Battery State of Charge estimation with Kalman filter

Pierre Haessig

▶ **To cite this version:**

Pierre Haessig. Battery State of Charge estimation with Kalman filter. CentraleSupélec; IETR UMR 6164. 2024. hal-04701587v2

HAL Id: hal-04701587
https://hal.science/hal-04701587v2

Submitted on 18 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Battery State of Charge estimation with Kalman filter

This notebook explores the State of Charge (SoC) estimation of a battery using a state observer algorithm, the Kalman filter, or more precisely its nonlinear extension: the extended Kalman filter (EKF).

The battery model is based on a resistive voltage drop in series with a open circuit voltage which depends on the SoC (see below). The SoC estimation uses both the current and voltage measurement (*but no OCV measurement*) along with a previous SoC estimate.

The notebook provides three Python implementations of the Kalman filter:

1. a step-by-step literate programming version of the filter, using a sequence of notebook cells, to implement one step of the filter
2. a generic implementation (all the above steps wrapped in a single function)
3. a compact implementation specialized for SoC estimation with baked-in battery model

References are provided at the end. More explanations about the Kalman filter are at the start of the dedicated section.

🔋 Pierre Haessig, September 2024, CC-BY 🪫

(cc) BY (code fragments may be reused under the MIT license)

In [1]:
```
import numpy as np
from numpy import sqrt
from numpy.linalg import inv # matrix inversion
```

Shortcuts to create vectors and matrices of dimension 1 or (1,1)

In [2]:
```
v1 = lambda x: np.array([x])
m11 = lambda x: np.array([[x]])
I1 = m11(1.0) # Identity matrix of dim (1,1)
I1
```

Out[2]:
```
array([[1.]])
```

State space model of the battery

State space obsevers like the Kalman filter use a so-called state space model where the system is described with two functions:

- **state dynamics equation**: $f : (x_{k-1}, u_k) \mapsto x_k$
- **output equation**: $g : (x_k, u_k) \mapsto y_k$

where x, u and y are respectively the state, input and output vectors.

Battery state dynamics equation

- model state is the battery state of charge SoC (state vector x of dimension 1)
- model input is the current i (input vector u of dimension 1)

State equation is linear:

$$SoC_k = SoC_{k-1} + \frac{\Delta t}{Q_{rated}} i_k$$

where Q_{rated} is the rated cell capacity in Ah (if Δt in hours). This is the Euler backward integration of $dSoC/dt = i(t)/Q_{rated}$ over a time step Δt.

In [3]:
```
def state_dyn_batt(x, u, w=v1(0.), **params):
    """Battery state dynamics equation:
    x(k) = f(x(k-1), u(k), w(k))

    with:
    - x = [SoC], *previous* state of charge in [0,1]
    - u = [i], current in A (>0 when charging)
    - w = state evolution noise (optional, default to 0)

    params dict content:
    - Δt : time step (e.g. in hours)
    - Qrated: cell rated capacity, in Ah (if Δt in hours)
    """
    Δt = params['Δt']
    Qrated = params['Qrated']
    SoC = x[0]
    i = u[0]
    SoC_next = SoC + i/Qrated*Δt
    x_next = v1(SoC_next) + w
    return x_next

params_dyn = dict(Qrated=10, Δt=0.5)
SoC = 0.5
i = 4 # A → i*Δt = 2Ah = 20% SoC increase
x = v1(SoC)
u = v1(i)
state_dyn_batt(x, u, **params_dyn)
```

Out[3]:
```
array([0.7])
```

Jacobian, needed because state dynamics may be nonlinear. Since in the above example it is in fact linear, the Jacobian is in fact a constant 1.0

In [4]:
```
def state_jac_batt(x, u, **params):
    """Jacobian, with respect to state, of battery state dynamics equation:
    F = ∂f/∂x at (x,u)
      = ∂(SoC(k))/∂(SoC(k-1))
    """
    F = m11(1.0) # ∂(SoC + i/Qrated*Δt)/∂SoC = 1.0
    return F

state_jac_batt(x, u, **params_dyn)
```

Out[4]:
```
array([[1.]])
```

Battery output equation

Model output is the cell voltage $V = V_{oc}(SoC) + R.i$.

