the battery at time step n.

$p^{-}_{b,n}$ the amount of power discharged from the battery at time step n.

e^{init}_{b} the initial battery energy level.

e^{max}_{b} the maximum battery capacity.

$\bar{p}_b$ the maximum battery power rate.

e^{min}_{b} the minimum battery capacity.

η_b the battery round-trip efficiency.

$soc_{b,n}$ the state of charge at time step n.

c^{anual} the annual peak demand charge.

c^{summer} the summer peak demand charge.

l_n the forecast load at time step n.

f^{cost} the summer peak demand charge.

g^{health} the summer peak demand charge.

N the total number of time steps.

n the index of a time step.

Summary

Two batteries are available for minimising the annual peak demand and the summer peak demand and therefore the peak demand charges of the Monash Clayton campus. Each battery is modelled by an initial energy level at the beginning of the scheduling horizon, the minimum and maximum allowed energy capacities, the maximum power rate, the amount of power charged to or discharged from the battery per time step, the efficiency and the amount of energy remaining in the battery per time step. A scheduling horizon can be a day or shorter. Each battery can charge or discharge at each time below the maximum power rate, and store energy below the maximum capacity and above the minimum capacity. The energy remaining in the battery at each time step depends on the energy left at the previous time step as well as the charge and discharge. The battery must have energy left at a minimum level at the beginning of the scheduling horizon and recharge back up to that minimum level before the end of the horizon. The objective is to minimise the peak demand charges and the battery health cost (which is designed to avoid frequent charging and discharging). When the load forecast is given, a linear programming (LP) model can be used for solving the peak demand management problem (PDMP) and finding the best time to charge and discharge the battery during the scheduling horizon. A rolling horizon control can be also used to repeatedly solve the PDMP during the day whenever the load forecast is updated, in order to incorporate any changes in real time.

1 Introduction

This document presents the solution for the peak demand management for the Net Zero project at Monash Clayton campus, including the models of the optimisation problem for peak demand management, the solving method and the detailed implementation in Python.

The scope of this work is limited to scheduling batteries given load forecasts and rates for peak demand to minimising the peak demand costs. We assume that the load forecasts are given and updated during the day. In order to incorporate changes in load forecasts in real time, this algorithm needs to be rerun whenever the load forecast is updated in real time. Moreover, it needs to be rerun every day to update the minimal peak demand and the relevant cost. The detailed implementation of this work is available on BitBucket `https://bitbucket.org/dorahee2/battery-scheduling/src/master/`. Please email `dora.he3@monash.edu` for access.

2 Problem Model

The PDMP is concerned with scheduling batteries to minimise the peak demand charges for the yearly maximum demand and the summer peak demand. This section presents the problem model including the parameters, variables, constraints and objective functions.

2.1 Scheduling Horizon

A scheduling horizon (or a day) is divided into multiple time steps. Each time step has the same length (15 minutes in this work):

- N: the total number of time steps
- n: the index of a time step

2.2 Input Data

The input data for this work is the load forecast l_n for each time step n (in kWh).

2.3 Battery Model

Each battery b is represented by:

- e_b^{init}: the initial energy level at the beginning of the day (in kWh)
- e_b^{min}: the minimum allowed energy capacity (in kWh)
- e_b^{max}: the maximum allowed energy capacity (in kWh)
- $\bar{p}_b$: the maximum power rate (in kW)
- $p_{b,n}^{+}$: the amount of power charged to the battery per time step (in kW):
- $p_{b,n}^{-}$: the amount of power discharged from the battery per time step (in kW)
- η_b: the efficiency (between 0 and 1)
- $soc_{b,n}$: a state-of-charge (SOC) profile — the amount of energy remaining in the battery per time step (in kWh)

The $p_{b,n}^{+}$ and $p_{b,n}^{-}$ are the solutions we seek for the battery scheduling problem.

