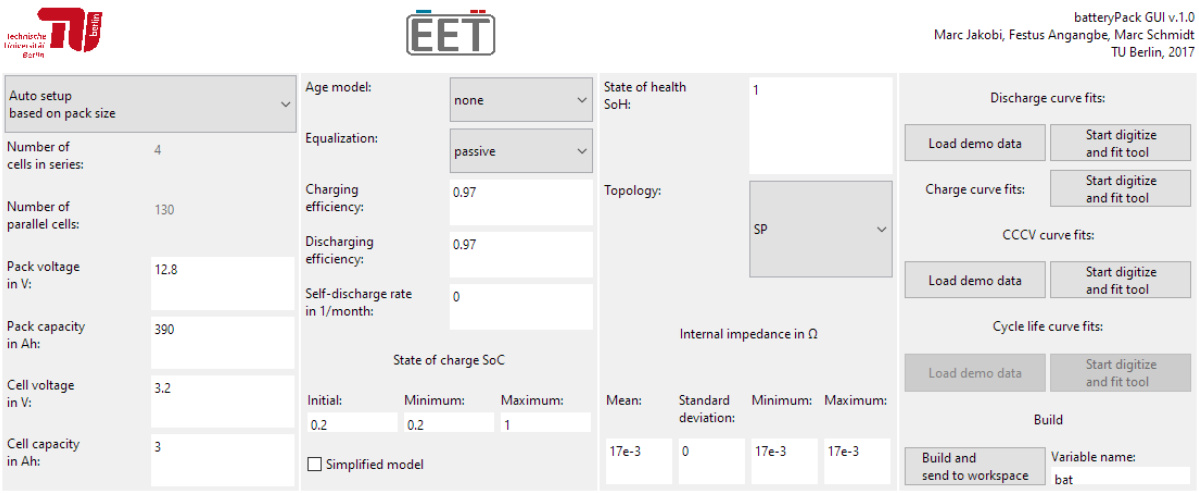

Cell Resolved Matlab® OOP Model of a Lithium Iron Phosphate Battery Pack

Marc Jakobi, Festus Anyangbe, Marc Schmitdt

March 2, 2017

HTW Berlin

Supervision:

M.Sc. Steven Neupert

TU Berlin

Contents

List of Figures

List of Tables

Acronyms

BMS battery management system

MEX Matlab® executable

OO object oriented

OOP object oriented programming

List of Symbols

Symbol	Unit	Description
C_{dis}	Ah	discharge capacity
$cDoC$	%	cycle-depth-of-cycle
DoD	-	depth of discharge
F	As/mol	Faraday constant
I	A	Current
N_{f}	-	number of cycles to failure
SoC	%	state of charge
SoC_{max}	%	maximum SoC
$SoC_{\text{max,l}}$	%	local maximum in an SoC profile
SoH	%	state of health
R	J/(mol · K)	universal gas constant
$rmse$	*various*	root mean squared error
T	K or °C	temperature
V	V	voltage
z_{Li}	-	ionic charge number of lithium

1 Discharge curves

Many battery data sheets provide measured discharge curves, on which the charging and discharging behaviour of this model is based. Rather than determining the curves according to the internal impedance, a common approach [1], this model determins the curves directly by means of digitizing the images and creating a curve fit. The classes used for fitting and modelling the discharge curves are described in the following subsections.

1.1 Single discharge curve

For modelling a single discharge curve, the class `dischargeFit` is used, which implements the interface `curveFitInterface`. The curve is fitted according to [2], using a function that is loosely based on the Nernst equation with two exonential functions superimposed as a correction for the voltage drops at the beginning and end of the curve.

Section describing interface, etc.

$$V(SoC) = x_1 - \frac{R \cdot T}{z_{\text{Li}} \cdot F} \cdot ln\left(\frac{SoC}{1 - SoC}\right) + x_2 \cdot SoC + x_3 + (x_4 + (x_5 + x_4 \cdot x_6) \cdot SoC) \cdot exp(-x_6 \cdot SoC) + x_7 \cdot exp(-x_8 \cdot SoC) \quad (1)$$

where x_1, .., x_8 are the fit parameters, $R = 8.3144598$ J/(mol · K) is the universal gas constant, $z_{\text{Li}} = 1$ is the ionic charge number of lithium, $F = 96485.3328959$ As/mol is the Faraday constant, SoC is the state of charge, V is the voltage in V and T is the temperature in K at which the curve was recorded. The curves are fitted using the levenberg-marquardt algorithm and either the `lsqcurvefit` method, the `fminsearch` method or a combination of both, depending on the user's preference.

