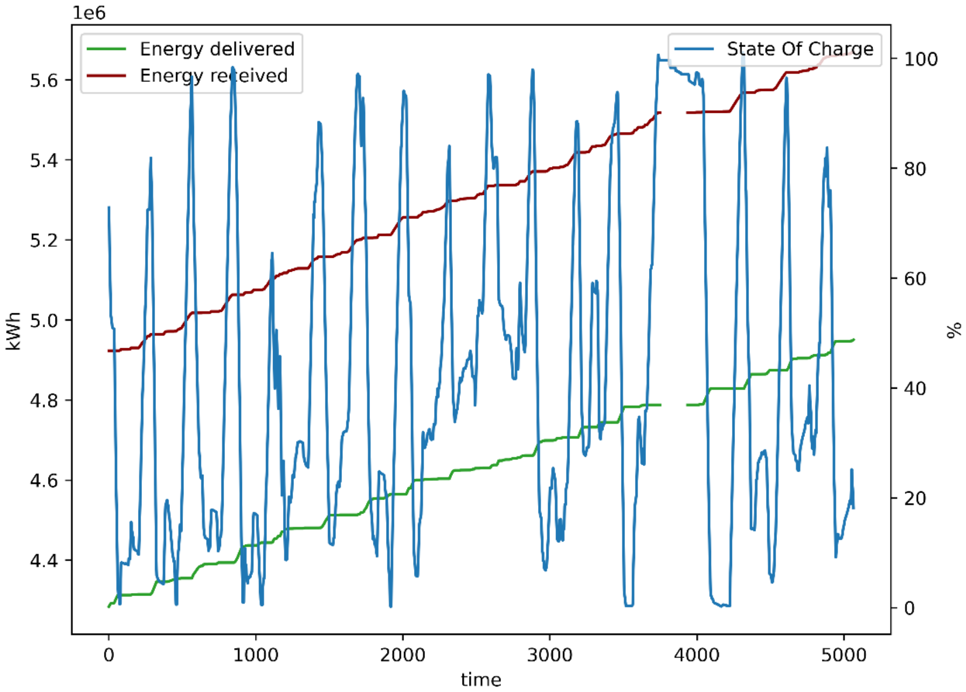
Round Trip Efficiency Estimation for Battery Energy Storage Systems

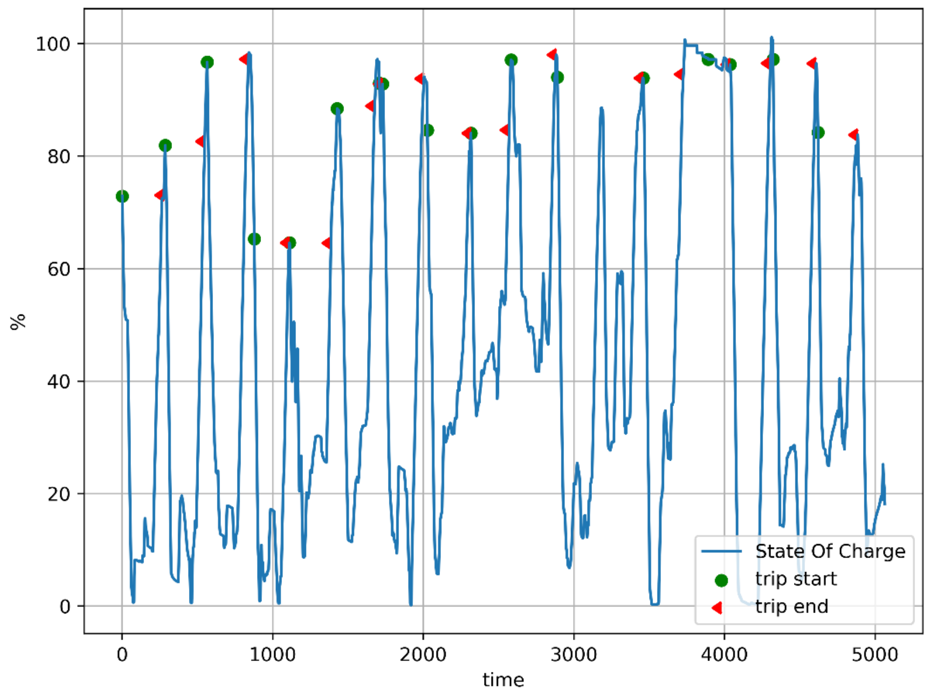
We start by examining the dataset:



Observations:

- There are no round-trips, which require the SOC going from 100% to 0% and then back to 100%.
- The energy columns have missing values

Consequently, we decide to estimate the RTE based on segments of the SOC curve, that we call **trips**.



Trips are a way to replace round-trips.

A trip starts at a local maximum of the SOC curve, and ends when we reach the same State-Of-Charge again

This ignores the oscillations found at low levels of charge

In some cases, to be able to end the trip at the same charge level, we may have to move the starting point from the local maximum.

We deal with missing values in two ways:

- the energy values are taken at the start and at the end of a trip, thus any missing values in between are irrelevant.
- The contribution of a trip to the average RTE is weighted by the Δ-SOC, the difference between the maximum and minimum SOC during the trip.
  If the difference in charge is very low, for instance because the missing values were due to the battery being disconnected, that segment will be less important

The resulting RTE estimation is:

Average RTE over 16 trips = 86.33%

Average RTE over 16 trips (weighted by each trip's Δ SOC) = **86.27%**

note: If our dataset covered a longer time period, with more trips (e.g. *n*>50), we would be able to obtain a confidence interval for the round-trip efficiency.