- If the $V_{oc}(SoC)$ hasn't a constant slope, this is a nonlinear output equation.

Open circuit cell voltage model

see e.g. OCV curve of Fig 1.a in (QQ Yu et al., 2018)

In [5]:
```python
def Voc(SoC, **params):
    """Open circuit cell voltage (OCV), function of SoC
    linear model:
    - Voc_empty at 0% SoC
    - Voc_full at 100% SoC
    with Voc_empty and Voc_full taken from the params dict
    """
    Voc_empty = params['Voc_empty']
    Voc_full = params['Voc_full']
    V = Voc_empty*(1-SoC) + Voc_full*SoC
    return V

def Voc_jac(SoC, **params):
    """Jacobian (sensitivity) of OCV with respect to SoC
    ∂Voc/∂SoC
    """
    Voc_empty = params['Voc_empty']
    Voc_full = params['Voc_full']
    return (Voc_full - Voc_empty)/1.0

params_Voc = dict(Voc_full=4.2, Voc_empty=3.6)
SoC = 0.5
Voc(SoC, **params_Voc), Voc_jac(SoC, **params_Voc)
```

Out[5]: (3.9000000000000004, 0.6000000000000001)

Output equation

In [6]:
```python
def output_batt(x, u, v=v1(0.), **params):
    """Battery output equation
    y(k) = g(x(k), u(k), v(k))

    with:
    - x = [SoC], *present* state of charge in [0,1]
    - u = [i], current in A (>0 when charging)
    - v = output measurement noise (optional, default to 0)

    params dict content:
    - Voc : Voc function of SoC (in V)
    - R: cell series resistance (in Ω)
    - any param needed by Voc
    """
    Voc = params['Voc']
    R = params['R']
    SoC = x[0]
    i = u[0]
    vbatt = Voc(SoC, **params) + R*i
    y = v1(vbatt) + v
    return y

params_out = dict(Voc=Voc, Voc_jac=Voc_jac, R=0.025) | params_Voc
SoC = 0.5 # → 3.9 V OCV
i = 8. # A → R*i = 0.8 V
x = v1(SoC)
u = v1(i)
output_batt(x, u, **params_out) # 4.1 V
```

Out[6]: array([4.1])

Jacobian, needed because output equation may be nonlinear. Since in the above example the open

circuit voltage is in fact linear (affine), the output Jacobian is a constant $V_{full} - V_{empty}$

In [7]:
```
def output_jac_batt(x, u, **params):
    """Jacobian, with respect to state, of battery output equation:
    H = ∂g/∂x at (x,u)
      = ∂(V(k))/∂(SoC(k))
    """
    Voc_jac = params['Voc_jac']
    SoC = x[0]
    H = m11(Voc_jac(SoC, **params))
    return H
output_jac_batt(x, u, **params_out) # V/SoC
```

Out[7]:
```
array([[0.6]])
```

grouping all model parameters in one dict

In [8]:
```
params = params_out | params_dyn
params
```

Out[8]:
```
{'Voc': <function __main__.Voc(SoC, **params)>,
 'Voc_jac': <function __main__.Voc_jac(SoC, **params)>,
 'R': 0.025,
 'Voc_full': 4.2,
 'Voc_empty': 3.6,
 'Qrated': 10,
 'Δt': 0.5}
```

State observer based on Kalman filter

The goal of a state observer is to estimate the present state $\hat{x}_k$ based on:

- previous state estimate $\hat{x}_{k-1}$
- present input u_k
- present output measurement y_k

Observer mapping: $(x_{k-1}, u_k, y_k) \mapsto \hat{x}_k$

It is a recursive computation (i.e. a signal filter) in the sense that it is iteratively applied at each sampling instant, using the latest state estimate $\hat{x}_k$ to estimate the next one $\hat{x}_{k+1}$...