1.1.1 Creation of a `dischargeFit` object

A `dischargeFit` object is created with the digitized raw data - the voltage V in V, the discharge capacity C_{dis} in Ah, the current I in A at which the curve was recorded and the temperature T in K at which the curve was recorded.

```
1 d = dischargeFit(V, C_dis, I, T);
```

`V` and `C_dis` are vectors containing the digitized raw data from the data sheet. Further options, such as initial values for the fit parameters x_1, .., x_8 and the fit method can be passed to the constructor using Matlab's name-value pair syntax:

```
1 d = dischargeFit(V, C_dis, I, T, 'OptionName', OptionValue);
```

By default, the initial fit parameters are set to zero and the curve is fit by first using `lsqcurvefit`, followed by `fminsearch`. The initial fit parameters are stored in a vector `x0`

of `length` 8, which can be passed via the option name `'x0'`, for example using the following syntax:

```
x0 = ones(8, 1);
d = dischargeFit(V, C_dis, I, T, 'x0', x0);
```

The method used for the curve fitting can be passed to the constructor using the option name `'mode'`. The corresponding value must be one of the following three strings:

- `'lsq'` for `lsqcurvefit`
- `'fmin'` for `fminsearch`
- `'both'` for `lsqcurvefit` followed by another fit using `fminsearch`

e.g.

```
d = dischargeFit(V, C_dis, I, T, 'mode', 'fmin');
d.plotResults
```

Depending on the curve and on the technology, one of the methods may return a better result.

1.1.2 Visual validation

A visual validation can be performed by calling the class's `plotResults` method (see above). In Figure 1, the results of two `dischargeFit` objects using the same raw data are compared.

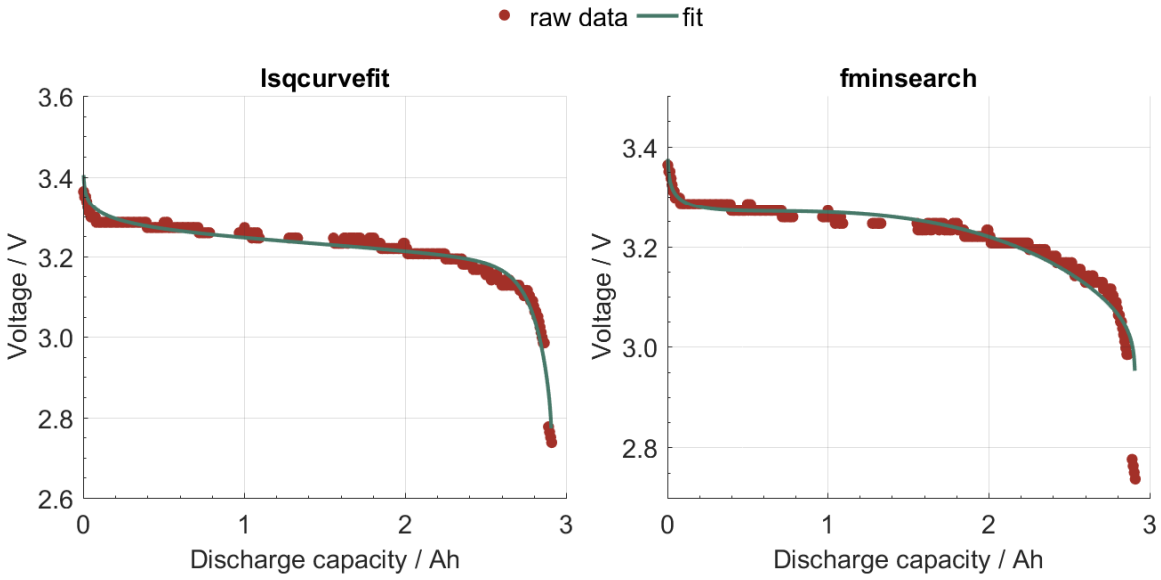

Figure 1: *Fit results of the `dischargeFit` class using the fit methods `lsqcurvefit` and `fminsearch`, respectively. The raw data was extracted from [3].*

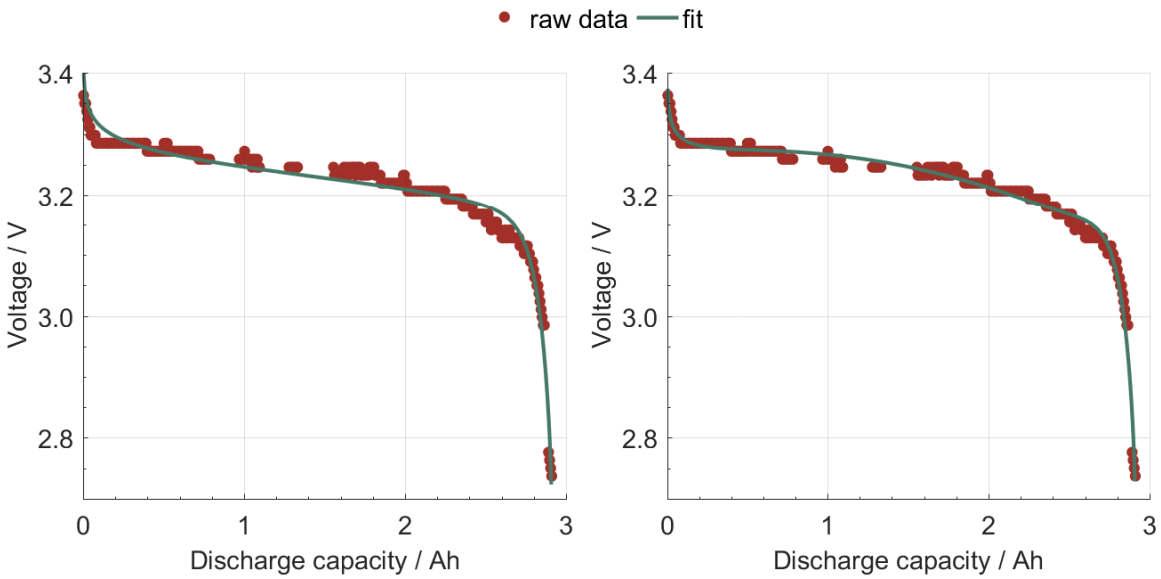

Figure 2: *Fit results of the* `dischargeFit` *class using the fit mode* `'both'` *with the default parameter initialization (left) and with a custom parameter initialization (right). The raw data was extracted from [3].*

