SoC Estimation Methods

Peerapol Yuvapoositanon

State of Charge (SoC) Estimation Methods: Overview and Comparison

In battery management systems (BMS), **State of Charge (SoC)** is like a fuel gauge for batteries — it tells how much usable energy remains. However, **you can't measure SoC directly**, like you can measure voltage or current. Instead, we **estimate** it.

- There are **Three major classes** of SoC estimation methods:

Why Estimate SoC?

- The State of Charge (SoC) is a critical parameter in battery management systems (BMS). It represents the remaining charge in the battery as a percentage of its total capacity. Accurate SoC estimation is essential for:
- Ensuring safe and efficient operation of the battery.
- Preventing overcharging or deep discharging, which can damage the battery.
- Providing users with accurate information about battery life.
- However, directly measuring SoC is not possible because it is an internal state of the battery. Instead, SoC must be estimated using indirect measurements like:
  - Current : Used for Coulomb counting (Ah-Integration).
  - Voltage : Related to the Open-Circuit Voltage (OCV), which depends on SoC.
- These measurements are often noisy and subject to inaccuracies, making robust estimation challenging.

Challenges in SoC Estimation

Several factors make SoC estimation difficult:

1. Measurement Noise :

- Current sensors and voltage measurements are prone to noise and errors.
- For example, voltage measurements may fluctuate due to temperature, internal resistance, or transient effects.

2. Model Uncertainty :

- Battery models (e.g., OCV vs. SoC relationships) are approximations and may not perfectly represent real-world behavior.
- Factors like aging, temperature, and hysteresis further complicate the model.

3. Drift in Coulomb Counting :
   - Coulomb counting estimates SoC by integrating current over time. However, small errors in current measurements accumulate over time, leading to drift.
4. Dynamic Behavior :
   - Batteries are dynamic systems where SoC changes continuously based on load conditions, making real-time estimation critical.

Category	Main Methods	Example Techniques
Direct Measurement	Voltage-based	Open Circuit Voltage (OCV)
Coulomb Counting	Current integration	Ampere-hour counting
Model-Based Estimation	Equivalent Circuit Models (ECM)	Kalman Filters (KF, EKF, UKF)
Adaptive / Observer-based*	Online correction during operation	Luenberger Observer, Sliding Mode Observer
Data-Driven / AI-based*	Machine Learning	Neural Networks (NN), Support Vector Machines (SVM)

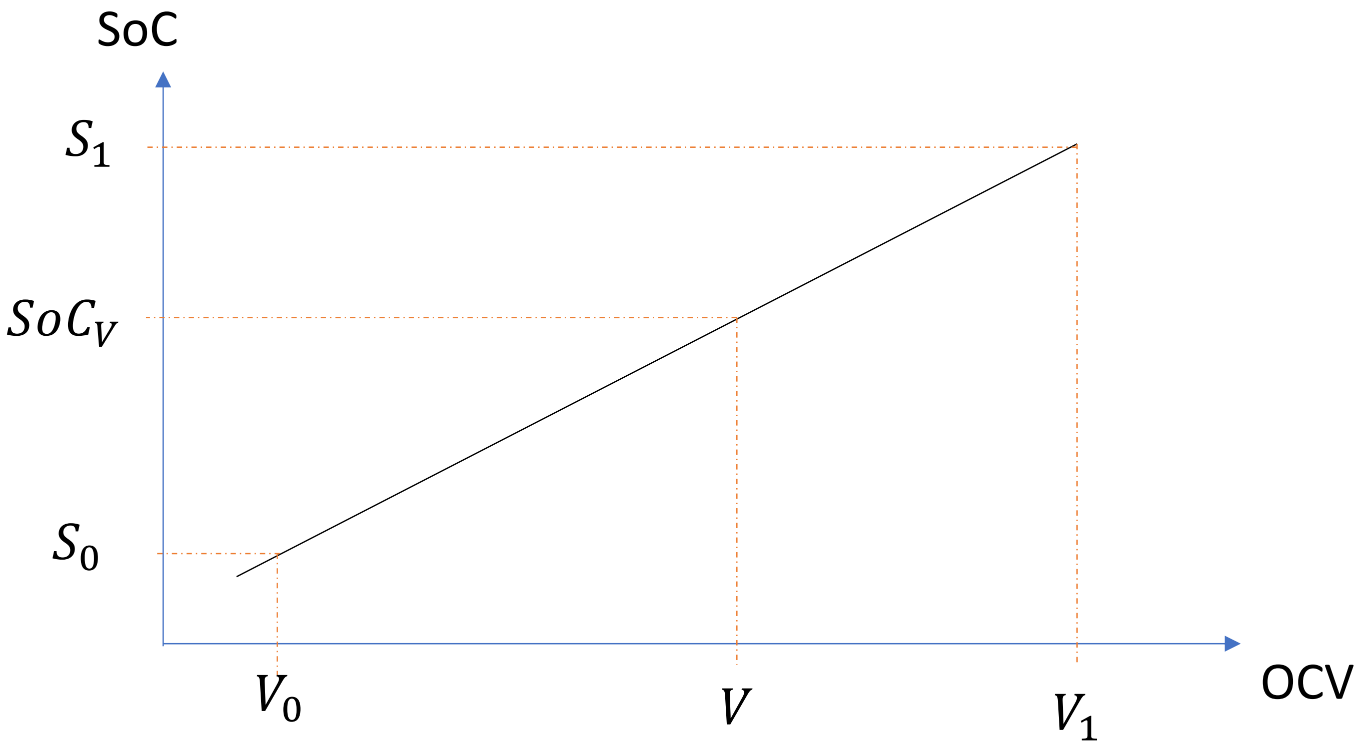
1) Direct Measurement Methods (OCV Method)