The state observer needs the following model data:

- the state dynamics equation $f : (x_{k-1}, u_k) \mapsto x_k$, with its Jacobian with respect to the state
- the output equation $g : (x_k, u_k) \mapsto y_k$, with its Jacobian with respect to the state

For the particuliar case of the Kalman filter, extra inputs are needed, namely covariance matrices which provide estimates of the uncertainty about 3 variables:

- Cov_x_{k-1}: covariance of previous state estimate $\hat{x}_{k-1}$
- Cov_w_k: covariance of state evolution noise w_k
- Cov_y_k: covariance of present output measurement y_k (output noise is often denoted v_k)
- remark: zero noise on the input u is assumed

and with these extra inputs, the Kalman filter yields, in addition to the state estimation $\hat{x}_k$, some extra outputs:

- Cov_x_k: covariance of present state estimate $\hat{x}_k$

    - this extra output is important, because it needs to be fed to the observer at the next iteration, along with $\hat{x}_k$
- $\hat{y}_k$: smoothed (denoised) estimate of the output measurement y_k

The Kalman filter state estimation algorithm is often decomposed into two steps:

1. **Predict** step: predict the next state using solely the knowledge of the previous state and the input, i.e. applying a noiseless state dynamics equation
2. **Update (correct)** step: use the actual ouput measurement to "fine-tune" the first predicion. it uses the output prediction error as a feedback and ensures the convergence of the estimation

(it can also be compactly written as in Ch8 of (Åström and Murray, 2021) textbook)

For the battery SoC estimation:

- Predict step corresponds to Current Counting, which is good for monitoring *relative* SoC changes, but giving no clue about the absolute SoC value.
- Update step exploits the changes in the OCV with respect to the SoC to make the SoC estimation converge (this implies that if the OCV(SoC) is flat, the Update step makes no changes).

Predict step

In [9]:
```
SoC = 0.5 # previous state
i = 4 # A, present current
v = 4.15 # V, present voltage measurement, higher than expected by 0.030 V

state_dyn =  state_dyn_batt
state_jac = state_jac_batt
output = output_batt
output_jac = output_jac_batt

x = v1(SoC) # previous state
u = v1(i) # present input
y = v1(v) # present output measurement
```

In [10]:
```
x_pred = state_dyn(x, u, **params)
print(f'predicted SoC: {x_pred[0]:.1%}')
x_pred
```

```
predicted SoC: 70.0%
```

Out[10]:
```
array([0.7])
```

Jacobian of state evoluation:

In [11]:
```
F = state_jac(x, u, **params) # also named A in the linear case
```

Covariance of predicted state:

In [12]:
```
Cov_x = m11(0.1 ** 2 ) # 10% SoC uncertainty
Cov_w = m11(1e-2 ** 2) # 1% SoC noise at each time step


Cov_xpred = F @ Cov_x @ F.T + Cov_w

std_SoC_pred  = sqrt(Cov_xpred[0,0]) # standard deviation (std)
print(f'predicted SoC std: {std_SoC_pred:.3%}')
Cov_xpred
```

```
predicted SoC std: 10.050%
```

Out[12]:
```
array([[0.0101]])
```

Update step

In [13]:
```
y_pred = output(x_pred, u, **params)
y_innov = y - y_pred
print(f"""voltage (output):
- predicted:   {y_pred[0]:.2f} V
- measured:    {y[0]:.2f} V
- innovation: {y_innov[0]:+.2f} V (meas - pred)
""")
y_innov
```

```
voltage (output):
- predicted:   4.12 V
- measured:    4.15 V
- innovation: +0.03 V (meas - pred)
```

Out[13]:
```
array([0.03])
```

output Jacobian (slope of OCV(SoC) curve)