In this example, `'lsq'` appears to return better results for the voltage drop at the end of the curve, while `'fmin'` results in a more precise fit for the voltage drop at the beginning of the curve. Further differences can be seen in the fits' curvatures. The `'lsq'` option results in a slightly flatter curve than the `'fmin'` mode. The results of a `dischargeFit` object using the `'both'` option are presented in Figure 2. Using the default fit parameter initialization of `zeros` (left) appears to improve the curvature and voltage drops slightly, compared to the other modes. Further improvements can be made by passing custom initial fit parameters to the constructor via the option `'x0'` (see Figure 2, right).

1.1.3 Object properties

Further fit quality analysis can be performed via the mean difference in voltage between the raw data and the curve fit at the respective positions of the raw data $\overline{\Delta V}$ in V and the maximum difference between the raw data and the curve fit at the respective positions ΔV_{max} in V. Additionally, every curve fit class (i.e. `dischargeCurves`, `woehlerFit`, etc.) in this package implements the `curveFitInterface`, which contains the root mean square error $rmse$ as a property. The $rmse$ for a curve fit with the raw data y_{raw} and the fitted data y_{fit} at the same respective x coordinates is defined as

$$rmse = \sqrt{\frac{\sum_{i=1}^{n}(|y_{\text{raw},i} - y_{\text{fit},i}|)^2}{n}} \tag{2}$$

where i is the index of the measurement and n is the number of measurements. In the case of a `dischargeFit` object, $y_{\text{raw},i}$ is the measured voltage at the discharge capacity $C_{\text{dis},i}$ and $y_{\text{fit},i}$ is the fitted voltage at $C_{\text{dis},i}$. Often used for forecasting models, the $rmse$ provides

Table 1: *Accessible properties of the* `dischargeFit` *class.*

Name	Description	Unit	Set access
`x`	8x1 vector of fit parameters	-	public
`dV_mean`	Mean voltage difference between raw data and fit	V	read only
`dV_max`	Max voltage difference between raw data and fit	V	read only
`T`	Temperature at which the curve was recorded	K	immutable
`z`	Current of the curve	A	immutable
`mode`	Method used for fitting (`'fmin'`, `'lsq'` or `'both'`)	-	public
`rmse`	Root mean square error	V	read only

a good measure of accuracy when comparing two models of the same data set [4]. In the previous examples, the curve fit using the `'lsq'` method (Figure 1, left) has an $rmse$ of 0.0244 V. Using the `'fmin'` mode (Figure 1, right) improves the $rmse$ to a value of 0.0162 V and using the fit mode `'both'` (Figure 2, left) further improves it to 0.0157 V. The lowest $rmse$ (0.0106 V) is achieved with the custom fit parameter initialization (Figure 2, right).
A list of the class's accessible properties is provided in Table 1. The `z` property is inherited from the `curveFitInterface`. Setting the `x` or `mode` properties will cause the object to re-run the fitting algorithm, thus likely resulting in different values for `x` than were set by the user.

1.1.4 Usage of a `dischargeFit` object

In order to calculate a voltage for a given discharge capacity, the object can be treated like a function handle, by using `subsref` indexing.

```
1 d = dischargeFit(V, C_dis, I, T, 'mode', 'fmin');
2 Cd = 1.5; % Discharge capacity in Ah
3 V = d(Cd); % Voltage in V
4 Cd_vect = linspace(0, 3, 1000); % Vector of discharge capacities in Ah
5 V_vect = d(Cd_vect); % Corresponding vector of voltages in V
```

A `dischargeFit` object is not accessed directly by the battery model, but rather stored in a `dischargeCurves` object. After creating a `dischargeFit`, it can be added to a `dischargeCurves` collection by using the `add()` method (see section 1.2). Alternatively, it can be added directly to a subclass of the `batteryInterface` (see section) using it's `addcurves()` method.

section ref

1.2 Collection of discharge curves

A single discharge curve can be used to model the behaviour of a battery for a given current. However, in reality, a battery will often be charged or discharged with different currents. In many cases, the current may change from one simulation time step to another. In order to be able to determine the voltage as a function of C_{dis} and I, multiple `dischargeFit` objects are wrapped by a `dischargeCurves` object, which is described in the following sections.

