State of Charge Estimation in Battery Management System Applications on Lithium Cells

Ömer Takkin
No.: 2516987
Electric and Electronic Engineer (EEE)
Middle East Technical University (METU)
Ankara, Turkey
omer.takkin@gmail.com

I. Introduction

Battery-powered systems (e.g. electric vehicles and grid storage) depend on sophisticated Battery Management Systems (BMSs) to ensure safe, reliable operation. A BMS continuously monitors each cell's voltage, current, and temperature and enforces charge/discharge limits so that cells never exit their tight safe-operating area. Lithium-ion batteries in particular offer very high energy density but allow little margin for error; violating the safe limits can rapidly compromise pack health or even trigger catastrophic failures such as thermal runaway. Within the BMS, estimating the battery's state of charge (SOC) – the remaining capacity – is a core function. Accurate SOC knowledge is essential for proper charge control and range prediction. As noted in the literature, precise SOC estimation "improves the system performance and reliability" and prevents the battery from being over-charged or over-discharged, thereby avoiding unplanned shutdowns or cell damage. In other words, knowing SOC exactly maximizes energy utilization and protects against the hazards of improper charging or discharging. Because there are no available tool for sensing SOC directly. The Extended Kalman Filter (EKF) is a powerful model-based estimator suited to this task. In summary, modern battery systems' critical safety, reliability, and performance requirements strongly motivate the implementation of EKF-based SOC estimation as a core BMS capability.

II. Problem Definition

We need to make some definitions and later introduce a model for our problem. We need to know SOC.

A. Definitions

1) Fully Charged: A cell is fully charged when open-circuit voltage (OCV) reaches a manufacturer-specified voltage $v_h(T)$. For lithium-manganese-oxide $v_h(25\,^\circ\mathrm{C}) = 4.2V$.

2) Fully Discharged: A cell is fully discharged when OCV reaches a manufacturer-specified voltage $v_l(T)$. For lithium-manganese-oxide $v_l(25\,^\circ\mathrm{C}) = 3.0V$.

3) Total Capacity: The total capacity Q of a cell is the quantity of charge removed from a cell as it is brought from a fully charged state to a fully discharged state. Units for Q are coulombs, ampere-hour and miliampere-hour.

4) Discharge Capacity: The discharge capacity $Q_{[rate]}$ of a cell is the quantity of charge removed from a cell as it is discharged at a constant rate from a fully charged state to a fully discharged state.

5) Nominal Capacity: The nominal capacity Q_{norm} of a cell is a manufacturer-specified quantity that indicates the amount of charge that the cell is rated to hold.

6) Residual Capacity: The residual capacity of a cell is the quantity of charge that would be removed from a cell if it were brought from its present state to a fully discharged state.

7) State of Charge: The state of charge (SOC) of a cell is the ratio of its residual capacity to its total capacity. The present average lithium concentration stoichiometry is:

$$\theta_k = c_{s,avg,k}/c_{s,max} \tag{1}$$

The equation for SOC is:

$$z_k = \frac{\theta_k - \theta_{0\%}}{\theta_k - \theta_{100\%}} \tag{2}$$

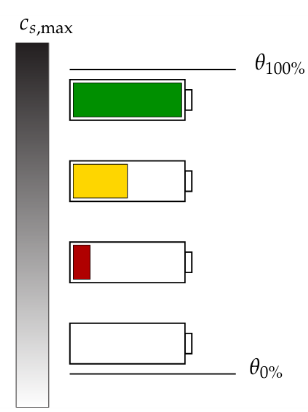

Fig. 1. Relationship between negative-electrode average concentration and cell SOC.

In modern technology, there is no way to measure the concentration of the average lithium. Therefore, we must estimate SOC using only cell terminal voltage, current, and temperature.

B. Model

We need to obtain a mathematical model to use the Kalman filter. I decided to use an enhanced self-correcting cell model equivalent circuit that is shown in Figure 2. State equation for this model is given below. Derivation of equations is not scope of this paper. In our model we will only use R_1 and C_1 to model diffusion voltage.

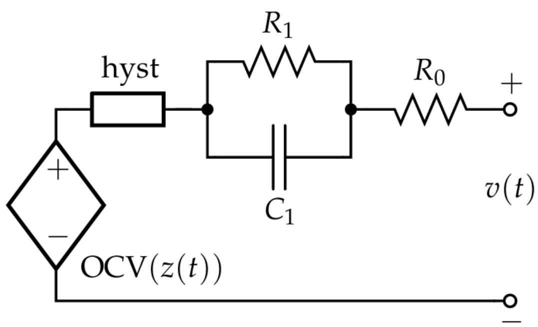

Fig. 2. The enhanced self-correcting cell model equivalent circuit

States of the OCV cell model are listed in Table 1. Inputs of the OCV cell model are listed in Table 1. Parameters of the OCV cell model are listed in Table 1.

TABLE I
MODEL STATES

States	Description
$z[k]$	State of charge at instance k
$i_R[k]$	Resistor current vector at instance k
$h[k]$	Hysteresis voltage at instance k

TABLE II
MODEL INPUTS

Variable	Description
$i[k]$	Terminal current magnitude at instance k
$\text{sgn}(i[k])$	Terminal current direction at instance k

TABLE III
MODEL PARAMETERS

Parameter	Description
OCV	OCV vector at which $SOC0$ and $SOCrel$ are stored
$OCV0$	Vector of OCV versus SOC at $0\,^{\circ}$C [V]
$OCVrel$	Vector of change in OCV versus SOC per $^{\circ}$C [V/$^{\circ}$C]
SOC	SOC vector at which $OCV0$ and $OCVrel$ are stored
$SOC0$	Vector of SOC versus OCV at $0\,^{\circ}$C
$SOCrel$	Vector of change in SOC versus OCV per $^{\circ}$C [1/$^{\circ}$C]
T	Temperatures at which dynamic parameters are stored [$^{\circ}$C]
Q	Capacity at each temperature [Ah]
M	Hysteresis voltage parameter [V]
M_0	Instantaneous hysteresis voltage parameter [V]
R_0	Series Resistance parameter [Ω]
R_iC_i	The R-C time constant parameter [sec]
R_i	Resistance of the R-C parameter [Ω]

State-space representation of a cell is shown in equation 3.

$$\begin{bmatrix} z[k+1] \\ i_R[k+1] \\ h[k+1] \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A_{RC} & 0 \\ 0 & 0 & A_H[k] \end{bmatrix} \begin{bmatrix} z[k] \\ i_R[k] \\ h[k] \end{bmatrix} + \begin{bmatrix} -\frac{\eta \Delta t}{Q} & 0 \\ B_{RC} & 0 \\ 0 & (A_H[k]-1) \end{bmatrix} \begin{bmatrix} i[k] \\ \text{sgn}(i[k]) \end{bmatrix} \tag{3}$$

$$HysteresisVoltage = M_0 s[k] + M h[k] \tag{4}$$

$$s[k] = \begin{cases} sgn(i[k]), & \text{if } |i[k]| > 0 \\ s[k-1], & \text{otherwise} \end{cases} \tag{5}$$

$$A_H[k] = exp\left(-\left|\frac{\eta[k] i[k] \gamma \Delta t}{Q}\right|\right) \tag{6}$$

$$A_{RC} = \begin{bmatrix} F_1 & 0 & \dots \\ 0 & F_2 & \\ \vdots & & \ddots \end{bmatrix} \tag{7}$$

$$B_{RC} = \begin{bmatrix} (1-F_1) \\ (1-F_2) \\ \vdots \end{bmatrix} \tag{8}$$

$$F_i = exp\left(\frac{-\Delta t}{R_i C_i}\right) \tag{9}$$

III. Description of the EKF and Background

The Kalman filter is an algorithm that computes a provably optimal state despite uncertainties. The Kalman filter is a special case of a general solution framework known as sequential probabilistic inference. Equations for the Kalman filter are:

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \mathbf{w}_{k-1}) \tag{10}$$
$$\mathbf{y}_k = h(\mathbf{x}_k, \mathbf{u}_k, \mathbf{v}_k) \tag{11}$$

$\mathbf{x}_k$ is the model state vector. $\mathbf{u}_k$ is the input of the system, which is deterministic or measured. $\mathbf{w}_k$ is the unknown and unmeasurable process-noise random input signal. $\mathbf{v}_k$ is the unknown and unmeasurable sensor-noise random input signal.