Example: Open Circuit Voltage (OCV)

- **Idea:** SoC correlates to battery voltage when the battery is at rest (no current flowing).
- **Pros:**
  - Very simple.
  - No complex modeling required.
- **Cons:**
  - **Only accurate when the battery is at rest** for a long period (several hours sometimes).
  - Not usable during active operation (driving an EV, for example).

OCV Method

$$SoC_V = S_0 + (S_1 - S_0)\frac{V - V_0}{V_1 - V_0}$$



OCV Method

```
[3]: def ocv_to_soc(voltage, ocv_table):
    """
    Estimate SoC from OCV using interpolation.
    voltage: Measured open-circuit voltage (V)
    ocv_table: Dictionary of {voltage: soc} pairs
    """
    voltages = sorted(ocv_table.keys())
    socs = [ocv_table[v] for v in voltages]

    if voltage <= voltages[0]:
        return socs[0]
    if voltage >= voltages[-1]:
        return socs[-1]

    # Linear interpolation
    for i in range(len(voltages) - 1):
        if voltages[i] <= voltage < voltages[i + 1]:
            v0, v1 = voltages[i], voltages[i + 1]
            s0, s1 = socs[i], socs[i + 1]
            soc = s0 + (s1 - s0) * (voltage - v0) / (v1 - v0)
            return soc

# Example OCV-SoC table
ocv_table = {3.6: 0.2, 3.8: 0.5, 4.0: 0.8}
voltage = 3.8

soc = ocv_to_soc(voltage, ocv_table)
print(f"SoC: {soc * 100:.1f}%")
```

```
SoC: 50.0%
```

https://github.com/DrHammerhead/SoC-estimation/blob/main/SOC_Ah_OCV_30april25.ipynb

2) Coulomb Counting (Ah-Integration)

- **Idea:** Integrate the current over time to track how much charge enters or leaves the battery.
- **Formula:**

$$\mathrm{SoC}(t) = SoC(t_0) + \frac{1}{C_{rated}} \int_{t_0}^{1} I(\tau)d\tau$$

- $where\ C_{rated}\ is\ the\ nominal\ capacity,$
- **Pros:**
  - Easy to implement.
  - Good for short periods.
- **Cons:**
  - Accumulates error over time (drift).
  - Needs very accurate current measurement.
  - Sensitive to temperature and aging of the battery.

Ah Integration Method

[1]:
```python
def ah_integration(soc_initial, current, time_step, capacity):
    """
    Estimate SoC using Ah integration.
    soc_initial: Initial SoC (fraction, 0 to 1)
    current: Current in A (positive for charge, negative for discharge)
    time_step: Time interval in hours
    capacity: Battery capacity in Ah
    """
    charge_change = current * time_step  # Ah added or removed
    soc_change = charge_change / capacity
    soc_new = soc_initial + soc_change
    return max(0, min(1, soc_new))  # Clamp between 0 and 1

# Example usage
soc_0 = 0.8  # 80%
current = -5 # 5A discharge
time_step = 2  # 2 hours
capacity = 100  # 100 Ah

soc = ah_integration(soc_0, current, time_step, capacity)
print(f"New SoC: {soc * 100:.1f}%")
```

```
New SoC: 70.0%
```

https://github.com/DrHammerhead/SoC-estimation/blob/main/SOC_Ah_OCV_30april25.ipynb

3) Model-Based Estimation (Kalman Filters)

- **Idea:** Model the battery using electrical equivalents (resistors, capacitors) and apply estimation techniques.
- **Popular Tools:**
  - **Extended Kalman Filter (EKF)**
  - **Unscented Kalman Filter (UKF)**
- **Pros:**
  - Good balance between accuracy and real-time performance.
  - Can correct for some noise and errors.
- **Cons:**
  - Requires good battery models.
  - More complex to implement.
  - Computationally heavier than simple counting.

Why Use a Kalman Filter?

The Kalman Filter is an ideal tool for SoC estimation because it addresses the above challenges effectively:

1. Combines Predictions and Measurements :

   I. The Kalman Filter uses **Coulomb counting (current integration)** to predict SoC and then refines the prediction using voltage measurements .