If *n* is significant, by the Central Limit Theorem the sample mean of the RTE of a trip follows a normal distribution, and we can define a confidence interval with

$$CI = \bar{x} \pm z * \frac{s}{\sqrt{n}}$$

(with $\bar{x}$=sample mean, z=necessary value for a confidence level α, s=sample standard deviation, n=number of samples)

Code

File: rte.py

```
import logging
import sys
from math import isnan
import pandas as pd

# We do not have roundtrips that go 100-to-0-to-100, so the estimation of
the RTE is based on trips:
# A trip is a segment of the State-Of-Charge curve, starting at a local
maximum and ending at the same SOC level
class Trip:
    def __init__(self, start_idx):
        self.start_idx = start_idx
        self.end_idx = 0
        self.energy_delivered = 0
        self.energy_received = 0
        self.min_soc = 0
        self.max_soc = 0

    # the minimum and maximum SOC values during the trip are used to weigh
its contribution to the average RTE
    def set_max_soc(self, y3_soc):
        self.max_soc = y3_soc[self.start_idx]

    def set_min_soc(self, y3_soc):
        trip_soc_values = [y3_soc[i] for i in range(self.start_idx,
self.end_idx)]
        self.min_soc = min(trip_soc_values)
```

```
# The local maxima in the SOC curve constitute the starting point of trips
# We exclude small oscillations at a low level of charge using the
min_value parameter
def get_local_max(soc_ls, k=20, min_value=60):
    # Parameters: k = time slots of monotonic increase of the SOC, followed
by monotonic decrease
    #             min_value = the minimum SOC value for a point to be
recognized as a local maximum

    local_maxima_indices = [0]  # the start is included

    for i in range(k,len(soc_ls)-k):
        prev = soc_ls[i-k:i]
        next = soc_ls[i+1:i+k+1]
        if all([soc_ls[i] > soc for soc in prev]):
            if all([soc_ls[i] > soc for soc in next]):
                if soc_ls[i] > min_value:
                    local_maxima_indices.append(i)

    return local_maxima_indices



# Define the start and the end of a trip, together with the energy
delivered and received
def conclude_trip(end_idx, trip, y1_ed, y2_er, y3_soc):
    proposed_endpoint_soc = y3_soc[end_idx]

    if abs(y3_soc[trip.start_idx] - proposed_endpoint_soc) < 1:
        trip.end_idx = end_idx

    else:  # we must change the extremes of the roundtrip. Either:
        if proposed_endpoint_soc >= y3_soc[trip.start_idx]:  # 1) backtrack
            for j in range(end_idx, trip.start_idx , -1):
                if abs(y3_soc[trip.start_idx] - y3_soc[j]) < 1:
                    trip.end_idx = j
                    break
                trip.end_idx = None
        else:  # or 2) bring the starting index forward
            for s in range(trip.start_idx, end_idx):
                trip.end_idx = end_idx  # endpoint unchanged
                if abs(y3_soc[s] - y3_soc[end_idx]) < 1:
                    trip.start_idx = s
                    break
                trip.start_idx = None  # there may be no way to obtain a
valid trip (e.g. at the end of the SOC line)

    # update the values of energy delivered, energy received, and the SOC
delta of the trip
    trip.energy_delivered = y1_ed[trip.end_idx] - y1_ed[trip.start_idx]
    trip.energy_received = y2_er[trip.end_idx] - y2_er[trip.start_idx]
    trip.set_max_soc(y3_soc)
    trip.set_min_soc(y3_soc)
```

```
# Go from the first to the last time instant, defining the trips and their
energy values, necessary to estimate the RTE
def process_trips():
    # load the data
    df = pd.read_csv("BESS_op_data.csv")
    y1_ed = df["ENERGY_DELIVERED"].to_list()
    y2_er = df["ENERGY_RECEIVED"].to_list()
    y3_soc = df["SOC"].to_list()
    lmax_indices = get_local_max(y3_soc)

    trips_ls = [Trip(0)]

    for i in range(1,len(y3_soc)):
        trip = trips_ls[-1]  # select the current trip
        # if we have to start a new trip from a local maximum: conclude the
previous one, record its energy values
        if i in lmax_indices:
            conclude_trip(i, trip, y1_ed, y2_er, y3_soc)
            if i != max(lmax_indices):
                new_trip = Trip(i)
                trips_ls.append(new_trip)

    trips_ls = [t for t in trips_ls if t.start_idx is not None and
t.end_idx is not None
                and not(isnan(t.energy_received)) and
not(isnan(t.energy_delivered))]  # exclude trip with missing values

    return trips_ls



if __name__ == "__main__":

    trips = process_roundtrips()
    rte_ls = []
    for trip in trips:
        trip_efficiency = trip.energy_delivered / trip.energy_received
        rte_ls.append(trip_efficiency)

    logging.debug("Trips efficiency: " + str(rte_ls))
    print("Average RTE over " + str(len(trips)) + " trips = " + str(
        round(sum(rte_ls) / len(rte_ls) * 100, 2)) + "%")

    soc_deltas = [t.max_soc - t.min_soc for t in trips]
    weights = [delta / sum(soc_deltas) for delta in soc_deltas]

    weighted_avg = sum([weights[i] * rte_ls[i] for i in range(len(trips))])
    print("Average RTE over " + str(len(trips)) + " trips (weighted by each
trip's Δ SOC) = " + str(
        round(weighted_avg * 100, 2)) + "%")
```