In our case, $\mathbf{u}_k$ is the measured cell input current. $\mathbf{y}_k$ is noisy measurement of cell voltage.

A. Implementing Extended Kalman Filter

To use the extended Kalman Filter, we should make some assumptions. The first assumption is

$$\mathbb{E}[\mathbf{fn}(\mathbf{x})] \approx \mathbf{fn}(\mathbb{E}[\mathbf{x}])$$

The second assumption is that when computing covariance estimates, EKF uses a truncated Taylor-series expansion to linearize the system equations around the present operating point. This is the why EKF works best on mild nonlinearities.

$$\mathbf{fn}(\mathbf{x}) \approx \mathbf{fn}(\mathbf{x})|_{\mathbf{a}} + \frac{d}{d\mathbf{x}} \mathbf{fn}(\mathbf{x})|_{\mathbf{a}} (\mathbf{x} - \mathbf{a})$$

1) Step 1: State Prediction Time Update: Calculate predicted state $\hat{\mathbf{x}}_{\mathbf{k}}^{-}$ based on past state $\mathbf{x_{k-1}}$ input $\mathbf{u_{k-1}}$, and noise $\mathbf{w_{k-1}}$ that given set of outputs $\mathbb{Y}_{k-1} = \{\mathbf{y_0}, \mathbf{y_1}, \dots, \mathbf{y_{k-1}}\}$. In that step we are using first assumption.

$$\hat{\mathbf{x}}_{\mathbf{k}}^{-} = \mathbb{E}[f(\mathbf{x_{k-1}}, \mathbf{u_{k-1}}, \mathbf{w_{k-1}})|\mathbb{Y}_{k-1}] \tag{12}$$
$$\approx f(\hat{\mathbf{x}}_{\mathbf{k-1}}^{+}, \mathbf{u_{k-1}}, \bar{\mathbf{w}}_{\mathbf{k-1}}) \tag{13}$$

2) Step 2: Error Covariance Time Update: We are define error at prediction as $\tilde{\mathbf{x}}_{\mathbf{k}}^{-}$.

$$\tilde{\mathbf{x}}_{\mathbf{k}}^{-} = \mathbf{x_k} - \hat{\mathbf{x}}_{\mathbf{k}}^{-} \tag{14}$$
$$= f(\mathbf{x_{k-1}}, \mathbf{u_{k-1}}, \mathbf{w_{k-1}}) - f(\hat{\mathbf{x}}_{\mathbf{k-1}}^{+}, \mathbf{u_{k-1}}, \bar{\mathbf{w}}_{\mathbf{k-1}}) \tag{15}$$

Then we use second assumption to break $\mathbf{x_k}$ into pieces.

$$\mathbf{x_k} = f(\hat{\mathbf{x}}_{\mathbf{k-1}}^{+}, \mathbf{u_{k-1}}, \bar{\mathbf{w}}_{\mathbf{k-1}}) \tag{16}$$
$$+ \hat{\mathbf{A}}_{\mathbf{k-1}}(\mathbf{x_{k-1}} - \hat{\mathbf{x}}_{\mathbf{k-1}}^{+}) \tag{17}$$
$$+ \hat{\mathbf{B}}_{\mathbf{k-1}}(\mathbf{w_{k-1}} - \bar{\mathbf{w}}_{\mathbf{k-1}}) \tag{18}$$
$$\hat{\mathbf{A}}_{\mathbf{k-1}} = \frac{d}{d\mathbf{x_{k-1}}} f(\mathbf{x_{k-1}}, \mathbf{u_{k-1}}, \mathbf{w_{k-1}})|_{\{\hat{x}_{k-1}^{+}, u_{k-1}, \bar{w}_{k-1}\}} \tag{19}$$
$$\hat{\mathbf{B}}_{\mathbf{k-1}} = \frac{d}{d\mathbf{w_{k-1}}} f(\mathbf{x_{k-1}}, \mathbf{u_{k-1}}, \mathbf{w_{k-1}})|_{\{\hat{x}_{k-1}^{+}, u_{k-1}, \bar{w}_{k-1}\}} \tag{20}$$

In that way we are able to write error at prediction as error at prediction of previous instances as shown in equation 21.

$$\tilde{\mathbf{x}}_{\mathbf{k}}^{-} \approx \hat{\mathbf{A}}_{\mathbf{k-1}} \tilde{\mathbf{x}}_{\mathbf{k-1}}^{-} + \hat{\mathbf{B}}_{\mathbf{k-1}} \tilde{\mathbf{w}}_{\mathbf{k-1}}^{-} \tag{21}$$

Now, we can find the prediction error covariance. This is also Bayesian mean square error of estimation. Again, we want to write error covariance as equation of error covariance of previous instance as shown in equation 25.

$$\mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}} = \mathbb{E}[(\tilde{x}_k)(\tilde{x}_k)^T] \tag{22}$$
$$\approx \mathbb{E}[(\hat{\mathbf{A}}_{\mathbf{k-1}} \tilde{\mathbf{x}}_{\mathbf{k-1}}^{-} + \hat{\mathbf{B}}_{\mathbf{k-1}} \tilde{\mathbf{w}}_{\mathbf{k-1}}^{-}) \tag{23}$$
$$\times (\hat{\mathbf{A}}_{\mathbf{k-1}} \tilde{\mathbf{x}}_{\mathbf{k-1}}^{-} + \hat{\mathbf{B}}_{\mathbf{k-1}} \tilde{\mathbf{w}}_{\mathbf{k-1}}^{-})^T] \tag{24}$$
$$= \hat{\mathbf{A}}_{\mathbf{k-1}} \mathbf{C}_{\tilde{\mathbf{x}},\mathbf{k-1}}^{+} + \hat{\mathbf{B}}_{\mathbf{k-1}} \mathbf{C}_{\tilde{\mathbf{w}}} \hat{\mathbf{B}}_{\mathbf{k-1}}^{\mathbf{T}} \tag{25}$$

3) Step 3: Predict System output: Calculate predicted output of system $\hat{\mathbf{y}}_{\mathbf{k}}$ based on current state $\mathbf{x_k}$ input $\mathbf{u_k}$, and sensor noise $\mathbf{v_k}$ that given set of outputs $\mathbb{Y}_{k} = \{\mathbf{y_0}, \mathbf{y_1}, \dots, \mathbf{y_k}\}$. Inthat step we are using first assumption.

$$\hat{\mathbf{y}}_{\mathbf{k}} = \mathbb{E}[h(\mathbf{x_k}, \mathbf{u_k}, \mathbf{v_k})|\mathbb{Y}_{k-1}] \tag{26}$$
$$\approx h(\hat{\mathbf{x}}_{\mathbf{k}}^{-}, \mathbf{u_k}, \bar{\mathbf{v}}_{\mathbf{k}}) \tag{27}$$

4) Step 4: Estimator Gain Matrix: We are define error at prediction as $\tilde{\mathbf{y}}_{\mathbf{k}}$.

$$\tilde{\mathbf{y}}_{\mathbf{k}} = \mathbf{y_k} - \hat{\mathbf{y}}_{\mathbf{k}}^{-} \tag{28}$$
$$= h(\mathbf{x_k}, \mathbf{u_k}, \mathbf{v_k}) - h(\hat{\mathbf{x}}_{\mathbf{k}}^{-}, \mathbf{u_k}, \bar{\mathbf{v}}_{\mathbf{k}}) \tag{29}$$

Then we use second assumption to break $\mathbf{y_k}$ into pieces.