   II. This combination reduces reliance on either method alone, improving accuracy.

2. Handles Noise :

   I. The filter accounts for uncertainties in both the prediction (process noise) and measurements (measurement noise).

   II. By weighting predictions and measurements based on their respective uncertainties, the Kalman Filter minimizes the impact of noise.

3. Recursive and Real-Time :

   I. The Kalman Filter operates recursively, updating the SoC estimate at each time step. This makes it suitable for real-time applications like BMS.

4. Optimal Estimation :

   I. Under the assumptions of linearity and Gaussian noise, the Kalman Filter provides the minimum mean square error (MMSE) estimate of the state.

- The Extended Kalman Filter (EKF) is a powerful extension of the standard Kalman Filter that allows it to handle nonlinear systems.
- In a nonlinear system, the state transition and measurement models are expressed as:

Problem Setup

State Transition Model

The state evolves according to a nonlinear function:

$$\boldsymbol{x}_k = f(\boldsymbol{x}_{k-1}, \boldsymbol{u}_k) + \boldsymbol{w}_k$$

where:

- $\boldsymbol{x}_k$: State vector at time k.
- $\boldsymbol{u}_k$: Control input (e.g., current in battery systems).
- $f(\cdot)$: Nonlinear function describing the system dynamics.
- $\boldsymbol{w}_k$: Process noise (assumed Gaussian with covariance $\mathbf{Q}k$).

Measurement Model

The measurement is related to the state through another nonlinear function:

$$\boldsymbol{z}_k = h(\boldsymbol{x}_k) + \boldsymbol{v}_k$$

where:

- $\boldsymbol{z}_k$: Measurement vector at time k.
- $h(\cdot)$: Nonlinear function relating the state to the measurement.
- $\boldsymbol{v}_k$: Measurement noise (assumed Gaussian with covariance $\boldsymbol{R}_k$).

Step 1: Prediction

1. Predict the State Estimate : Use the nonlinear state transition function $f(\cdot)$ to predict the next state:

$$\hat{\boldsymbol{x}}_{k|\,k-1} = f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k)$$

1. Linearize the State Transition Model : Compute the Jacobian matrix **F***k* of the state transition function $f(\cdot)$ with respect to the state:

$$\boldsymbol{F}_k = \left.\frac{\partial f}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k}$$

1. Predict the Error Covariance : Update the error covariance matrix **P** using the linearized model:

$$\boldsymbol{P}_{k|k-1} = \boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^T + \boldsymbol{Q}_k$$

Step 2: Update

1.Predict the Measurement : Use the nonlinear measurement function $h(\cdot)$ to predict the measurement:

$$\hat{\boldsymbol{z}}_{k|k-1} = h(\widehat{\boldsymbol{x}}_{k|k-1})$$

2. Linearize the Measurement Model : Compute the Jacobian matrix **H***k* of the measurement function $h(\cdot)$ with respect to the state:

$$\boldsymbol{H}_k = \left.\frac{\partial h}{\partial \boldsymbol{x}}\right|_{\widehat{\boldsymbol{x}}_{k|k-1}}$$

3. Compute the Kalman Gain : Calculate the Kalman gain **K***k* using the linearized measurement model:

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^T\left(\boldsymbol{H}_k\boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^T + \boldsymbol{R}_k\right)^{-1}$$

4. Update the State Estimate : Correct the predicted state using the innovation (difference between actual and predicted measurements):

$$\widehat{\boldsymbol{x}}_{k|k} = \widehat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_{k|k-1})$$

5. Update the Error Covariance : Update the error covariance matrix:

$$\boldsymbol{P}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}_{k|k-1}$$

Application to Battery SoC Estimation

1. State Transition Model : The state (SoC) evolves based on Coulomb counting:

$$SoC_k = SoC_{k-1} - CI_k\Delta t$$

This is linear, so no linearization is needed here.

2. Measurement Model : The voltage measurement is related to SoC through the nonlinear OCV function:

$$V_k = OCV(SoC_k) - I_kR_{int} + v_k$$

Here:

- $OCV(SoC_k)$ is nonlinear (e.g., quadratic or lookup table).
- The derivative $\frac{\partial OCV}{\partial SoC}$ is used to compute the Jacobian $\boldsymbol{H}_k$.