2.4 Battery Constraint

Each battery b is constrained by the followings:

- at each time step n, a battery can either charge or discharge:

$$\forall n \in [1, N],\ p_{b,n}^{+} \times p_{b,n}^{-} = 0 \tag{1}$$

- at each time step n, a battery cannot charge or discharge at a rate higher than the maximum power rate:

$$\forall n \in [1, N],\ 0 \leq p_{b,n}^{+} \leq \bar{p}_b \tag{2}$$

$$\forall n \in [1, N],\ 0 \leq -p_{b,n}^{-} \leq \bar{p}_b \tag{3}$$

- at each time step n, a battery cannot have more (or less) than the maximum (or the minimum) allowed energy:

$$\forall n \in [1, N],\ e_b^{min} \leq soc_{b,n} \leq e_b^{max} \tag{4}$$

- at the first time step of the scheduling horizon, the battery must have satisfy an initial energy level:

$$soc_{b,1} = e_b^{init} \tag{5}$$

- we **assume** that the battery needs to be charged back up to the initial energy level by the end of the scheduling horizon:

$$soc_{b,N} = e_b^{init} \tag{6}$$

- at each time step n, the SOC depends on the SOC, charge and discharge at time step $n-1$:

$$\forall n \in [2, N],\ (soc_{b,n} - soc_{b,n-1}) \times (60/15) = p^+_{b,n-1} + p^-_{b,n-1} \tag{7}$$

2.5 Peak Demand Charge

Two peak demand charges are considered in this work:

- c^{anual}: the annual peak demand charge for all months in a year
- c^{summer}: the summer peak demand charge for December, January and February each year.

2.6 Objective Function

The objective is to minimise the total peak demand charges each (financial year) while keeping the battery operates in an healthy manner. Two objectives are considered:

- peak demand cost:

$$l'_n = l_n \times (60\ 15) + \sum_b (p^+_{b,n}\ \eta_b + p^-_{b,n} \times \eta_b) \tag{8}$$

$$f^{cost} = max([l'_n\ |n \in [1, N]]) \times (c^{anual} + \alpha \times c^{summer}) \tag{9}$$

$$\alpha = \begin{cases} 1, & \text{if the current month is Dec/Jan/Feb} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

- battery health cost:

$$g^{health} = \sum_b \sum_{n=1}^{N} p^+_{b,n}/\bar{p}_b \tag{11}$$

Note that we have added the battery health cost to avoid charging and discharging the battery too frequently.

2.7 Formal Problem Formulation

This problem seeks the best values for the charge/discharge per time step: $p^+_{b,n}$ and $p^-_{b,n}$ that solves the following problem:

$$\begin{aligned} &\textsf{minimise} && f^{cost} + g^{health} \\ &\textsf{subject to} && (1),\ (2),\ (3),\ (4),\ (5),\ (6),\ (7) \end{aligned}$$

3 Method

The main solution for solving the peak demand management problem (PDMP) include a linear programming (LP) model and the rolling horizon control. The LP model schedules the battery to solve the Problem 2.7 when the load forecast is received, and the rolling horizon control repeats the scheduling problem whenever the load forecast is updated. The main steps are described as follows:

1. set the current demand threshold to be zero,
2. at each time step, read the load forecast,
3. check if the forecast maximum demand will exceed the current demand threshold, if yes:
   (a) run the LP model to schedule the battery in ways that minimise the peak demand costs,
   (b) update the current demand threshold to the new optimised maximum demand
4. repeat Step 2 and Step 3 when new forecast is available for at the next time step.

4 Exeperiments

We have tested the model with dataset named “Corrected_MondoData(released 2021-06-10).csv” provided by the Net Zero team. Specifically, we have used the metered data of “V4_WH+”. We have learnt from experiments that the highest peak demand cost occurs during summers and our model is able to reduce both the annual peak demand and the summer peak demand to the same level for each year. We have illustrated the results from 2019 to 2020 at the file called “results.html”.

5 Detailed Implementation

This section presents the detailed implementation of the whole algorithm. I have chosen to implement the algorithm in Python, the LP in Python for MiniZinc, and solve the LP model using a software called Gurobi. **Note that** the choice of languages and solver can be changed according to needs as well as the input and output formats in the actual integration.

The code is available on BitBucket `https://bitbucket.org/dorahee2/battery-scheduling/src/master/` (email `dora.he3@monash.edu` for access).