$$\mathbf{y_k} \approx h(\hat{\mathbf{x}}_{\mathbf{k}}^{-}, \mathbf{u_k}, \bar{\mathbf{v}}_{\mathbf{k}}) \tag{30}$$
$$+ \hat{\mathbf{C}}_{\mathbf{k}}(\mathbf{x_k} - \hat{\mathbf{x}}_{\mathbf{k}}^{-}) \tag{31}$$
$$+ \hat{\mathbf{D}}_{\mathbf{k}}(\mathbf{v_k} - \bar{\mathbf{v}}_{\mathbf{k}}^{-}) \tag{32}$$
$$\hat{\mathbf{C}}_{\mathbf{k}} = \frac{d}{d\mathbf{x_k}} h(\mathbf{x_k}, \mathbf{u_k}, \mathbf{v_k})|_{\{\hat{x}_k^{-}, u_k, \bar{v}_k\}} \tag{33}$$
$$\hat{\mathbf{D}}_{\mathbf{k}} = \frac{d}{d\mathbf{v_k}} h(\mathbf{x_k}, \mathbf{u_k}, \mathbf{v_k})|_{\{\hat{x}_k^{-}, u_k, \bar{v}_k\}} \tag{34}$$

In that way we are able to write error at output as error at output of previous instances as shown in equation 35.

$$\tilde{\mathbf{y}}_{\mathbf{k}} = \hat{\mathbf{C}}_{\mathbf{k}} \tilde{\mathbf{x}}_{\mathbf{k}}^{-} + \hat{\mathbf{D}}_{\mathbf{k}} \tilde{\mathbf{v}}_{\mathbf{k}} \tag{35}$$

Now, we can find the output error covariance and correlation between state error and output error as shown in equations 36 and 37.

$$\mathbf{\Sigma}_{\tilde{\mathbf{y}},\mathbf{k}} \approx \hat{\mathbf{C}}_{\mathbf{k}} \mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{-} \hat{\mathbf{C}}_{\mathbf{k}}^{\mathbf{T}} + \hat{\mathbf{D}}_{\mathbf{k}} \mathbf{\Sigma}_{\tilde{\mathbf{v}},\mathbf{k}} \hat{\mathbf{D}}_{\mathbf{k}}^{\mathbf{T}} \tag{36}$$
$$\mathbf{\Sigma}_{\tilde{\mathbf{x}}\tilde{\mathbf{y}},\mathbf{k}} \approx \mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{-} \hat{\mathbf{C}}_{\mathbf{k}}^{\mathbf{T}} \tag{37}$$

Using these quantities we can calculate Kalman Gain that is shown in equation 38.

$$\mathbf{L_k} = \mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{-} \hat{\mathbf{C}}_{\mathbf{k}}^{\mathbf{T}} \left[\hat{\mathbf{C}}_{\mathbf{k}} \mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{-} \hat{\mathbf{C}}_{\mathbf{k}}^{\mathbf{T}} + \hat{\mathbf{D}}_{\mathbf{k}} \mathbf{\Sigma}_{\tilde{\mathbf{v}}} \hat{\mathbf{D}}_{\mathbf{k}}^{\mathbf{T}} \right]^{-1} \tag{38}$$

5) Step 5: State Estimate Measurement Update: In this step we update old state estimate of $\hat{\mathbf{x}}_{\mathbf{k}}^{-}$ to new state estimate of $\hat{\mathbf{x}}_{\mathbf{k}}^{+}$ using kalman gain $\mathbf{L_k}$ and innovation $\mathbf{y_k} - \hat{\mathbf{y}}_{\mathbf{k}}$.

$$\hat{\mathbf{x}}_{\mathbf{k}}^{+} = \hat{\mathbf{x}}_{\mathbf{k}}^{-} + \mathbf{L_k}(\mathbf{y_k} - \hat{\mathbf{y}}_{\mathbf{k}}) \tag{39}$$

6) Step 6: Error Covariance Measurement Update: The last step is to update the error at the state covariance.

$$\mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{+} = \mathbf{\Sigma}_{\tilde{\mathbf{x}},\mathbf{k}}^{-} + \mathbf{L_k} \mathbf{\Sigma}_{\tilde{\mathbf{y}},\mathbf{k}} \mathbf{L}_{\mathbf{k}}^{\mathbf{T}} \tag{40}$$

B. Implementing EKF to ESC cell model

To implement EKF, we must be able to calculate $\hat{\mathbf{A}}_{\mathbf{k}}$, $\hat{\mathbf{B}}_{\mathbf{k}}$, $\hat{\mathbf{C}}_{\mathbf{k}}$, and $\hat{\mathbf{D}}_{\mathbf{k}}$. In our cell model, the input that we are sensing is $\mathbf{i_k}$, but the true current passing through the system is $\mathbf{i_k} + \mathbf{w_k}$. For simplicity of the model, we will assume coulombic efficiency $\eta_k = 1$.

SOC equation, which is given in equation 3, and the two derivations we need are shown.

$$z_{k+1} = z_k - \frac{\Delta t}{Q}(i_k + w_k) \tag{41}$$
$$\left. \frac{\partial z_{k+1}}{\partial z_k} \right|_{z_k = \hat{z_k}^{+}} = 1 \tag{42}$$
$$\left. \frac{\partial z_{k+1}}{\partial w_k} \right|_{w_k = \bar{w}} = -\frac{\Delta t}{Q} \tag{43}$$

$$\tau_j = exp\left(\frac{-\Delta t}{R_j C_j}\right) \tag{44}$$

$$\mathbf{A}_{RC} = \begin{bmatrix} \tau_1 & 0 & \dots \\ 0 & \tau_2 & \\ \vdots & & \ddots \end{bmatrix} \tag{45}$$

$$\mathbf{B}_{RC} = \begin{bmatrix} (1-\tau_1) \\ (1-\tau_2) \\ \vdots \end{bmatrix} \tag{46}$$

$$\mathbf{i}_{R,k+1} = \mathbf{A}_{RC}\mathbf{i}_{R,k} + \mathbf{B}_{RC}(i_k + w_k) \tag{47}$$

Then we can calculate their derivatives.

$$\left.\frac{\partial \mathbf{i}_{R,k+1}}{\partial \mathbf{i}_{R,k}}\right|_{\mathbf{i}_{R,k}=\hat{\mathbf{i}}^+_{R,k}} = \mathbf{A}_{RC} \tag{48}$$

$$\left.\frac{\partial \mathbf{i}_{R,k+1}}{\partial w_k}\right|_{w_k=\bar{w}} = \mathbf{B}_{RC} \tag{49}$$

Now we need to consider the hysteresis state equation:

$$A_{H,k} = exp\left(-\left|\frac{(i_k + w_k)\gamma\Delta t}{Q}\right|\right) \tag{50}$$

$$h_{k+1} = A_{H,k}h_k + (1 - A_{H,k})sgn(i_k + w_k) \tag{51}$$

Then we can calculate its derivatives.

$$\left.\frac{\partial h_{k+1}}{\partial h_k}\right|_{\substack{h_k=\hat{h}^+_k \\ w_k=\bar{w}}} = \exp\left(-\left|\frac{(i_k + \bar{w}_k)\gamma\Delta t}{Q}\right|\right) = \bar{A}_{H,k} \tag{52}$$

$$\left.\frac{\partial h_{k+1}}{\partial w_k}\right|_{\substack{h_k=\hat{h}^+_k \\ w_k=\bar{w}}} = -\left|\frac{\gamma\Delta t}{Q}\right| \bar{A}_{H,k}\left(1 + \text{sgn}(i_k + \bar{w}_k)\hat{h}^+_k\right) \tag{53}$$

Now we need to consider the zero-state hysteresis state equation:

$$s_{k+1} = \begin{cases} sgn(i_k + w_k), & |i_k + w_k| > 0 \\ s_k, & \text{otherwise} \end{cases} \tag{54}$$

Then we can calculate its derivatives. We will consider $i_k + w_k = 0$ as not possible.

$$\frac{\partial s_{k+1}}{\partial s_k} = 0 \tag{55}$$

$$\frac{\partial s_{k+1}}{\partial w_k} = 0 \tag{56}$$

Now we can look at parameters that determines $\hat{\mathbf{C}}_\mathbf{k}$ and $\hat{\mathbf{D}}_\mathbf{k}$. We will first look at output of our system which is sensed terminal voltage of cell.

$$y_k = OCV(z_k) + Mh_k + M_0 s_k - \sum_j R_j i_{R_j,k} - R_0 i_k + v_k \tag{57}$$

Using Equation 57 we derive their derivative.

$$\left.\frac{\partial y_k}{\partial s_k}\right| = M_0 \tag{58}$$

$$\left.\frac{\partial y_k}{\partial h_k}\right| = M \tag{59}$$

$$\left.\frac{\partial y_k}{\partial i_{R_j,k}}\right| = -R_j \tag{60}$$

$$\left.\frac{\partial y_k}{\partial v_k}\right| = 1 \tag{61}$$

$$\left.\frac{\partial y_k}{\partial z_k}\right|_{z_k=\hat{z}^-_k} = \left.\frac{\partial OCV(z_k)}{\partial z_k}\right|_{z_k=\hat{z}^-_k} \tag{62}$$

IV. Implementation and Results

In this section, we will examine the results of EKF performance in different scenarios.