3. Linearization : The derivative of the OCV function is computed at each step:

$$\boldsymbol{H}_k = \frac{\partial h}{\partial SoC} = \frac{\partial OCV}{\partial SoC}$$

- This linearization allows the EKF to handle the nonlinear relationship between SoC and voltage

SoC_Kalman_30april25.ipynb

jupyter SoC_Kalman_30april25 Last Checkpoint: 48 seconds ago

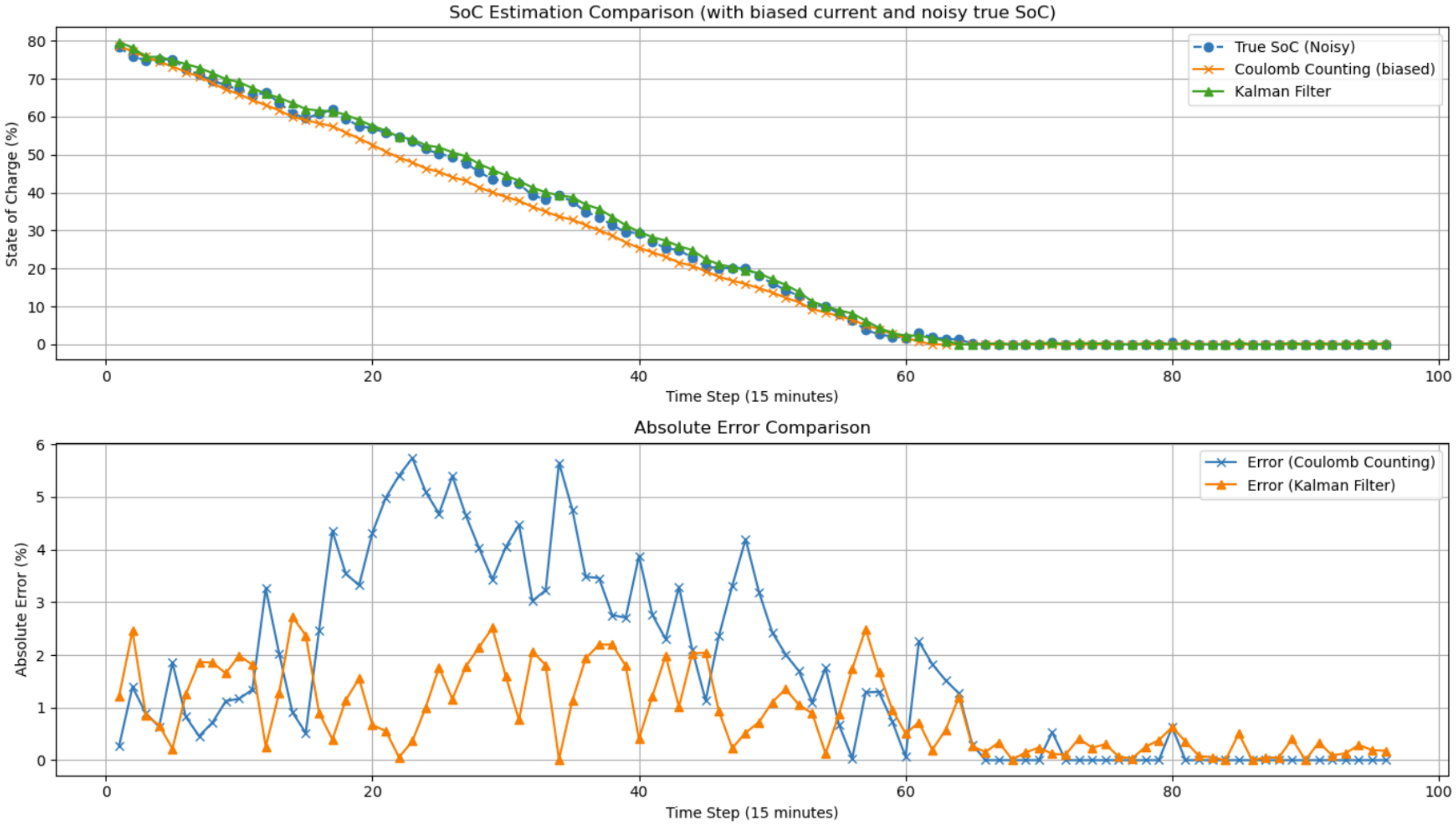
File Edit View Run Kernel Settings Help Truste

Code JupyterLab Python 3 (ipykernel)

```
        soc_initial=0.8,
        capacity=100,           # 100 Ah
        r_int=0.05,             # Internal resistance
        process_noise=1e-5,     # Increased process noise for better adaptability
        measurement_noise=1e-4 # Slightly increased measurement noise for better adaptation
    )

    # Simulation parameters
    np.random.seed(0)  # Reproducibility
    current_profile = np.random.uniform(3, 7, size=96)  # 96 samples (every 15 minutes)
    dt = 0.25  # Time step (15 minutes)
    true_soc = 0.8  # Start at 80%

    time_steps = []
    true_socs = []
    predicted_socs = []
    updated_socs = []
    coulomb_counting_socs = []

    soc_coulomb = 0.8
```

https://github.com/DrHammerhead/SoC-estimation/blob/main/SoC_Kalman_30april25.ipynb

```
Mean Absolute Error (Coulomb Counting): 1.73%
Mean Absolute Error (Kalman Filter): 0.91%
```



SoC Estimation Comparison (with biased current and noisy true SoC)
State of Charge (%)
Time Step (15 minutes)
True SoC (Noisy)
Coulomb Counting (biased)
Kalman Filter
0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
Absolute Error Comparison
Absolute Error (%)
Time Step (15 minutes)
Error (Coulomb Counting)
Error (Kalman Filter)
0
1
2
3
4
5
6
0
20
40
60
80
100

How Temperature Affects Coulomb Counting

a. Battery Capacity Is Temperature-Dependent

Battery capacity CCC **decreases at low temperatures** because:

- Electrochemical reactions slow down
- Internal resistance increases
- Some capacity becomes inaccessible

Example:

A 100Ah battery might only deliver ~80Ah at -10°C.