1.2.1 Creation of a `dischargeCurves` object

There are two ways to initialize a `dischargeCurves` object. The first option is to create an empty object and using the class's `dischargeFit()` method to add curve fits. The `dischargeFit()` method has the same syntax as the `dischargeFit` class's constructor.

```
dC = dischargeCurves;
I = [0.6; 1; 3; 5; 10; 20]; % Vector of currents in A
T = 293; % Temperature in K
for i = 1:6
    dC.dischargeFit(raw(i).V, raw(i).Cd, I(i), T)
end
% raw is a struct array containing the measured curve data
% from the data sheet.
```

This option has the advantage of reducing clutter in the workspace. However, changing the parameters and analysing the accuracy of the individual curve fits is more complicated. Alternatively, the `dischargeFit` objects can be created, modified and then passed to the `dischargeCurves` constructor.

```
d1 = dischargeFit(raw(1).V, raw(1).Cd, I(1), T);
    % Quality analysis and fit perfection here...
    % More curve dischargeFit object initializations here...
d6 = dischargeFit(raw(6).V, raw(6).Cd, I(6), T);
    % Quality analysis and fit perfection here...
dC = dischargeCurves(d1, d2, d3, d4, d5, d6);
% Equivalent:
dC = dischargeCurves;
dC.add(d1)
dC.add(d2)
% ...
dC.add(d6)
```

If a `dischargeFit` is passed to a `dischargeCurves` object that already holds a reference to a `dischargeFit` with the same current, the stored reference is replaced by the new one. Similarly, if two or more `dischargeFit` objects with the same current are passed to a `dischargeCurves` constructor, the first one is ignored.

1.2.2 Interpolation between curves

The calculation of the voltage for any given current and discharge capacity is done via Matlab's built-in `griddedInterpolant` class, which is called from within the `interp()` method. The syntax for a `dischargeCurves` object `dC` is as follows:

```
1 V = dC.interp(I, Cd);
2 V = interp(dC, I, Cd); % equivalent
```

Where `V` is the voltage in V, `I` is the charging or discharging current in A and `Cd` is the discharge capacity after charging or discharging in Ah. If `I` is equal to one of the stored `dischargeFit` objects' currents, `Cd` is simply passed on to the respective object, which returns the voltage. If `I` does not match any of the stored objects and either of the input arguments is not found in the object's cache, `Cd` is passed on to each of the stored `dischargeFit` references, creating a vector of voltages for the different currents. Finally, a `griddedInterpolant` is created using the sample points, `I` is passed to it and the interpolated voltage is returned and cached. The interpolation method (the default is `'spline'`) can be changed by setting the property `interpMethod`.

A visual validation of the interpolation using the `'linear'` and `'spline'` methods, respectively, is depicted in Figure 3. A collection of `dischargeFit` objects for six currents was created and the fit results were plotted. Then, all fits except for the one at 10 A were added to a `dischargeCurves` object. Finally, the `interp()` method was called for a current of 10 A and a range of discharge capacities, in an attempt to replicate the `dischargeFit` results using interpolation. The linearly interpolated curve (Figure 3, left) is almost identical to the fit until the beginning of the voltage drop at the end. However, the spline interpolation results

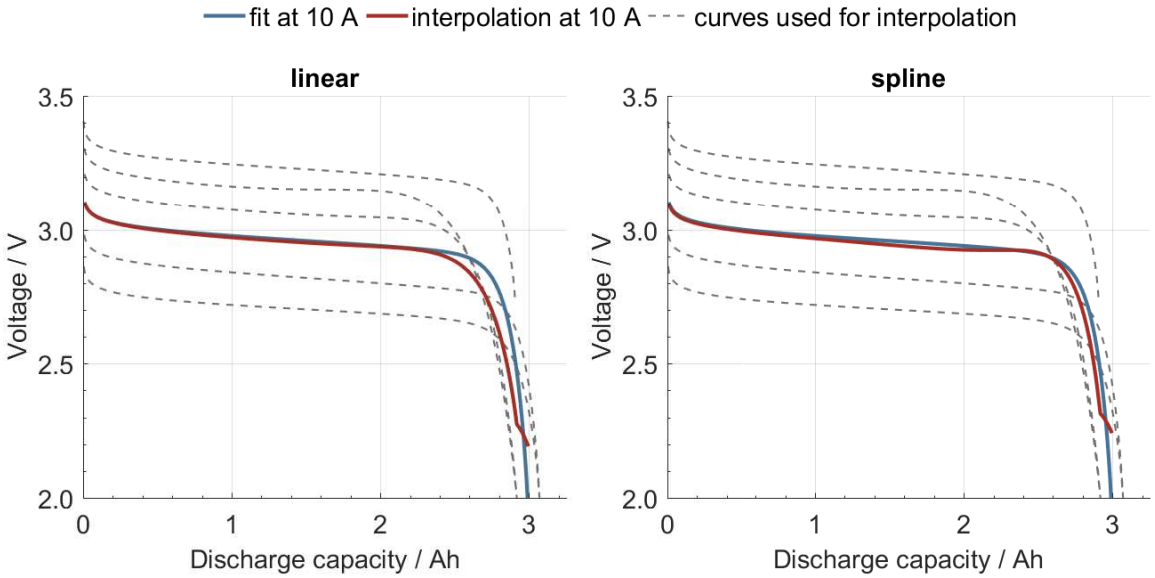

Figure 3: *Comparison of the `dischargeCurves` results using linear interpolation and spline interpolation, respectively. The raw data was extracted from [3].*