A. Constant Discharging Test

In this test, a constant but low current is applied to the cell, allowing the OCV curve to be clearly observed. As expected, the SOC decreases steadily over time. The Extended Kalman Filter (EKF) successfully tracks the SOC, with a root mean square (RMS) estimation error of 0.4432%. The error exceeds the estimated bounds only 0.01035% of the time.

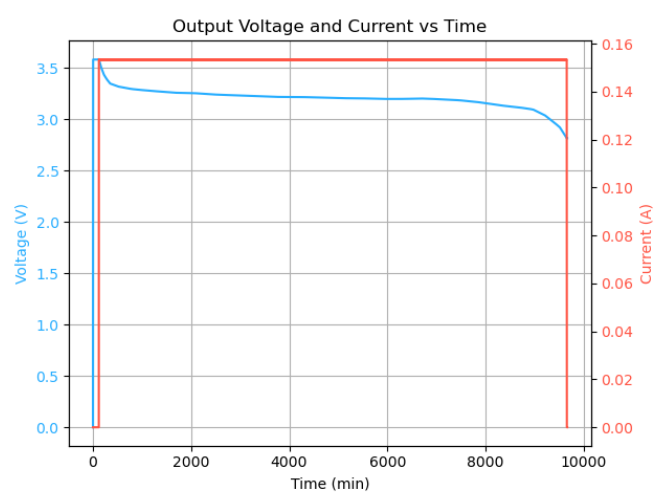

Fig. 3. Input Current and output voltage versus time in test 1

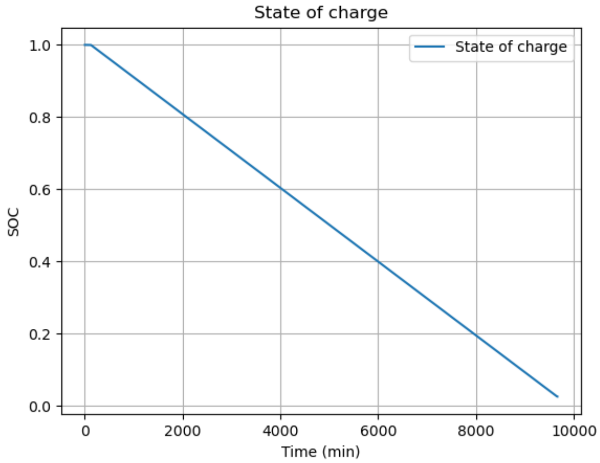

Fig. 4. SOC versus time in test 1

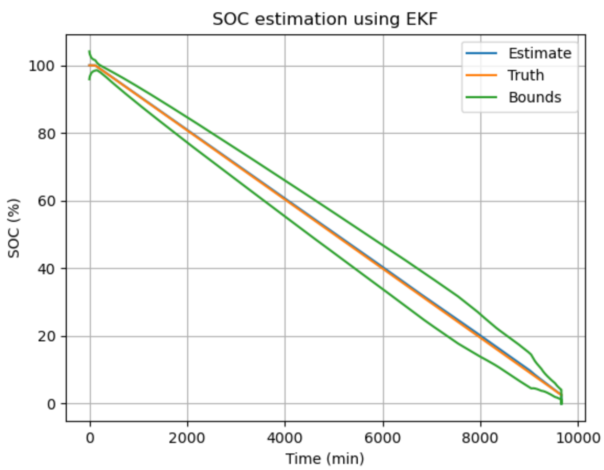

Fig. 5. Estimation of SOC in test 1

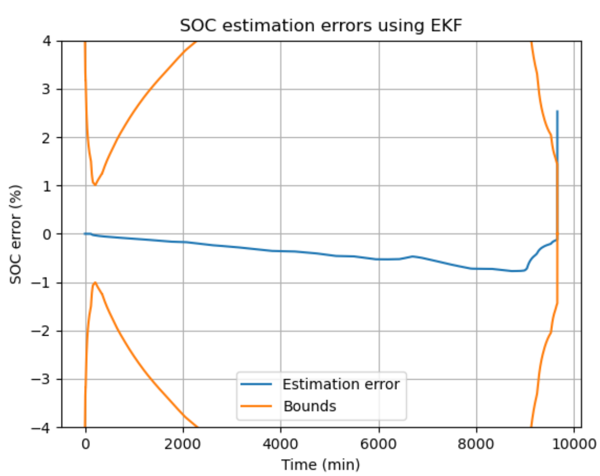

Fig. 6. Bounds and Error at estimation in test 1

B. Terrain Road Environment Test

In this test, we simulate a car driving on the road, where the cell frequently switches between charging and discharging. As a result, the output voltage oscillates and shows a slight upward trend. The SOC initially drops to around 0.1 before partially recovering due to intermittent charging. The EKF effectively tracks the SOC, achieving a root mean square (RMS) estimation error of 0.1028%. The estimation error exceeds the confidence bounds only 0.007126% of the time.