5.1 Parameters

These are the parameters used in the program:

```
b_name = "battery_name"
b_min_capacities = "min_energy_capacities"
b_max_capacities = "max_energy_capacities"
b_max_powers = "max_powers"
b_efficiencies = "efficiencies"
b_init_energy_levels = "init_energy_levels"
b_eod_energy_level = "end_of_day_energy_levels"
b_charges = "battery_charges"
b_discharges = "battery_discharges"
b_soc = "battery_socs"
b_modified_demand = "modified_demand"
b_modified_max_demand = "modified_max_demand"
b_num_batteries = "num_batteries"

d_net_demand = "existing_demands"
```

```
d_datetime = "timestamp"
d_demand = "demand"

r_annual_max = "annual_max_charge"
r_summer_max = "summer_max_charge"
r_peak_demand_charge = "peak_demand_charge"
r_months = "months"
r_demand_threshold = "demand_threshold"
r_charge_name = "charge_name"
r_cycle_start_month = "begin_cycle_month"

status_updated = "demand_threshold_updated"
status_unchanged = "demand_threshold_unchanged"
```

5.2 Battery Class

This class is responsible for capturing the specifications (initial energy levels, minimum and maximum capacities, maximum power rates and efficiencies) of batteries.

```
class Battery:

def __init__(self):
    self.specs = dict()
    self.num_batteries = 0
    self.specs_fields = [b_name, b_init_energy_levels,
        b_max_powers, b_min_capacities, b_max_capacities,
        b_efficiencies]
    for key in self.specs_fields:
        self.specs[key] = []

# add new battery specifications
def add_battery(self, initial_capacity, min_capacity,
    max_capacity, power, efficiency, name=""):
    self.specs[b_name].append(name)
    self.specs[b_init_energy_levels].append(initial_capacity
        )
    self.specs[b_min_capacities].append(min_capacity)
    self.specs[b_max_capacities].append(max_capacity)
    self.specs[b_max_powers].append(power)
```

```
    self.specs[b_efficiencies].append(efficiency)
    self.num_batteries = len(self.specs)

# update the battery initial energy levels for the next scheduling horizon after the battery has been scheduled
def update_init_energy_levels(self, results):
    self.specs[b_init_energy_levels]
    = results[b_eod_energy_level]
```

5.3 Load Class

This class is responsible for reading the load forecast.

```
import pandas as pd
from scripts.param import *


class LoadsForecast:

def __init__(self):
    self.num_intervals_day = 0
    self.minutes_interval = 0
    self.num_intervals_hour = 0
    self.forecast_loads = []
    self.forecast_demands = []
    self.forecast_datetime_range = []

def add_loads_forecast(self, forecast_df, frequency):
    column_loads = forecast_df.columns[3]
    column_datetime = forecast_df.columns[0]
    self.forecast_loads = list(forecast_df[column_loads])
    self.forecast_datetime_range = forecast_df[column_datetime]
    self.minutes_interval = int(frequency)
    self.num_intervals_day = int(1440 / self.minutes_interval)
    self.num_intervals_hour = int(60 / self.minutes_interval)
    self.forecast_demands = [l * self.num_intervals_hour for l in self.forecast_loads]
```

```
    if len(self.forecast_datetime_range) != self.num_intervals_day:
        print(self.forecast_datetime_range.iloc[0], "loads forecast has missing data.")
        print("--------------------")
        return False
    else:
        return True
```

5.4 PeakDemandCharge Class

This class is responsible for capturing the peak demand charges, the current demand threshold for each charge and resetting the demand threshold for each year.

```
class PeakDemandCharge:

def __init__(self):
    self.num_charges = 0
    self.demand_charges = dict()
    self.demand_charge_fields = [r_charge_name, r_peak_demand_charge, r_months, r_cycle_start_month, r_demand_threshold]
    for key in self.demand_charge_fields:
        self.demand_charges[key] = []

def set_demand_charge_fields(self, fields):
    self.demand_charge_fields = fields

# add peak demand charges
def add_charge(self, name, rate, months, cycle_start_month):
    self.demand_charges[r_charge_name].append(name)
    self.demand_charges[r_peak_demand_charge].append(rate)
    self.demand_charges[r_cycle_start_month].append(cycle_start_month)
    self.demand_charges[r_months].append(months)
    self.demand_charges[r_demand_threshold].append(0)
    self.num_charges += 1
```

```python
# check if the demand threshold for each peak demand charge needs to be reset to zero
def check_if_new_cycle_begins(self, current_time_step, next_time_step):

    next_month = next_time_step.month
    current_month = current_time_step.month

    if next_month != current_month:
        for i in range(self.num_charges):
            if ((next_month == self.demand_charges[r_cycle_start_month][i] or next_month not in self.demand_charges[r_months][i]) and self.demand_charges[r_demand_threshold][i] > 0):
                self.demand_charges[r_demand_threshold][i] = 0
```