In [14]:
```
H = output_jac(x_pred, u, **params)
H # V/SoC
```

Out[14]:
```
array([[0.6]])
```

In [15]:
```
std_Vmeas = 10e-3 # 10 mV voltage measurement noise
Cov_y = m11(std_Vmeas ** 2)

Cov_ypred = H @ Cov_xpred @ H.T
Cov_yinnov = Cov_ypred + Cov_y
print(f'innovation std: {sqrt(Cov_yinnov[0,0]):.5f} V')
print(f'(compared to    {sqrt(Cov_ypred[0,0]):.5f} V if measurement were noiseless)')

Cov_yinnov
```

```
innovation std: 0.06112 V
(compared to    0.06030 V if measurement were noiseless)
```

Out[15]:
```
array([[0.003736]])
```

In [16]:
```
Cov_yinnov_inv = inv(Cov_yinnov) # ⚠ Cov_yinnov needs to be invertible
Cov_yinnov_inv
```

Out[16]:
```
array([[267.66595289]])
```

Kalman gain (feedback gain of the measurement error)

Remarks

- when `H @ Cov_xpred @ H.T` is dominant compared to `Cov_y` (low measurement noise) in the computation of `Cov_yinnov`, then `L ~ 1/H` (in scalar case)
- when output Jacobian $H = 0$ (e.g. flat OCV(SoC) curve), then $L = 0$ and the Update step makes nothing

In [17]:
```
L = Cov_xpred @ H.T @ Cov_yinnov_inv
L, 1/H
```

Out[17]:
```
(array([[1.62205567]]), array([[1.66666667]]))
```

Prediction correction: the innovation error, amplified by Kalman gain, is added to the first prediction

In [18]:
```
L@y_innov # SoC correction
```

Out[18]:
```
array([0.04866167])
```

In [19]:
```
x_next = x_pred + L@y_innov
print(f'updated SoC estimate: {x_next[0]:.1%} (compared to {x_pred[0]:.1%} prediction

x_next
```

```
updated SoC estimate: 74.9% (compared to 70.0% prediction)
```

Out[19]:
```
array([0.74866167])
```

Covariance of updated state estimation:

In [20]:
```
I1 - L@H # Covariance scaling matrix, which should be shrinking so that Cov_x converg
```

Out[20]:
```
array([[0.0267666]])
```

In [21]:
```
Cov_xnext = (I1 - L@H)*Cov_xpred
std_SoC_next  = sqrt(Cov_xnext[0,0]) # standard deviation (std)
print(f'updated SoC std: {std_SoC_next:.3%} (compared to {std_SoC_pred:.3%} for the p
Cov_xnext
```

```
updated SoC std: 1.644% (compared to 10.050% for the predict step)
```

Out[21]:
```
array([[0.00027034]])
```

Updated measurement prediction and residual

In [22]:
```
y_upd = output_batt(x_next, u, **params)
y_res = y - y_upd
print(f"""voltage (output):
- updated pred: {y_upd[0]:.4f} V
- measured:     {y[0]:.4f} V (±{std_Vmeas:.4f} V)
- residual:    {y_res[0]:+.4f} V (meas - updated pred)
""")
y_innov
```

```
voltage (output):
- updated pred: 4.1492 V
- measured:     4.1500 V (±0.0100 V)
- residual:    +0.0008 V (meas - updated pred)
```