➡️ **Coulomb counting will overestimate SoC at low temps**, since it assumes a fixed 100Ah.

b. Current Sensor Drift

Shunt-based or Hall-effect current sensors can **drift with temperature**, leading to:

- Small, systematic errors in current
- Which accumulate into significant SoC errors over time

➡️ **Thermal compensation** is often needed in sensor electronics.

c. Self-Discharge and Parasitic Losses

These increase with temperature:

- Coulomb counting doesn't track self-discharge
- High temperatures make this worse

➡️ Coulomb counting **underestimates** SoC at high temps if not corrected.

d. Internal Resistance & Peukert Effect

Though less significant for simple Coulomb counting:

- Resistance increases at low T
- High internal resistance causes greater power loss

Temperature Effects

Temperature	Capacity Impact	Coulomb Counting Error
Cold (< 10°C)	↓↓ capacity	**Overestimates SoC**
Hot (> 35°C)	Minor ↓ capacity; ↑ self-discharge	**Underestimates SoC (slowly)**

How to Improve Accuracy

1. **Use a Temperature-Compensated Capacity**:

$$C(T) = C_{rated} \cdot f(T)$$

where $f(T)$ is a correction factor from testing or datasheets.

Temperature-Compensated Current Sensing:

1. Calibrate or use ICs with internal compensation (e.g., INA219/INA226).
2. Apply digital corrections.

Fuse with OCV method

Use **OCV** to merge Coulomb counting and voltage at rest.

SoC_AH_TempandOCV_30april25.ipynb

jupyter SoC_AH_TempandOCV_30april25 Last Checkpoint: 13 seconds ago

File Edit View Run Kernel Settings Help Trusted

Markdown JupyterLab Python 3 (ipykernel)

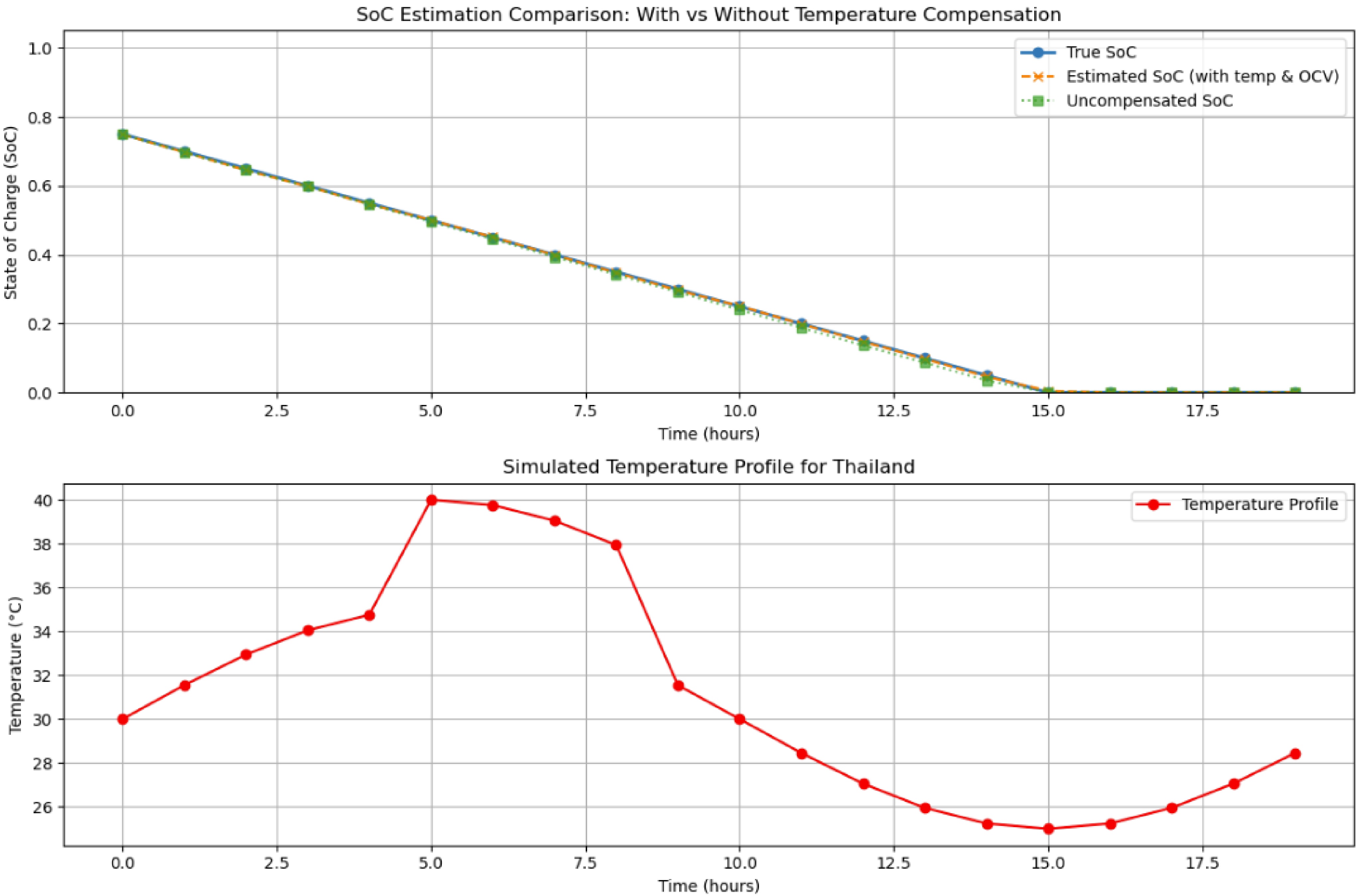
Battery State of Charge (SoC) Estimation Using Temperature-Compensated Ah Integration with OCV Recalibration