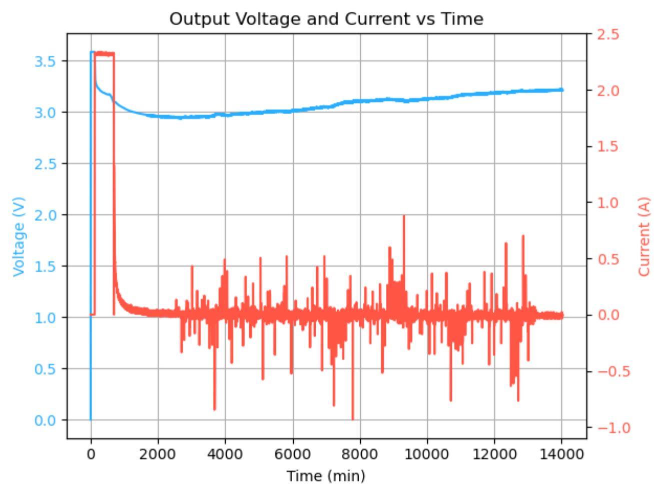

Fig. 7. Input Current and output voltage versus time in test 2

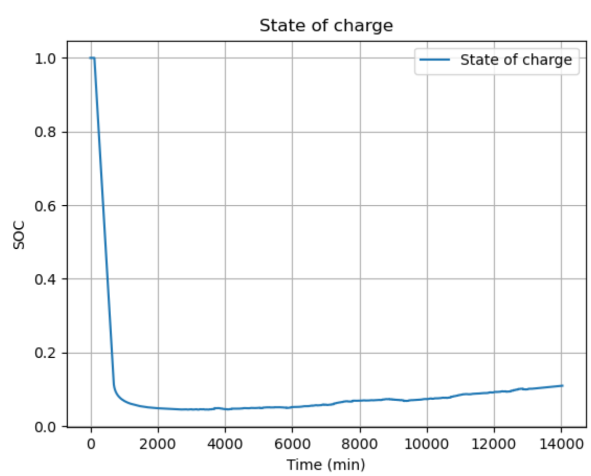

Fig. 8. SOC versus time in test 2

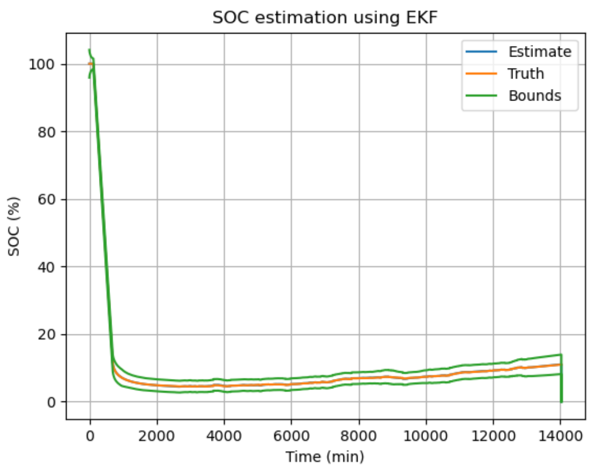

Fig. 9. Estimation of SOC in test 2

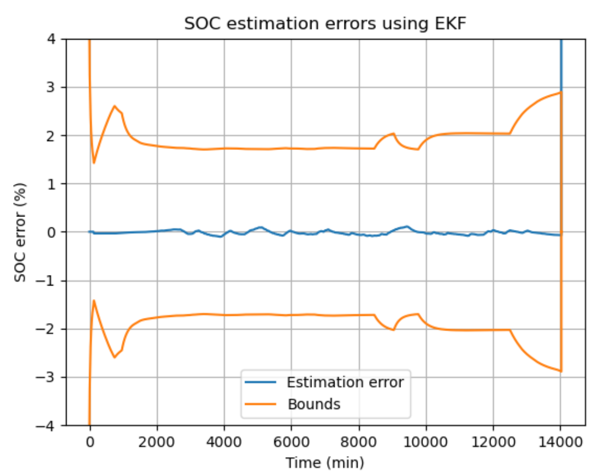

Fig. 10. Bounds and Error at estimation in test 2

C. Constant Charging Test

In this test, the cell is simulated under constant current charging. The EKF tracks the SOC with a root mean square (RMS) estimation error of 1.098%. Percent of time error outside bounds = 0.01057%

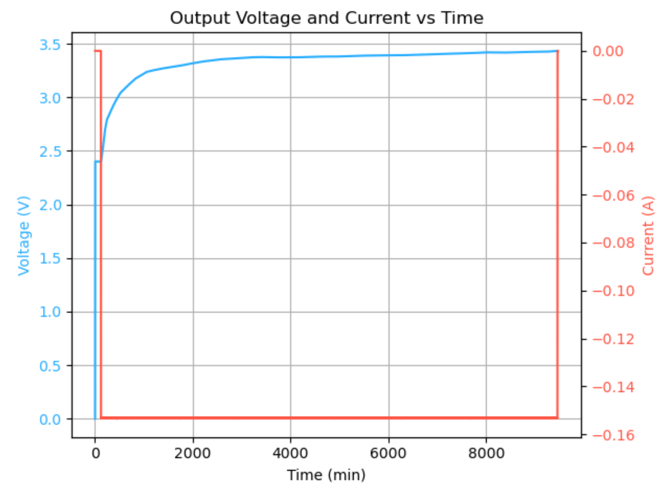

Fig. 11. Input Current and output voltage versus time in test 3

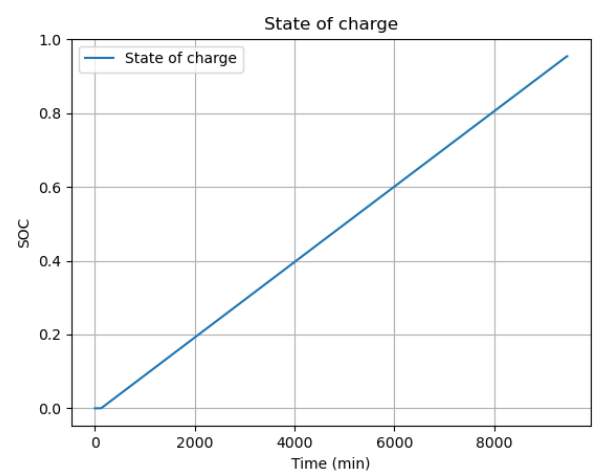

Fig. 12. SOC versus time in test 3

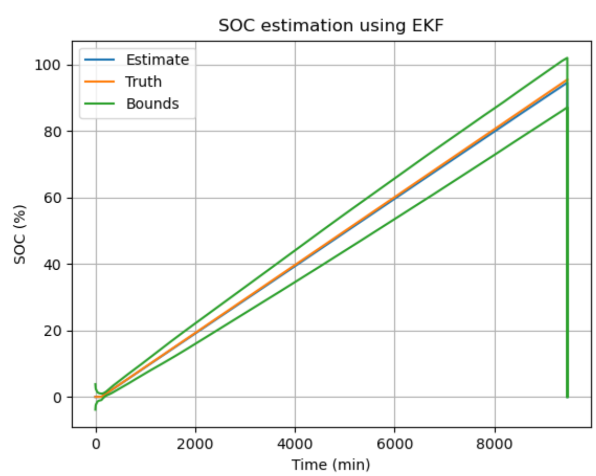

Fig. 13. Estimation of SOC in test 3

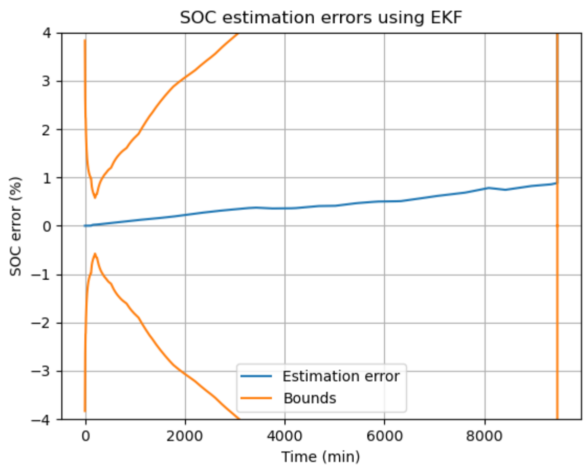

Fig. 14. Bounds and Error at estimation in test 3

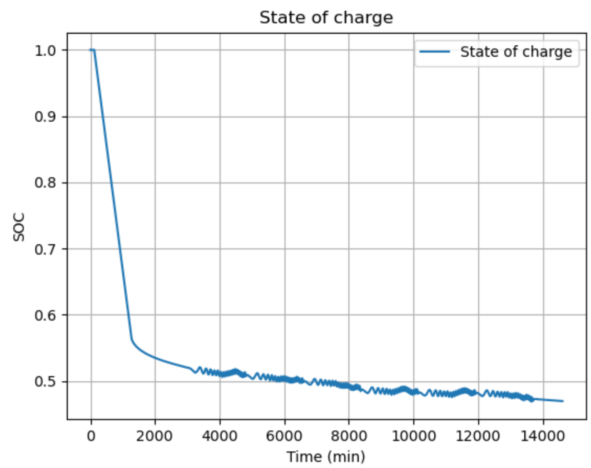

Fig. 16. SOC versus time in test 4

D. Oscilating Current Test

In this test, we evaluate the EKF's performance under oscillating current conditions. The root mean square (RMS) SOC estimation error is 0.4758%, and the error exceeds the confidence bounds only 0.006845% of the time.