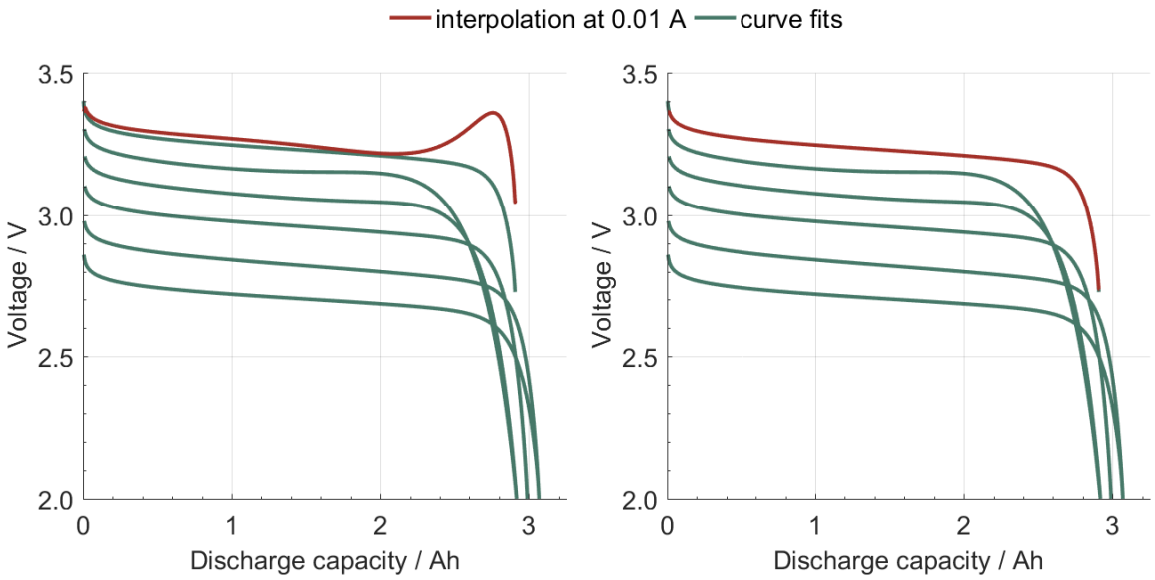

Figure 4: *Result of the `interp()` method for a current below the lowest measured current without output limitation (left) and with output limitation (right). The raw data was extracted from [3].*

in an overall more precise replication of the fit if the entire curve is regarded. This indicates that the most suitable interpolation method may depend on the maximum depth of discharge DoD of the modelled battery. As can be seen in Figure 3, the interpolation bends slightly at the end of the curve (close to a discharge capacity of 3 Ah). This is due to the fact that the voltage returned by a `dischargeFit` object is limited to the minimum and maximum of the raw data, respectively. If it were not limited, it could return `-Inf` or `Inf`, causing the interpolation to fail. Since most lithium ion batteries' DoD are limited to 0.8 or 0.9, this bend should rarely cause any issues.

As demonstrated Figure 4 (left), the `interp` method using spline interpolation does not provide a good extrapolation of currents. In order to correct this, the voltage output is limited by the curve fit with the lowest current I_{min} and by the curve fit with the highest current I_{max}, respectively. As a result, the `dischargeFit` recorded at I_{min} is called for any current below I_{min} (see Figure 4, left) and the `dischargeFit` recorded at I_{max} is called for any current above I_{max}. In this model, the battery's maximum discharging current is limited by the `dischargeCurves` object's I_{max} (see section).

ref section

1.2.3 Usage of a `dischargeCurves` object

Similarly to a `dischargeFit`, the results of a `dischargeCurves` object can be visually validated using the `plotResults()` method. Individual curve fit references removed using the `remove()` method and the respective currents.

```
% d = dischargeFit object
% I = current
dC.add(d) % add d to dischargeCurves dC
dC.remove(I) % remove the dischargeFit object with current I from dC
```

In order to access the `dischargeFit` references stored within a `dischargeCurves` object, the `createIterator()` method can be used. This creates an iterator object, a Matlab® implementation of the `java.util.iterator` interface [5]. The object can be used to iterate through the wrapped `dischargeFit` objects using a similar syntax to that of a JAVA™ iterator.

```
it = dC.createIterator; % returns an scIterator object
while it.hasNext % returns true if there is another object to
                 % iterate through
    d = it.next; % returns a dischargeFit object
    % more code here
end
it.reset % resets the scIterator
```

For usage in a battery model, a `dischargeCurves` object is passed to an implementation of the `batteryInterface` (see section) using it's `addCurves()` method.

ref section

2 Age model

The age model is implemented using the Observer design pattern via Matlab's "Events and Listeners[i]" [6]. This way, various age models (predefined or custom) can be dynamically added to a battery model at run time or even left out completely. Ageing can be simulated on the battery pack level (by treating all cells as one entity) or on the cell level (by observing each cell separately). The event oriented age model provided in this package is based solely on cycle counting, for which a mathematical approach developed by [7] is implemented. Descriptions of the counting algorithm and the classes used to implement the age model are provided in the following sections.