This script simulates the estimation of a battery's State of Charge (SoC) under varying temperature conditions using a temperature-compensated Ampere-hour (Ah) integration method combined with periodic Open Circuit Voltage (OCV) recalibration. The implementation accounts for the following key factors:

1. **Temperature Compensation**:
   - Adjusts the battery's effective capacity based on operating temperature.
   - Models reduced capacity at lower temperatures (e.g., 0.5% reduction per °C below 25°C).
   - Ensures accurate SoC estimation in environments with fluctuating temperatures.
2. **Ampere-Hour (Ah) Integration**:
   - Tracks the charge flowing into or out of the battery over time.
   - Updates the SoC estimate based on the effective capacity adjusted for temperature.
3. **OCV-Based Recalibration**:
   - Periodically measures the Open Circuit Voltage (OCV) during rest periods to provide an absolute reference for SoC estimation.
   - Blends Coulomb-counted SoC with OCV-based SoC using a weighted average to reduce drift and improve accuracy.
4. **Realistic Simulation**:
   - Includes a dynamic temperature profile to simulate real-world environmental conditions (e.g., Thailand's high-temperature climate).
   - Incorporates noisy and biased current measurements to reflect practical measurement errors.
   - Provides a comparison between temperature-compensated SoC estimation and uncompensated Coulomb counting.
5. **Performance Metrics**:
   - Calculates the Mean Absolute Percentage Error (MAPE) to quantitatively evaluate the accuracy of the compensated and uncompensated SoC estimates

https://github.com/DrHammerhead/SoC-estimation/blob/main/SoC_AH_TempandOCV_30april25.ipynb

```
MAPE (Estimated SoC): 1.13%
MAPE (Uncompensated SoC): 4.82%
```



Explanation of SoC_AH_TempandOCV_30april25.ipynb

1. Temperature-Corrected Capacity

python

```python
def temperature_corrected_capacity(nominal_capacity, temperature_c):
    """
    Adjust capacity based on temperature.
    Capacity decreases ~0.5% per °C below 25°C.
    """
    if temperature_c < 25:
        reduction_factor = 1 - 0.005 * (25 - temperature_c)
    else:
        reduction_factor = 1  # No gain above 25°C
    return nominal_capacity * reduction_factor
```

Explanation:

- Purpose : This function adjusts the battery's nominal capacity based on the operating temperature. At lower temperatures, the battery's usable capacity decreases due to reduced chemical activity.
- Logic :
  - If the temperature is below 25°C, the capacity is reduced by 0.5% for every degree Celsius below 25°C :

$$reduction\_factor = 1 - 0.005 \cdot (25 - temperature\_c)$$

  - If the temperature is above or equal to 25°C , the capacity remains unchanged
- Output : The effective capacity at the given temperature, adjusted from the nominal capacity.

2. OCV-to-SoC Conversion

python

```python
def ocv_to_soc(ocv):
    """
    Dummy function to map OCV to SoC.
    Replace this with real battery OCV-SoC curve fitting.
    """
    ocv_min = 3.0
    ocv_max = 4.2
    return max(0, min(1, (ocv - ocv_min) / (ocv_max - ocv_min)))
```

- Explanation:
- Purpose : This function converts the measured Open Circuit Voltage (OCV) into an estimated State of Charge (SoC). In real-world applications, this relationship is nonlinear and specific to the battery chemistry.
- Logic :
- A dummy linear mapping is used here for simplicity:
- ocv_min = 3.0 V: The minimum voltage corresponding to 0% SoC.
- ocv_max = 4.2 V: The maximum voltage corresponding to 100% SoC.