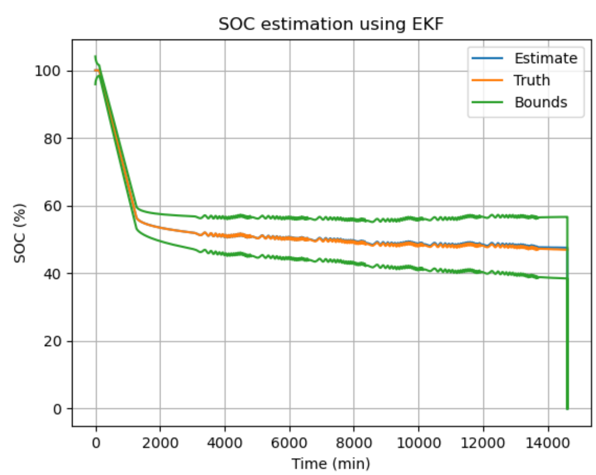

Fig. 17. Estimation of SOC in test 4

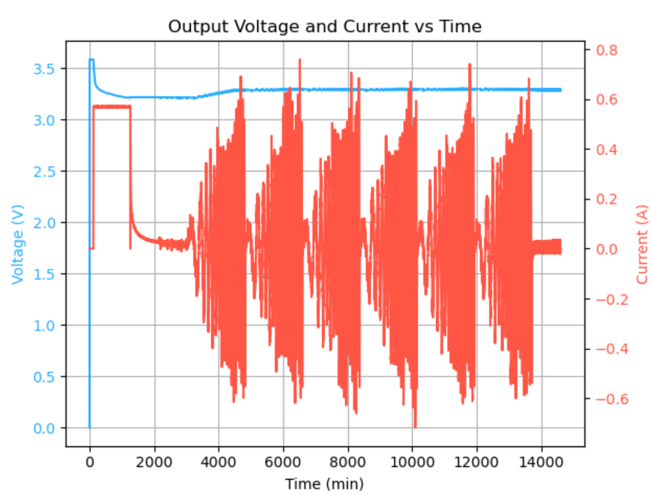

Fig. 15. Input Current and output voltage versus time in test 4

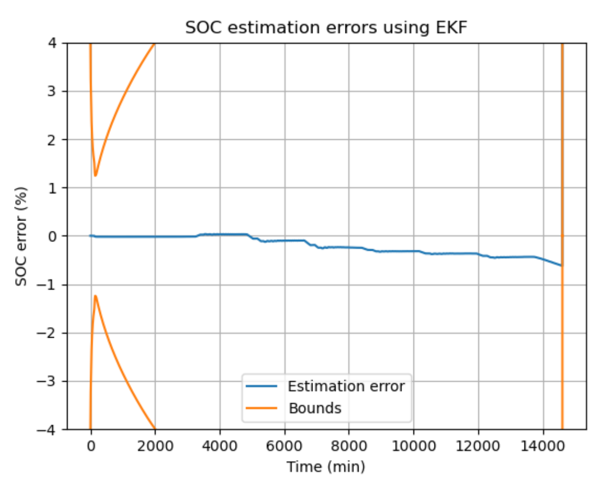

Fig. 18. Bounds and Error at estimation in test 4

E. Constant High Current in Constant Intervals Test

In this section, a high constant current is applied to the cell at regular intervals. The EKF achieves a root mean square (RMS) SOC estimation error of 0.3622%, with the error exceeding the confidence bounds 0.2088% of the time.

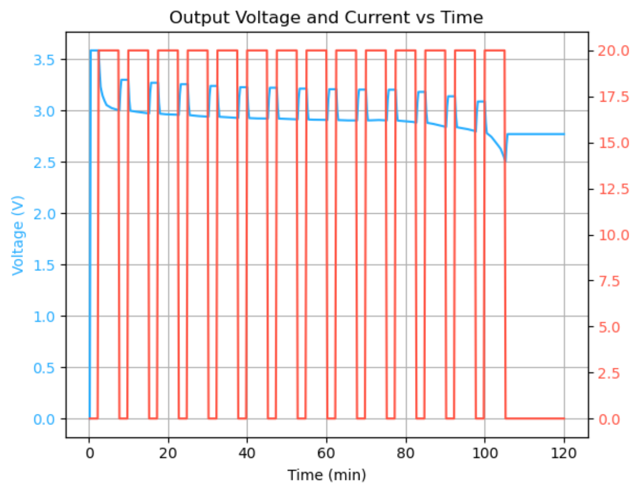

Fig. 19. Input Current and output voltage versus time in test 5

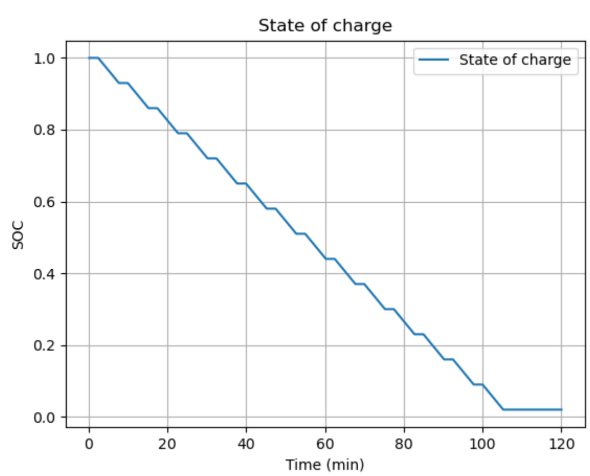

Fig. 20. SOC versus time in test 5

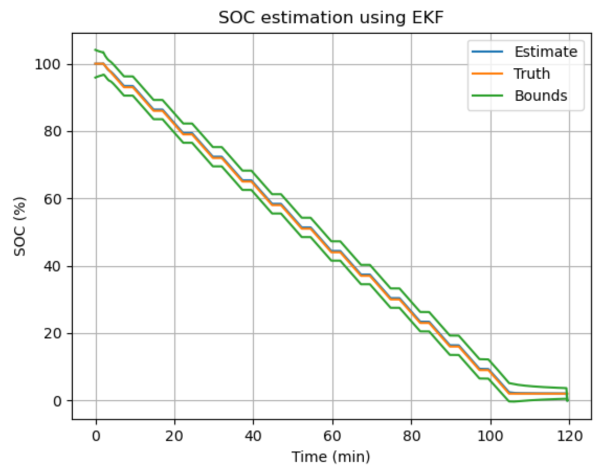

Fig. 21. Estimation of SOC in test 5

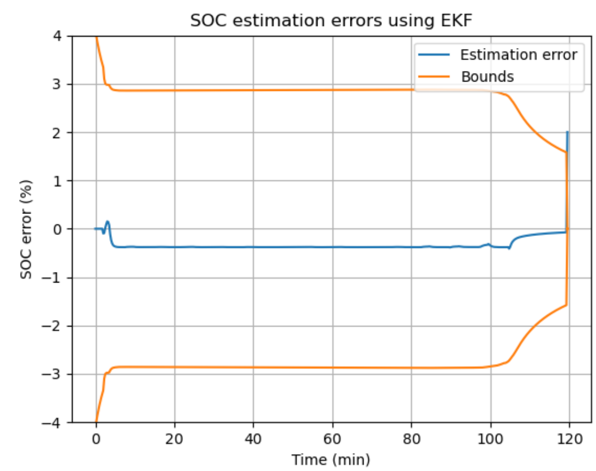

Fig. 22. Bounds and Error at estimation in test 5

Appendix

A. OCVmodel.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Read input file
input_file = pd.read_csv('model_input/input_5.csv')
t = np.array(input_file['time'].dropna())      # time array, shape (N,)
i = np.array(input_file['current'].dropna())  # current array, shape (N,)
init_soc = input_file['initsoc'][0]           # initial SOC, scalar value

# Set simulation length based on current array
dt = t[1] - t[0]                  # assume uniform sampling
sample_freq = 1 / dt
sim_steps = len(i)                # number of simulation steps

# Truncate t to match current
t = t[:sim_steps]

idx_T = 0  # [-25 -15 -5 5 15 25 45] degrees Celsius, index for temperature

# read your cell model once
cell_model = pd.read_csv('model_param/cell_model.csv')

# Fields pertaining to the OCV versus SOC relationship:
OCV   = cell_model['OCV'].dropna()         # OCV vector at which SOC0 and SOCrel are stored
OCV0  = cell_model['OCV0'].dropna()        # Vector of OCV versus SOC at 0 degree Celsius
OCVrel= cell_model['OCVrel'].dropna()      # Vector of change in OCV versus SOC per degree Celsius [V/C]
SOC   = cell_model['SOC'].dropna()         # SOC vector at which OCV0 and OCVrel are stored
SOC0  = cell_model['SOC0'].dropna()        # Vector OF SOC versus OCV at 0 degree Celsius
SOCrel= cell_model['SOCrel'].dropna()      # Vector of change in SOC versus OCV per degree Celsius [1/C]

# Fields pertaining to the dynamic relationship:
temps   = cell_model['temps'].dropna()     # Temperatures at which dynamic parameters are stored [C]
QParam  = cell_model['QParam'].dropna()    # Capacity Q at each temperature [Ah]
etaParam= cell_model['etaParam'].dropna() # Coulombic efficiency eta at each temperature [unitless]
GParam  = cell_model['GParam'].dropna()    # Hysteresis "gamma" parameter [unitless]
MParam  = cell_model['MParam'].dropna()    # Hysteresis M parameter [V]
M0Param = cell_model['M0Param'].dropna()   # Hysteresis M0 parameter [V]
R0Param = cell_model['R0Param'].dropna()   # Series resistance parameter R_0 [ohm]
RCParam = cell_model['RCParam'].dropna()   # The R-C time constant parameter R_j C_j [s]
RParam  = cell_model['RParam'].dropna()    # Resistance R_j of R-C parameter [ohm]


def get_OCV0(z):
  OCV0_curr = np.interp(z, SOC0, OCV0)
  return OCV0_curr
def get_OCVrel(z):
  OCVrel_curr = np.interp(z, SOCrel, OCVrel)
  return OCVrel_curr