2.1 Overview

Cycle counting algorithms are designed to count cycles from a set of measured data. A challenge for a running simulation or a battery management system (BMS) that relies on cycle counting is to decide when to count the cycles of an accumulated data set. Counting could be done at fixed time intervals or it could be triggered by a certain event. The latter is the approach implemented by the `cycleCounter` interface, which acts both as an observer of a battery cell or pack as well as a subject for the `eoAgeModel` class.
The observation of charge cycles is handled by the abstract `cycleCounter` interface, in which all methods except for the `count()` method are predefined. An object that implements the interface is regularly updated with the observed battery's SoC, which is stored within the object's memory. The cycle counting occurs every time the SoC reaches an upper threshold, i.e. the observed battery's maximum SoC. After counting, the `NewCycle` event is triggered, causing all of the object's observers (i.e. an `eoAgeModel` object) to be notified that new data is available for simulation. The age model then uses the data to determine the battery's new state of health SoH and passes it on to the battery.
An Observer Pattern class diagram of the age model is depicted in Figure 5. The observation is handled by the respective abstract interfaces, while the actual simulation is handled by the implementations. This makes the model highly flexible. For example, a lightweight implementation could be to observe a `batteryPack` using a single `cycleCounter` and a single `batteryAgeModel`. Another option could be to use multiple `cycleCounter` and `ageModel` objects in order to simulate the ageing of each `batteryCell` within a pack individually. The cycle counting can be implemented by various algorithms (two are provided in this package). And advanced users could even replace the default age model implementation (`eoAgeModel`) with a custom class that takes other factors into account, e.g. calendar ageing or thermal influences. In large simulations, it may be of interest to neglect the battery ageing in order to save simulation time. This can either be done by simply not linking up the components at runtime or by including a `dummyCycleCounter` and a `dummyAgeModel`. These classes

[i] in Matlab®, an observer is often referred to as a "listener". However, "observer" is the more common term in OOP design pattern terminology and will be used throughout this documentation.