5.5 Scheduler Class

This class is responsible for scheduling the battery when the peak demand management is needed.

```python
from minizinc import *
from scripts.param import *
import numpy as np


class BatteryScheduler:

def __init__(self):
    self.results = dict()
    self.status = ""

# schedule the battery to manage the peak demand
def peak_demand_management(self, loads, batteries, peak_demand_charges, current_month, solver):

# read the relevant demand charges and thresholds for the current month
    relevant_thresholds = []
```

```
    max_demand_charge = 0
    for charge, months, threshold in zip(peak_demand_charges.demand_charges[r_peak_demand_charge],
        peak_demand_charges.demand_charges[r_months],
        peak_demand_charges.demand_charges[r_demand_threshold]):
        if current_month in months:
            max_demand_charge += charge
            relevant_thresholds.append(threshold)
            min_relevant_demand_threshold = min(relevant_thresholds)

# check if the peak demand management event needs to be triggered
    scheduling_horizon_max_demand = max(loads.forecast_demands)
    if scheduling_horizon_max_demand > min_relevant_demand_threshold:

        results = self.__trigger_peak_demand_management(num_intervals_day=loads.num_intervals_day,
num_intervals_hour=loads.num_intervals_hour,
    current_demand_threshold=min_relevant_demand_threshold,
    solver=solver, batteries=batteries.specs,
    max_demand_charge=max_demand_charge, demands=loads.forecast_demands)
        self.status = status_updated
    else:
        results = self.__do_nothing(num_intervals_day=loads.num_intervals_day, current_demand_threshold=min_relevant_demand_threshold, batteries=batteries.specs, demands=loads.forecast_demands)
        self.status = status_unchanged

        results[d_datetime] = loads.forecast_datetime_range
        self.results = results

# do nothing is the peak demand management is not triggered. This function is optional as it is designed more for visualing the results.
```

```
def __do_nothing(self, num_intervals_day,
    current_demand_threshold, demands, batteries):

    results2 = dict()
    battery_socs = [[e] * num_intervals_day for e in
        batteries[b_init_energy_levels]]
    no_battery_activities = [[0 for i in range(
        num_intervals_day)] for _ in range(len(battery_socs))
        ]
    results2[b_charges] = no_battery_activities
    results2[b_discharges] = no_battery_activities
    results2[b_soc] = battery_socs
    results2[b_eod_energy_level] = [soc[-1] for soc in
        battery_socs]
    results2[b_modified_demand] = demands
    results2[b_modified_max_demand] =
        current_demand_threshold
    results2[d_net_demand] = demands

    return results2

# run the linear programming model if peak demand management
     is needed
def __trigger_peak_demand_management(self, num_intervals_day
    , num_intervals_hour, current_demand_threshold, demands,
    solver, batteries, max_demand_charge):
    model = Model()
    model.add_string(
"""
% time
int: num_intervals;
int: num_intervals_hour;
set of int: INTERVALS = 1..num_intervals;

% batteries
int: num_batteries;
set of int: BATTERIES = 1..num_batteries;

array[BATTERIES] of float: init_energy_levels;  % in kwh
array[BATTERIES] of float: min_energy_capacities;  % in kwh
```

```
array[BATTERIES] of float: max_energy_capacities;  % in kwh
array[BATTERIES] of float: max_powers;
float: power_limit = max(max_powers);
array[BATTERIES] of float: efficiencies;

% demands
float: current_demand_threshold;
array[INTERVALS] of float: demand_forecast;
float: demand_limit;

% peak demand charges
float: max_demand_charge;

% decision variables
var 0..demand_limit: daily_max_demand;
array[BATTERIES, INTERVALS] of var 0..power_limit: charges;
array[BATTERIES, INTERVALS] of var -power_limit..0: discharges;
array[BATTERIES, INTERVALS] of var float: soc;
% array[INTERVALS] of var 0..demand_limit: aggregate_battery_profile =
% array1d([sum(b in BATTERIES)(charges[b, i] + discharges[b, i])