# Initialize arrays to match input data length
z = np.ones(sim_steps)  # initialize z
z[0] = init_soc  # set initial SOC
i_R = np.zeros(sim_steps)  # initialize i_R
h = np.zeros(sim_steps)  # initialize h
v = np.zeros(sim_steps)  # initialize v

s = 0  # Initialize s

# Simulation loop
for time in range(sim_steps - 1):
    # calculating next state
    A_RC = np.exp(-dt / RCParam[idx_T])
    B_RC = 1 - A_RC
    A_H = np.exp(-np.abs((etaParam[idx_T] * i[time] * GParam[idx_T] * dt) / (QParam[idx_T]*3600)))
    z[time + 1] = z[time] + (-etaParam[idx_T] * dt / (QParam[idx_T]*3600)) * i[time]
    i_R[time + 1] = A_RC * i_R[time] + B_RC * i[time]
    h[time + 1] = A_H * h[time] + (A_H - 1) * np.sign(i[time])

```

```python
    # calculating current output
    if abs(i[time]) > 0:
        s = np.sign(i[time])
    OCV = get_OCV0(z[time]) + get_OCVrel(z[time])*0.001*temps[idx_T]
    v[time + 1] = OCV + M0Param[idx_T]*s + MParam[idx_T]*h[time] - RParam[idx_T]*i_R[time] - R0Param[idx_T]*i[time]

# Debug print statements
print(f"Last SOC (z): {z[-1]:.2f}")

data = pd.DataFrame({
    'time': t,
    'current': i,
    'voltage': v,
    'soc': z,
})
data.to_csv('model_out/sim_data.csv', index=False) # This saves the simulated time, true SOC (z), voltage (v), and current (i) for use by the EKF script.

mins = t / 60  # Convert time to minutes for plotting
# Plotting
fig, ax1 = plt.subplots()

# Plot voltage on left y-axis
ax1.plot(mins, v, label="Output Voltage", color=[0.1529, 0.6824, 1])
ax1.set_xlabel("Time (min)")
ax1.set_ylabel("Voltage (V)", color=[0.1529, 0.6824, 1])
ax1.tick_params(axis='y', labelcolor=[0.1529, 0.6824, 1])
ax1.grid(True)
# Create a second y-axis that shares the same x-axis
ax2 = ax1.twinx()
ax2.plot(mins, i, label="Current", color=[1, 0.3333, 0.2706])
ax2.set_ylabel("Current (A)", color=[1, 0.3333, 0.2706])
ax2.tick_params(axis='y', labelcolor=[1, 0.3333, 0.2706])
# Title and show
plt.title("Output Voltage and Current vs Time")
fig.tight_layout()
plt.show()


plt.plot(mins, z, label="State of charge")
plt.title("State of charge")
plt.xlabel("Time (min)")
plt.ylabel("SOC")
plt.grid(True)
plt.legend()
plt.show()
```

B. main.py

```python
from EKF import EKF
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


cell_model = pd.read_csv('model_param/cell_model.csv')
Cell_DYN_P5 = pd.read_csv('model_out/sim_data.csv')
idx_T = 0  # [-25 -15 -5 5 15 25 45]
T = -25  # degrees Celsius

time = Cell_DYN_P5['time'].values[1:]       # Time
time = time - time[0]                       # Normalize time to start from 0
deltat = time[1] - time[0]                  # Sample interval
current = Cell_DYN_P5['current'].values[1:] # Current , discharge> 0;charge <0
voltage = Cell_DYN_P5['voltage'].values[1:] # Voltage
soc = Cell_DYN_P5['soc'].values[1:]         # True SOC

# Reserve storage for computed results, for plotting
sochat = np.zeros_like(soc)
socbound = np.zeros_like(soc)

# Covariance values
SigmaX0 = np.diag([1e-6, 1e-8, 2e-4])   # uncertainty in initial state
SigmaW = 2e-1   # uncertainty in current sensor, state equation
SigmaV = 2e-1   # uncertainty in voltage sensor, output equation

# Create ekfData structure and initialize variables using first voltage measurement and first temperature measurement
EKF_model = EKF( voltage[0] , idx_T , SigmaX0, SigmaV, SigmaW , cell_model )

# Now enter a loop for remainder of time, where we update the EKF once per sample interval

for k in range(voltage.shape[0] - 1):

    vk = voltage[k] # 'measure' voltage
    ik = current[k] # 'measure' current
    Tk = T          # 'measure' temperature

    # Update SOC (and model state)
    sochat[k],socbound[k] = EKF_model.iterEKF(vk,ik,Tk,deltat)
    print(f"Iteration {k+1}/{voltage.shape[0]-1}: SOC = {sochat[k]:.4f}, Bound = {socbound[k]:.4f}")

# Plot results

# Plot 1: SOC estimation
plt.figure(1)
plt.clf()
plt.plot(time / 60, 100 * sochat, label='Estimate')
plt.plot(time / 60, 100 * soc, label='Truth')

# Plot bounds
plt.plot(np.concatenate((time / 60, [np.nan], time / 60)),
         np.concatenate((100 * (sochat + socbound), [np.nan], 100 * (sochat - socbound))),
         label='Bounds')

plt.title('SOC estimation using EKF')
plt.xlabel('Time (min)')
plt.ylabel('SOC (%)')
plt.legend()
plt.grid(True)

# Print RMS error
rms_error = np.sqrt(np.mean((100 * (soc - sochat))**2))
print(f'RMS SOC estimation error = {rms_error:.4g}%')

# Plot 2: SOC estimation error
plt.figure(2)
plt.clf()
plt.plot(time / 60, 100 * (soc - sochat), label='Estimation error')

# Plot error bounds
```

```python
plt.plot(np.concatenate((time / 60, [np.nan], time / 60)),
         np.concatenate((100 * socbound * np.ones_like(sochat), [np.nan], -100 * socbound * np.ones_like(sochat))),
         label='Bounds')

plt.title('SOC estimation errors using EKF')
plt.xlabel('Time (min)')
plt.ylabel('SOC error (%)')
plt.ylim([-4, 4])
plt.legend()
plt.grid(True)

# Compute percentage of time error outside bounds
ind = np.where(np.abs(soc - sochat) > socbound)[0]
percent_outside = len(ind) / len(soc) * 100
print(f'Percent of time error outside bounds = {percent_outside:.4g}%')

plt.show()
```

C. EKF.py

```
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

class EKF:
    def __init__(self, v0, idx_T0, SigmaX0, SigmaV, SigmaW, model):

        # Store model
        self.model = model

        # Extract only valid pairs for OCV0/SOC0 and OCVrel/SOCrel
        mask0 = (~self.model['SOC0'].isna()) & (~self.model['OCV0'].isna())
        self.SOC0 = np.array(self.model.loc[mask0, 'SOC0'])
        self.OCV0 = np.array(self.model.loc[mask0, 'OCV0'])

        maskrel = (~self.model['SOCrel'].isna()) & (~self.model['OCVrel'].isna())
        self.SOCrel = np.array(self.model.loc[maskrel, 'SOCrel'])
        self.OCVrel = np.array(self.model.loc[maskrel, 'OCVrel'])

        self.temps = np.array(self.model['temps'].dropna())
        self.idx_T0 = idx_T0
        T0 = self.temps[idx_T0]

        # Initialize state description
        ir0 = 0.0  # Initial diffusion current
        hk0 = 0.0  # Initial hysteresis voltage
        SOC0 = self.SOCfromOCVtemp(v0, T0)

        """
        ### Debugging
        print(f"\nInitial voltage: {v0}, Initial SOC from OCV: {SOC0}")
        soc_range = np.linspace(0, 1, 100)
        ocv_curve = [self.OCVfromSOCtemp(z, T0) for z in soc_range]
        plt.plot(soc_range, ocv_curve)
        plt.xlabel('SOC')
        plt.ylabel('OCV')
        plt.title('OCV vs SOC')
        plt.show()
        """