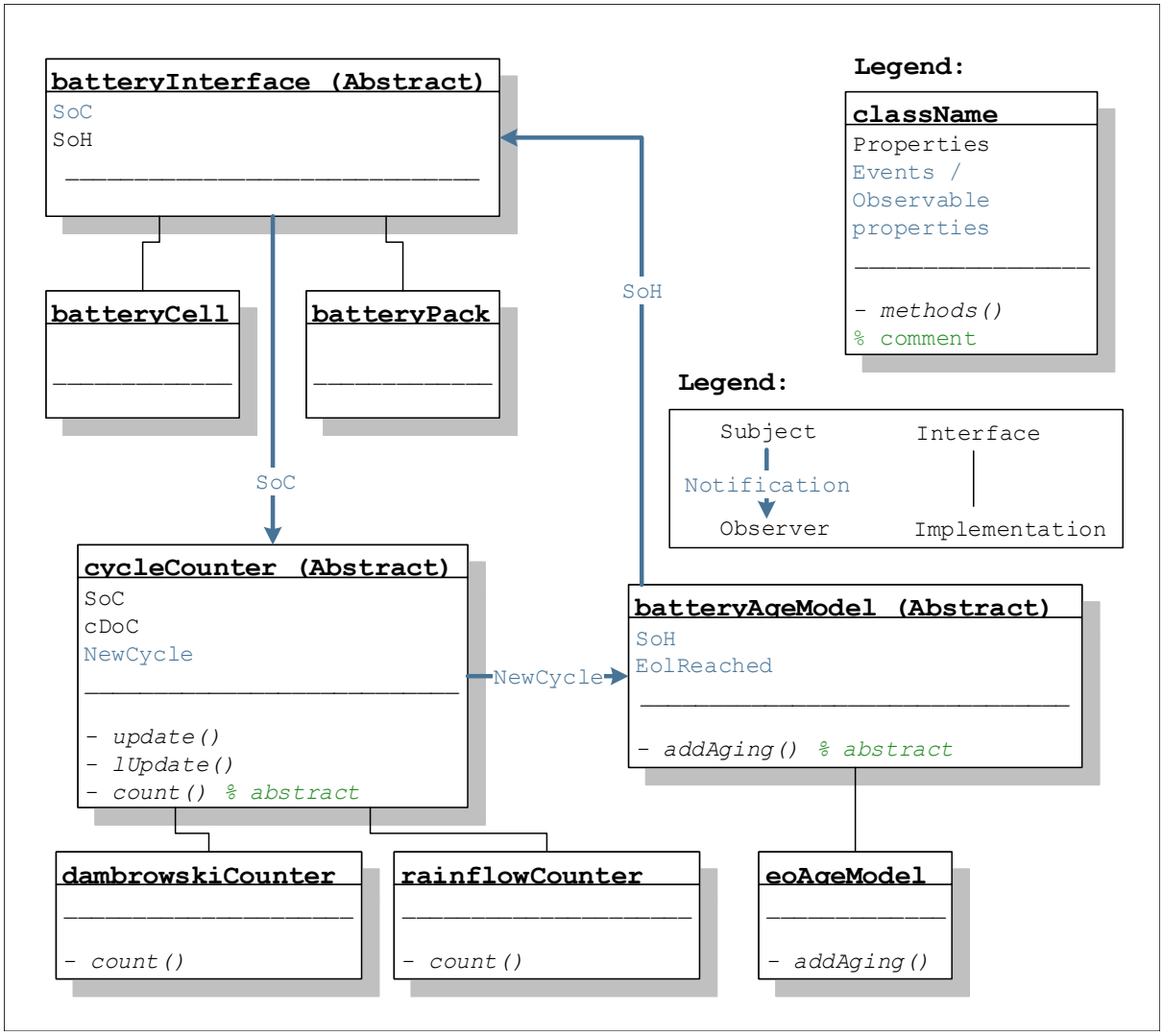

Figure 5: *Overview of the Observer implementation of the age model with communication flows and inheritance links.*

implement the `cycleCounter` and `batteryAgeModel` interfaces, respectively. However, calling their methods does nothing. The former option is faster, due to reduced method overhead, but the latter may be more robust in some cases.

2.2 Cycle counting

In this pack, two classes have been created to implement the `cycleCounter`'s `count()` method. A `cycleCounter` subclass can be constructed in one of the following ways:

```
1 c = cycleCounter; % sets the initial SoC to 0.2 and the max. SoC to 1
2 c = cycleCounter(init_soc); % sets the initial SoC
3 c = cycleCounter(init_soc, soc_max); % sets the initial SoC and the
4                                      % max. SoC
```

where `cycleCounter` must be replaced with the name of the respective class that is being constructed (e.g. `dambrowskiCounter` or `rainflowCounter`). To register the object as an observer of a battery object `bat`[i], the `initAgeModel()` method can be used.

ref section

```
% Extract intitial SoC and max. SoC from battery
init_soc = bat.SoC;
soc_max = bat.socMax;
% Replace "cycleCounter" with the respective subclass
c = cycleCounter(init_soc, soc_max);
% Intitialize event oriented age model with cycle counter c
bat.initAgeModel('ageModel', 'EO', 'cycleCounter', c)
```

Note that the `'ageModel'` option must be specified, otherwise `bat` will internally replace `c` with a `dummyCycleCounter` object to prevent runtime errors. If this happens, a warning message is printed to the command window.

2.2.1 The `rainflowCounter` class

The state of the art algorithm for cycle counting, "rainflow", was originally developed for mechanical stress modelling [8] and has recently become popular in the field of battery charge cycle counting [9]. The `rainflowCounter` class was added to this package for the purpose of demonstrating the flexibility of the age model implementation. It acts as an adapter for the popular FileExchange contribution, "Rainflow Counting Algorithm" by Adam Nieslony [10]. In order for the class to work, the MEX functions must be downloaded from [10] and placed within Matlab's search path. They are not included in this package and attempting to construct a `rainflowCounter` object will fail if they are not found. To register a `rainflowCounter` with a battery `bat`, the above syntax must be used, whereby `cycleCounter(init_soc, ... soc_max)` is replaced by `rainflowCounter(init_soc, soc_max)`.

2.2.2 The `dambrowskiCounter` class

In 2012, J. Dambrowski, S. Pichlmaier and A. Jossen developed a mathematical definition of a battery's charge cycles along with an algorithm for counting them [7]. Since the counting algorithm was not named, the `dambrowskiCounter` class that implements it in this package was named after one of the authors. In their approach, so-called pre-cycles are counted and compared with each other. This is visualized in Figure 6. The twice depicted SoC curve (grey) has two local maxima $SoC_{\mathrm{max,l},i}$, since the last value is counted as a local maximum. Starting from an $SoC_{\mathrm{max,l},i}$, a pre-cycle of "prior equality" is defined as the SoC within an interval between the respective $SoC_{\mathrm{max,l}}$ and the last point at which the SoC was equal to $SoC_{\mathrm{max,l}}$. Two such pre-cycles are depicted in Figure 6 (left) and coloured in red and blue, respectively. A pre-cycle of "subsequent equality" (Figure 6, right, coloured in blue) is defined as the SoC

[i] `bat` can be an object of any class that implements the `batteryInterface`.

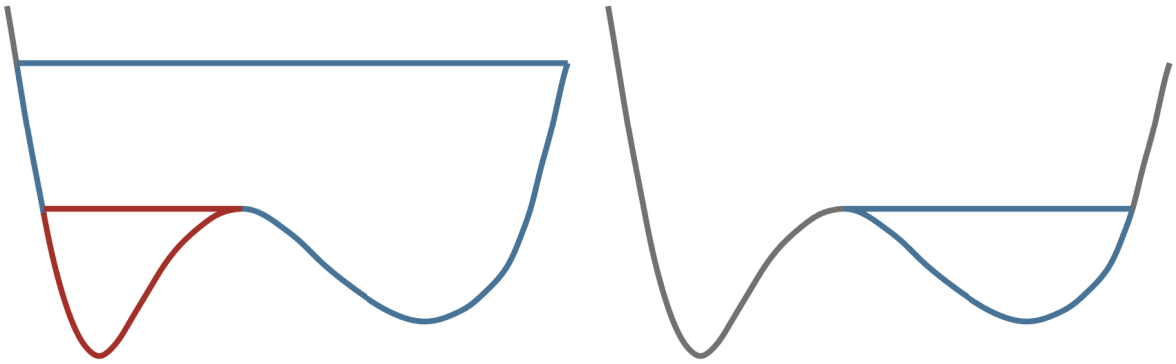

Figure 6: *Qualitative visualization of pre-cycle counting according to [7]: Two pre-cycles of prior equality (left) and a pre-cycle of subsequent equality (right).*