- The formula calculates the normalized SoC:

$$SoC = \frac{OCV - OCV_{\min}}{OCV_{\max} - OCV_{\min}}$$

- The result is clamped between 0 and 1 to ensure valid SoC values.
- Note : In practice, this function should be replaced with a more accurate model (e.g., polynomial or lookup table) based on experimental data for the specific battery.

3. Recalibration of SoC

```python
def recalibrate_soc(soc_coulomb, ocv, alpha=0.9):
    """
    Blend Coulomb-counted SoC with OCV-based SoC estimate.
    alpha: weight given to Coulomb result (e.g., 0.9 means 90% trust in integration)
    """
    soc_ocv = ocv_to_soc(ocv)
    return alpha * soc_coulomb + (1 - alpha) * soc_ocv
```

- Explanation:
- Purpose : This function recalibrates the SoC estimate by blending the Coulomb-counted SoC (soc_coulomb) with the OCV-based SoC (soc_ocv).
- Logic :
- The OCV-based SoC is calculated using the ocv_to_soc function.
- A weighted average is used to combine the two estimates:

$$SoC\_new = \alpha \cdot SoC\_coulomb + (1 - \alpha) \cdot SoC\_ocv$$

- alpha determines how much weight is given to the Coulomb-counted SoC. For example:
- alpha = 0.9: 90% trust in Coulomb counting, 10% trust in OCV.
- alpha = 0.5: Equal trust in both methods.
- Use Case : This recalibration is particularly useful during rest periods when the OCV provides a reliable absolute reference for SoC.

4. Ah Integration with Temperature Correction and Recalibration

python

```python
def ah_integration_temp(soc_initial, current, time_step, nominal_capacity, temperature_c, is_res
    """
    Estimate SoC using Ah integration with temperature correction and optional OCV recalibration
    """
    capacity_effective = temperature_corrected_capacity(nominal_capacity, temperature_c)
    charge_change = current * time_step  # Ah
    soc_change = charge_change / capacity_effective
    soc_new = soc_initial + soc_change
    soc_new = max(0, min(1, soc_new))  # Clamp SoC

    if is_rest_period and ocv_measured is not None:
        # Apply recalibration
        soc_new = recalibrate_soc(soc_new, ocv_measured, alpha=0.9)

    return soc_new
```

Explanation:

- Purpose : This function estimates the new SoC by integrating the current over time while accounting for temperature effects and optionally recalibrating using OCV.
- Steps :
- 1) Temperature Correction :
  - The effective capacity is calculated using the temperature_corrected_capacity function.
- 2) Charge Change Calculation :
  - The change in charge (charge_change) is calculated as:

$$charge\_change = current \cdot time\_step$$

- Positive current indicates charging, while negative current indicates discharging.

- 3) SoC Update :
- The change in SoC (soc_change) is proportional to the charge change divided by the effective capacity:

$$SoC\_change = \frac{charge\_change}{capacity\_effective}$$

- The new SoC is updated by adding the change to the initial SoC:

$$SoC\_new = SoC\_initial + SoC\_change$$

- The result is clamped between 0 and 1 to ensure valid SoC values.

- 4) Recalibration During Rest :
- If the battery is in a rest period (is_rest_period = True) and an OCV measurement is available (ocv_measured), the SoC is recalibrated using the recalibrate_soc function.
- Output : The updated SoC value after integrating the current and applying corrections.

5. Uncompensated Coulomb Counting

python

```python
def ah_integration_uncompensated(soc_initial, current, time_step, nominal_capacity):
    """
    Traditional Coulomb counting without temp compensation or recalibration.
    """
    charge_change = current * time_step
    soc_change = charge_change / nominal_capacity
    soc_new = soc_initial + soc_change
    return max(0, min(1, soc_new))
```

Explanation:

- Purpose : This function performs traditional Coulomb counting without accounting for temperature effects or OCV recalibration.
- Logic :
  - The change in SoC is calculated using the nominal capacity instead of the temperature-corrected capacity.
  - No recalibration is applied during rest periods.
- Use Case : This serves as a baseline for comparison with the compensated method.

6. Realistic High-Temperature Profile

python

```
def temperature_profile_function(t):
    """
    Simulate a realistic high-temperature profile for Thailand.
    Example: Baseline at 30°C, small fluctuations (±2°C), and occasional peaks to 35°C.
    """
    baseline = 30  # Typical ambient temperature in Thailand
    fluctuation = 2 * np.sin(2 * np.pi * t / steps)  # Small sinusoidal fluctuations
    peak = 5 if t in [5, 6, 7, 8] else 0  # Occasional peak to 35°C during midday
    return baseline + fluctuation + peak
```

Explanation:

- Purpose : This function simulates a realistic temperature profile for Thailand, where temperatures are consistently high with minor fluctuations and occasional peaks.
- Logic :
  - Baseline : Starts at 30°C , representing the typical ambient temperature in Thailand.
  - Fluctuations : Adds small sinusoidal variations (±2°C) to simulate minor temperature changes throughout the day.
  - Peaks : Introduces a 5°C increase during steps 5–8 to simulate the hottest part of the day.
- This profile reflects the warm and relatively stable climate of Thailand.