Out[22]:
```
array([0.03])
```

Wrapping everything in a function

Generic Kalman filter implementation

using generic control theory (x, u, y...) notation

In [23]:
```
def observer_ekf(x, u, y, funs, Cov, **params):
    """State observer using the extended Kalman filter (EKF)

    Parameters:
    - x: previous state estimate
    - u: present input
    - y: present output measurement
    - funs: state space model dict of functions, with keys state_dyn, state_jac, outp
```

```
    - Cov: covariance matrices dict, with keys x (state estimation), w (state evoluti

    Returns:
    - x_next: present state estimation
    - Cov_xnext: present state covariance
    - y_upd: smoothed (denoised) estimate of the output measurement
    """
    # Extract data
    Cov_x = Cov['x']
    Cov_w = Cov['w']
    Cov_y = Cov['y']

    # 1) Prediction step
    x_pred = funs['state_dyn'](x, u, **params)
    F = funs['state_jac'](x, u, **params)
    Cov_xpred = F @ Cov_x @ F.T + Cov_w

    # 2) Update step
    y_pred = funs['output'](x_pred, u, **params)
    y_innov = y - y_pred
    H = funs['output_jac'](x_pred, u, **params)
    Cov_yinnov = H @ Cov_xpred @ H.T + Cov_y
    Cov_yinnov_inv = inv(Cov_yinnov)
    L = Cov_xpred @ H.T @ Cov_yinnov_inv
    # updated state estimate:
    x_next = x_pred + L@y_innov
    Cov_xnext = (I1 - L@H)*Cov_xpred
    y_upd = output_batt(x_next, u, **params)

    return x_next, Cov_xnext, y_upd
```

Test of the implementation

In [24]:

```
SoC = 0.5 # previous state
i = 4 # A, present current
v = 4.14 # V, present voltage measurement, higher than expected by 0.030 V
std_Vmeas = 10e-3 # 10 mV voltage measurement noise

funs = dict(
    state_dyn = state_dyn_batt,
    state_jac = state_jac_batt,
    output = output_batt,
    output_jac = output_jac_batt
)

Cov = dict(
    x = m11(0.1 ** 2 ), # 10% SoC uncertainty
    w = m11(1e-2 ** 2), # 1% SoC noise at each time step
    y = m11(std_Vmeas ** 2)
)

x = v1(SoC) # previous state
u = v1(i) # present input
y = v1(v) # present output measurement

x_next, Cov_xnext, y_upd = observer_ekf(x, u, y, funs, Cov, **params)

std_SoC_next  = sqrt(Cov_xnext[0,0]) # standard deviation (std)
ΔSoC = i*params['Δt']/params['Qrated']

print(f"""State estimation (single step):
- Voltage measurement: {v:.4f} V ±{sqrt(Cov['y'][0,0]):.4f} V
- Current counting i.Δt/Q: {ΔSoC:+.1%}
- Prev SoC estimate: {x[0]:.1%} ±{sqrt(Cov['x'][0,0]):.1%}
```

```
- Next SoC estimate: {x_next[0]:.1%} ±{std_SoC_next:5.1%} → V={y_upd[0]:.4f} V
""")
```

```
State estimation (single step):
- Voltage measurement: 4.1400 V ±0.0100 V
- Current counting i.Δt/Q: +20.0%
- Prev SoC estimate: 50.0% ±10.0%
- Next SoC estimate: 73.2% ± 1.6% → V=4.1395 V
```

Compact implementation specialized for SoC estimation

In [25]:
```python
def SoC_ekf(SoC, i, v, Cov, **params):
    """Battery State of Charge estimation using the extended Kalman filter (EKF)

    A specialized compact version of the Kalman filter for SoC estimation,
    where the battery model, with one scalar state (the SoC) is baked-in,
    except for the OCV function.

    Parameters:
    - SoC: previous SoC estimate
    - i: present current
    - v: present voltage measurement
    - Cov: dict of variance scalars, with keys SoC (state estimation), w (state evolu

    Returns:
    - SoC_next: present state estimation
    - Cov_SoCnext: present state covariance
    - v_upd: smoothed (denoised) estimate of the voltage measurement
    """
    # Extract data
    Cov_SoC = Cov['SoC']
    Cov_w = Cov['w']
    Cov_v = Cov['v']
    Δt = params['Δt']
    Qrated = params['Qrated']
    Voc = params['Voc']
    R = params['R']
    Voc_jac = params['Voc_jac']

    # 1) Prediction step
    SoC_pred = SoC + i/Qrated*Δt # Jacobian F = 1.0
    Cov_SoCpred = Cov_SoC + Cov_w

    # 2) Update step
    v_pred = Voc(SoC_pred, **params) + R*i
    v_innov = v - v_pred
    H = Voc_jac(SoC_pred, **params)
    Cov_vinnov = H * Cov_SoCpred * H + Cov_v
    Cov_vinnov_inv = 1/Cov_vinnov
    L = Cov_SoCpred * H * Cov_vinnov_inv
    # updated state estimate:
    SoC_next = SoC_pred + L*v_innov
    Cov_SoCnext = (1 - L*H)*Cov_SoCpred
    v_upd = Voc(SoC_next, **params) + R*i

    return SoC_next, Cov_SoCnext, v_upd
```