% | i in INTERVALS]);
array[INTERVALS] of var 0..demand_limit: modified_demand =
array1d([demand_forecast[i] +
sum(b in BATTERIES)(charges[b, i]/efficiencies[b] + discharges[b, i] * efficiencies[b])
| i in INTERVALS]);

% objective
var float: obj = (daily_max_demand) * max_demand_charge
+ sum(b in BATTERIES, i in INTERVALS)(charges[b, i]) / power_limit;

% either charge or discharge constraint
constraint forall(b in BATTERIES, i in INTERVALS) (charges[b, i] * discharges[b, i] = 0);

% charge constraints
```

```
constraint forall(b in BATTERIES, i in INTERVALS)
(discharges[b, i] <= 0.0);
constraint forall(b in BATTERIES, i in INTERVALS)
(discharges[b, i] >= -max_powers[b]);

% discharge constraints
constraint forall(b in BATTERIES, i in INTERVALS)
(charges[b, i] <= max_powers[b]);
constraint forall(b in BATTERIES, i in INTERVALS)
(charges[b, i] >= 0.0);

% soc constraints
constraint forall(b in BATTERIES, i in INTERVALS)
(soc[b, i] <= max_energy_capacities[b]);

constraint forall(b in BATTERIES, i in INTERVALS)
(soc[b, i] >= min_energy_capacities[b]);

% initial soc
constraint forall(b in BATTERIES)
(soc[b, 1] = init_energy_levels[b]);

% final soc
constraint forall(b in BATTERIES)
(soc[b, num_intervals] = init_energy_levels[b]);

% soc dynamics
constraint forall(b in BATTERIES, i in 2..num_intervals)
(soc[b, i] * num_intervals_hour - soc[b, i - 1] *
    num_intervals_hour =
charges[b, i - 1] + discharges[b, i - 1]);

% max demand
constraint forall(i in INTERVALS)
(daily_max_demand >= modified_demand[i]);
constraint daily_max_demand >= current_demand_threshold;

% solve
solve minimize obj;
"""
```

```python
)
    mip_solver = Solver.lookup(solver)
    ins = Instance(mip_solver, model)

    # time parameters
    ins["num_intervals"] = int(num_intervals_day)
    ins["num_intervals_hour"] = int(num_intervals_hour)

# battery parameters
    num_batteries = len(batteries[b_min_capacities])
    ins["num_batteries"] = num_batteries
    ins["init_energy_levels"] = batteries[
        b_init_energy_levels]
    ins["min_energy_capacities"] = batteries[
        b_min_capacities]
    ins["max_energy_capacities"] = batteries[
        b_max_capacities]
    ins["max_powers"] = batteries[b_max_powers]
    efficiencies = batteries[b_efficiencies]
    ins["efficiencies"] = efficiencies
    ins["demand_forecast"] = demands
    ins["current_demand_threshold"] =
        current_demand_threshold
    ins["demand_limit"] = max(demands) * 999

# peak charges
    ins["max_demand_charge"] = max_demand_charge

    try:
        results = ins.solve()
    except:
        print("error")

    socs = np.array(results.solution.soc).round(2)
    charges = np.array(results.solution.charges).round(2)
    discharges = np.array(results.solution.discharges).round
        (2)
    max_demand_threshold = np.round(results.solution.
        daily_max_demand, 2)
```

```
# actual demand from charging
    actual_demands_from_charging \
        = [np.array(x) / eff for x, eff in zip(charges,
            efficiencies)]
    actual_demand_from_discharging \
        = [np.array(x) * eff for x, eff in zip(discharges,
            efficiencies)]

    total_actual_demand_from_charging = np.array(
        actual_demands_from_charging).sum(axis=0)
    total_actual_demand_from_discharging = np.array(
        actual_demand_from_discharging).sum(axis=0)

    modified_demand = np.array([d + ch + dis for d, ch, dis
        in zip(demands, total_actual_demand_from_charging,
        total_actual_demand_from_discharging)]).round(2)
    if not max_demand_threshold == max(modified_demand):
        print("Modified demand threshold", max(
            modified_demand))

    results2 = dict()
    results2[b_charges] = charges
    results2[b_discharges] = discharges
    results2[b_soc] = socs
    results2[b_eod_energy_level] = [soc[-1] for soc in socs]
    results2[b_modified_demand] = modified_demand
    results2[b_modified_max_demand] = round(
        max_demand_threshold, 2)
    results2[d_net_demand] = demands
    return results2
```