        # State variable indices
        self.irInd = 0
        self.hkInd = 1
        self.zkInd = 2

        # Initial state (column matrix)
        self.xhat = np.array([[ir0] , [hk0] , [SOC0]])

        # Covariances - ensure they are positive definite
        self.SigmaW = np.abs(SigmaW)  # Ensure positive
        self.SigmaV = np.abs(SigmaV)  # Ensure positive
        self.SigmaX = np.diag(np.diag(SigmaX0))  # Use only diagonal elements initially
        self.SXbump = 5

        # Previous current value
        self.priorI = 0.0
        self.signIK = 0.0

        # Add minimum covariance values to prevent numerical issues
        self.min_cov = 1e-6

    def OCVfromSOCtemp(self, z, T):
        """
        Get temperature-compensated OCV from SOC (z) and T ( C )
        """
        # Interpolate OCV0 at given SOC
        ocv0 = np.interp(z, self.SOC0, self.OCV0)

        # Interpolate OCVrel at given SOC
        ocvrel = np.interp(z, self.SOCrel, self.OCVrel)

```

```python
        # If OCV0 is at 0 C , use T directly. If at 25 C , use (T-25)
        ocv = ocv0 + ocvrel * 0.001 * T

        return ocv

    def SOCfromOCVtemp(self, v, T):
        """
        Estimate SOC from OCV and temperature using interpolation
        """
        # Create a function to find SOC that gives the target OCV
        def find_soc(soc_guess):
            return self.OCVfromSOCtemp(soc_guess, T) - v

        # Try different initial guesses
        initial_guesses = [0.2, 0.5, 0.8]
        best_soc = None
        min_error = float('inf')

        for guess in initial_guesses:
            try:
                soc = fsolve(find_soc, guess, full_output=True)
                if soc[1]['fvec'][0] < min_error:
                    min_error = soc[1]['fvec'][0]
                    best_soc = soc[0][0]
            except:
                continue

        if best_soc is None:
            # Fallback to linear interpolation if root finding fails
            # Find the closest OCV values in our lookup table
            ocv0_values = self.OCV0
            soc0_values = self.SOC0
            best_soc = np.interp(v, ocv0_values, soc0_values)

        # Ensure SOC is within valid range [0, 1]
        soc = np.clip(best_soc, 0, 1)

        return soc

    def dOCVfromSOCtemp(self, z, T):
        """
        Compute dOCV/dSOC at a given SOC z and temperature T
        """
        # Small perturbation for numerical derivative
        delta = 1e-6

        # Calculate OCV at slightly higher and lower SOC
        ocv_plus = self.OCVfromSOCtemp(z + delta, T)
        ocv_minus = self.OCVfromSOCtemp(z - delta, T)

        # Compute numerical derivative
        dOCV = (ocv_plus - ocv_minus) / (2 * delta)

        return dOCV


    def iterEKF(self, vk, ik, Tk, deltat):

        # Load the cell model parameters for the present operating temp
        Q = self.model['QParam'][self.idx_T0]
        G = self.model['GParam'][self.idx_T0]
        M = self.model['MParam'][self.idx_T0]
        M0 = self.model['M0Param'][self.idx_T0]
        RC = self.model['RCParam'][self.idx_T0]
        R = self.model['RParam'][self.idx_T0]
        R0 = self.model['R0Param'][self.idx_T0]
        eta = self.model['etaParam'][self.idx_T0]

        if ik<0: ik=ik*eta # adjust current if charging cell

        # Get data stored in data structure
        SigmaX = self.SigmaX
        SigmaW = self.SigmaW
        SigmaV = self.SigmaV
```

```python
        irInd = self.irInd
        hkInd = self.hkInd
        zkInd = self.zkInd
        xhat = self.xhat
        nx = xhat.shape[0]
        I = self.priorI
        if abs(ik) > Q/100: self.signIK = np.sign(ik)  # Update sign of current if large enough
        signIK = self.signIK

        # EKF Step 1: State prediction time update
        # First compute Ahat[k-1] and Bhat[k-1]
        Ah = np.exp(-np.abs((I * G * deltat) / (3600 * Q)))
        Bh = - np.abs(G * deltat / (3600 * Q)) * Ah * (1 + np.sign(I) * float(xhat[hkInd, 0]))

        Ahat = np.zeros((nx, nx))
        Bhat = np.zeros((nx, 1))

        Ahat[zkInd, zkInd] = 1
        Bhat[zkInd] = - deltat / (3600 * Q)

        Ahat[irInd, irInd] = np.diag([RC])
        Bhat[irInd] = [1 - RC]

        B = np.hstack([Bhat, np.zeros_like(Bhat)])

        Ahat[hkInd, hkInd] = Ah
        Bhat[hkInd, 0] = Bh

        B[hkInd, 1] = Ah - 1

        # Next update xhat
        xhat = Ahat @ xhat + B @ np.array([[I], [np.sign(I)]])

        # EKF Step 2: Error covariance prediction time update
        SigmaX = Ahat @ SigmaX @ Ahat.T + Bhat @ np.atleast_2d(SigmaW) @ Bhat.T

        # EKF Step 3: Output estimate
        yhat = self.OCVfromSOCtemp(xhat[zkInd, 0], Tk) + M0 * signIK + M * xhat[hkInd, 0] - R * xhat[irInd,
             0] - R0 * ik

        # EKF Step 4: Estimator gain matrix
        Chat = np.zeros((1, nx))
        Chat[0, zkInd] = self.dOCVfromSOCtemp(float(xhat[zkInd, 0]), Tk)
        Chat[0, hkInd] = M
        Chat[0, irInd] = -R
        Dhat = np.array([[1.0]])

        # Ensure numerical stability in covariance calculations
        SigmaY = Chat @ SigmaX @ Chat.T + Dhat * SigmaV

        L = SigmaX @ Chat.T @ np.linalg.inv(SigmaY)

        """
        try:
            L = SigmaX @ Chat.T @ np.linalg.inv(SigmaY)
        except np.linalg.LinAlgError:
            # If matrix inversion fails, use a more stable approach
            L = SigmaX @ Chat.T / SigmaY
        """

        # EKF Step 5: State estimate measurement update
        r = vk - yhat  # residual

        # Adaptive measurement update
        r_scalar = float(np.atleast_1d(r).squeeze())
        SigmaY_scalar = float(np.atleast_1d(SigmaY).squeeze())

        if r_scalar**2 > 100 * SigmaY_scalar:
            L = np.zeros_like(L)  # Zero gain if residual is too large

        xhat = xhat + L * r
        xhat[hkInd, 0] = np.minimum(1, np.maximum(-1, xhat[hkInd, 0]))
        xhat[zkInd, 0] = np.minimum(1.05, np.maximum(-0.05, xhat[zkInd, 0]))

```

```
        # EKF Step 6: Error covariance measurement update
        SigmaX = SigmaX - L @ SigmaY @ L.T

        # More gradual adaptation based on residual
        if r_scalar**2 > 4 * SigmaY_scalar:
            print("Bumping SigmaX")
            SigmaX[zkInd, zkInd] = SigmaX[zkInd, zkInd] * self.SXbump

        # Force symmetry and positive definiteness
        SigmaX = (SigmaX + SigmaX.T) / 2
        eigvals = np.linalg.eigvals(SigmaX)
        if np.any(eigvals < 0):
            SigmaX = SigmaX + np.eye(nx) * self.min_cov

        """
        U, S, V = np.linalg.svd(SigmaX)
        HH = V @ S @ V.T
        SigmaX = (SigmaX + SigmaX.T + HH + HH.T) / 4
        np.fill_diagonal(SigmaX, np.maximum(np.diag(SigmaX), self.min_cov))
        """

        # Save data in EKF structure for next time
        self.priorI = ik
        self.SigmaX = SigmaX
        self.xhat = xhat
        zk = float(xhat[zkInd, 0])
        zkbnd = float(3 * np.sqrt(max(SigmaX[zkInd, zkInd], self.min_cov)))

        return zk, zkbnd
```