Test of the implementation

In [26]:
```python
SoC = 0.5 # previous state
i = 4 # A, present current
v = 4.14 # V, present voltage measurement, higher than expected by 0.030 V
std_Vmeas = 10e-3 # 10 mV voltage measurement noise
```

```
Cov = dict(
    SoC = 0.1**2, # 10% SoC uncertainty
    w = 1e-2**2, # 1% SoC noise at each time step
    v = std_Vmeas**2
)

SoC_next, Cov_SoCnext, v_upd = SoC_ekf(SoC, i, v, Cov, **params)

std_SoC_next  = sqrt(Cov_SoCnext) # standard deviation (std)
ΔSoC = i*params['Δt']/params['Qrated']

print(f"""State estimation (single step):
- Voltage measurement: {v:.4f} V ±{sqrt(Cov['v']):.4f} V
- Current counting i.Δt/Q: {ΔSoC:+.1%}
- Prev SoC estimate: {SoC:.1%} ±{sqrt(Cov['SoC']):.1%}
- Next SoC estimate: {SoC_next:.1%} ±{std_SoC_next:5.1%} → V={v_upd:.4f} V
""")
```

```
State estimation (single step):
- Voltage measurement: 4.1400 V ±0.0100 V
- Current counting i.Δt/Q: +20.0%
- Prev SoC estimate: 50.0% ±10.0%
- Next SoC estimate: 73.2% ± 1.6% → V=4.1395 V
```

Test the SoC estimation

🚧 *test along charge and discharge trajectories: to be done* 🚧

A key question is how the SoC estimation behaves in the presence of modeling error (error on OCV curve error, series resistance R...).

References

References on battery models

QQ Yu, R Xiong, LY Wang and C Lin “A Comparative Study on Open Circuit Voltage Models for Lithium-ion Batteries”. *Chin. J. Mech. Eng.*, 2018. https://doi.org/10.1186/s10033-018-0268-8

References on state oberver and Kalman filter

Wikipedia, “Kalman filter” https://en.wikipedia.org/wiki/Kalman_filter

- two-step formulation is given in the linear case (with no effect of the input in the output equation)
- for the nonlinear case, the extended Kalman filter (EKF) is quickly described, but split in a separate article https://en.wikipedia.org/wiki/Extended_Kalman_filter

KJ Åström and RM Murray, “Chapter 8 - Output Feedback” in “Feedback Systems: An Introduction for Scientists and Engineers”, 2nd ed, 2021. https://fbswiki.org/wiki/index.php/Output_Feedback

- useful for a general introduction on state estimation (for linear systems)
- only the compact single step formulation is given

References on Kalman filter for battery state estimation

Series of three articles by Plett (the model presented here is the “simple model” in the 2nd article), which also discusses initialization, parameter estimation...:

GL Plett, “Extended Kalman filtering for battery management systems of LiPB-based HEV battery packs: Part 1. Background,” *Journal of Power Sources*, 2004. https://doi.org/10.1016/j.jpowsour.2004.02.031

GL Plett “... Part 2. Modeling and identification” https://doi.org/10.1016/j.jpowsour.2004.02.032

GL Plett “... Part 3. State and parameter estimation” https://doi.org/10.1016/j.jpowsour.2004.02.033