5.6 Rolling Horizon Control

This script runs the battery scheduler every day. The scheduling frequency can be changed, e.g. to every 15 minutes, according to needs.

```

from scripts import load, scheduler, rate, output, asset
from scripts.param import *
import pandas as pd
```

```
def main(solver):
# step 1: add batteries
    batteries = asset.Battery()
    batteries.add_battery(name="Li-on", initial_capacity=134 * 1000, min_capacity=0, max_capacity=134 * 1000, power=120 * 1000, efficiency=0.88)
    batteries.add_battery(name="VFB", initial_capacity=900 * 1000, min_capacity=0, max_capacity=900 * 1000, power=180 * 1000, efficiency=0.65)
    print("Battery specifications are added. ")

# step 2: add peak demand charges
    peak_demand_charges = rate.PeakDemandCharge()
    peak_demand_charges.add_charge(name="annual_charge", rate=131.7 * 1000, cycle_start_month=1, months=[i for i in range(1, 13)])
    peak_demand_charges.add_charge(name="summer_charge", rate=162.5 * 1000, cycle_start_month=12, months=[12, 1, 2])
    print("Peak demand charges are added. ")

# step 3: use historic loads as forecasts
    file = "data/historic_loads.csv"
    historic_loads_df = pd.read_csv(f"{file}")
    historic_loads_df[historic_loads_df.columns[0]] = pd.to_datetime(historic_loads_df[historic_loads_df.columns[0]])
    column_datetime = historic_loads_df.columns[0]
    freq = historic_loads_df[column_datetime][1].minute - historic_loads_df[column_datetime][0].minute
    print("Historic loads are read. ")
    print("--------------------")

# step 4: rolling horizon control -- reschedule on a daily basis
    next_time_step = pd.to_datetime("2016-1-1 00:00")
    reschedule_horizon = pd.Timedelta(days=1)
    reschedule_frequency = pd.Timedelta(days=1)
```

```
optimiser = scheduler.BatteryScheduler()
out = output.Output()
while next_time_step in historic_loads_df[
    column_datetime].values and next_time_step.year <
    2021:

# step 4.1: read the load forecast
current_time_step = next_time_step
scheduling_horizon_end = current_time_step +
    reschedule_horizon
mask = (historic_loads_df[column_datetime] >=
    current_time_step) &  (historic_loads_df[
    column_datetime] < scheduling_horizon_end)
scheduling_horizon_loads = historic_loads_df.loc[mask]
forecast = load.LoadsForecast()
if forecast.add_loads_forecast(forecast_df=
    scheduling_horizon_loads, frequency=freq):

# step 4.2: call the optimiser
current_month = current_time_step.month
optimiser.peak_demand_management(loads=forecast,
    batteries=batteries, peak_demand_charges=
    peak_demand_charges, current_month=current_month,
    solver=solver)
batteries.update_init_energy_levels(results=optimiser.
    results)

# step 4.3: update the demand threshold
if optimiser.status is status_updated:
    updated_demand_threshold = optimiser.results[
        b_modified_max_demand]
    peak_demand_charges.demand_charges[
        r_demand_threshold] = [max(d,
        updated_demand_threshold) if current_month in m
        else d for m, d in zip(peak_demand_charges.
        demand_charges[r_months], peak_demand_charges.
        demand_charges[r_demand_threshold])]
    print(current_time_step, "demand␣thresholds␣are␣
        updated",
```

```
        peak_demand_charges.demand_charges[
            r_demand_threshold])
        print("--------------------")

    # step 4.4: record the daily results
    out.save_results(loads=forecast, optimiser=optimiser,
        peak_demand_charges=peak_demand_charges)

    # step 4.5: move to the next scheduling horizon and
        reset the demand thresholds for every new year
    next_time_step = current_time_step +
        reschedule_frequency
    peak_demand_charges.check_if_new_cycle_begins(
        current_time_step=current_time_step, next_time_step=
        next_time_step)

    out.make_graphs()


main(solver="gurobi")
```