7. Simulation Loop

```python
for t in range(steps):
    # Get the temperature at the current time step
    temperature = temperature_profile_function(t)
    temperature_profile.append(temperature)

    # Effective capacity at the current temperature
    effective_capacity = temperature_corrected_capacity(capacity, temperature)

    # Rest period logic
    is_rest = (t % 5 == 0 and t != 0)  # Rest every 5 steps
    ocv_measured = 3.0 + 1.2 * soc_true if is_rest else None

    # Simulate noisy and biased current measurement
    measured_current = current * current_bias + np.random.normal(0, current_noise_std)

    # Improved estimation with temperature correction and OCV recalibration
    soc_est = ah_integration_temp(soc_est, measured_current, time_step, capacity, temperature, i

    # Uncompensated Coulomb counting
    soc_uncomp = ah_integration_uncompensated(soc_uncomp, measured_current, time_step, capacity)

    # Simulated true SoC (affected by temperature)
    soc_true += (current * time_step) / effective_capacity
    soc_true = max(0, min(1, soc_true))

    # Record data
    time_list.append(t)
    true_soc_list.append(soc_true)
    est_soc_list.append(soc_est)
    uncomp_soc_list.append(soc_uncomp)
```

- Explanation:
- Temperature Update :
  - The temperature at each time step is determined using the temperature_profile_function.
- Effective Capacity :
  - The effective capacity is recalculated based on the current temperature.
- Rest Periods :
  - Every 5th step, the battery enters a rest period (is_rest = True), during which the OCV is measured and used for recalibration.
- Noisy and Biased Current Measurement :
  - The measured current includes a 2% bias and random noise to simulate real-world inaccuracies.
- True SoC Update :
  - The true SoC is updated based on the effective capacity at the simulated temperature.
- Record Data :
  - The true, estimated, and uncompensated SoC values are recorded for plotting

8. MAPE Calculation

python

```python
def calculate_mape(true_socs, estimated_socs):
    """
    Calculate Mean Absolute Percentage Error (MAPE).
    """
    errors = [
        abs((true_soc - est_soc) / true_soc) * 100
        for true_soc, est_soc in zip(true_socs, estimated_socs)
        if true_soc > 0  # Avoid division by zero
    ]
    return sum(errors) / len(errors) if errors else 0

mape_est = calculate_mape(true_soc_list, est_soc_list)
mape_uncomp = calculate_mape(true_soc_list, uncomp_soc_list)

print(f"MAPE (Estimated SoC): {mape_est:.2f}%")
print(f"MAPE (Uncompensated SoC): {mape_uncomp:.2f}%")
```

Explanation:

- Purpose : This function calculates the Mean Absolute Percentage Error (MAPE) between the true SoC and the estimated SoC values.
- Logic :
  - For each time step, the absolute percentage error is calculated as:

$$Error = \frac{|True\ SoC - Estimated\ SoC|}{|True\ SoC|} \times 100$$

  - The errors are averaged across all time steps to compute the MAPE.
  - Division by zero is avoided by skipping any time steps where the true SoC is zero.
- Output : The MAPE values for both the compensated and uncompensated SoC estimates.

9. Plotting

```python
plt.figure(figsize=(12, 8))

# Plot SoC comparison
plt.subplot(2, 1, 1)
plt.plot(time_list, true_soc_list, label="True SoC", marker='o', linewidth=2)
plt.plot(time_list, est_soc_list, label="Estimated SoC (with temp & OCV)", marker='x', linestyle
plt.plot(time_list, uncomp_soc_list, label="Uncompensated SoC", marker='s', linestyle=':', alpha
plt.xlabel("Time (hours)")
plt.ylabel("State of Charge (SoC)")
plt.title("SoC Estimation Comparison: With vs Without Temperature Compensation")
plt.legend()
plt.grid(True)
plt.ylim(0, 1.05)

# Plot temperature profile
plt.subplot(2, 1, 2)
plt.plot(time_list, temperature_profile, label="Temperature Profile", marker='o', color='red')
plt.xlabel("Time (hours)")
plt.ylabel("Temperature (°C)")
plt.title("Simulated Temperature Profile for Thailand")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Explanation:

- First Subplot :
  - Compares the true SoC, estimated SoC (with temperature compensation and recalibration), and uncompensated SoC over time.
- Second Subplot :
  - Shows the simulated temperature profile over time, reflecting the realistic high-temperature conditions of Thailand.