within an interval between the respective $SoC_{\text{max,I}}$ and the subsequent point at which the SoC is equal to $SoC_{\text{max,I}}$. Finally, a pre-cycle is counted as a cycle if there is no larger pre-cycle that encompasses the same interval and shares the same local minimum. This is not the case for the small cycle (coloured red) in Figure 6 (left); so the depicted curve contains two cycles.

In this package, `dambrowskiCounter` is the default cycle counter if an age model is specified. Thus, it does not have to be passed as an argument in a battery's `initAgeMode()` method.

```
1 bat.initAgeModel('ageModel', 'EO')
2 % Automatically initializes a dabrowskiCounter object with init_soc
3 % and soc_max set according to the battery's properties and links.
```

2.2.3 Comparison of the cycle counters

Each `cycleCounter` object's `count()` method converts the saved SoC profile into a cycle-Depth-of-Cycle $cDoC$ curve - a vector containing the depths of discharge DoD of all the counted cycles. The simulation results of two batteries using a `dambrowskiCounter` and a `rainflowCounter`, respectively, are compared in Figure 7. The cycles' DoDs are each sorted into 50 BINs and compared in a histogram. Overall, the histograms appear very similar, thus proving that both classes produce good results. However, more cycles are counted using the `dambrowskiCounter` class, possibly causing the simulated battery to age slightly faster. While both classes use different methods for determining the extrema[i], the amount local maxima found is the same. Thus, the determination of extrema can be ruled out as a cause

[i] `dambrowskiCounter` uses `cycleCounter`'s `iMaxima()` method and `rainflowCounter` uses the `sig2ext()` function [10].

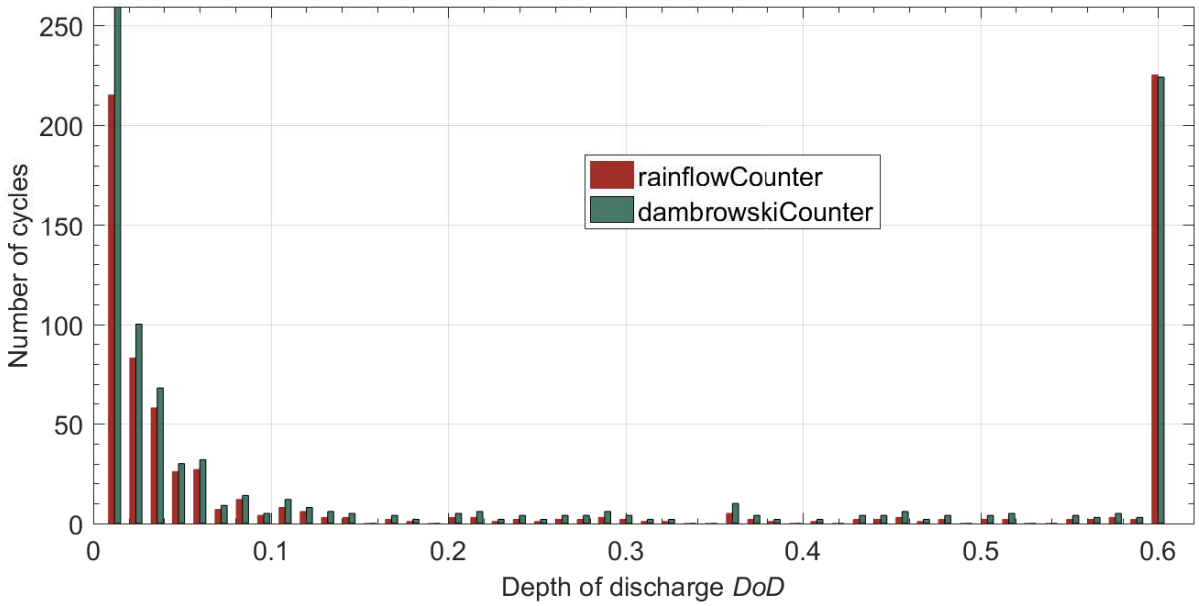

Figure 7: *Comparison of the counted cycles and their DoD between two simulations using the* `rainflowCounter` *and the* `dambrowskiCounter` *using the same SoC profile.*

and the root of the discrepancy must lie within the different counting approaches.

2.3 Event oriented ageing model

The event oriented ageing model - a very simple and lightweight model - is implemented by the `eoAgeModel` class, which subclasses the abstract `batteryAgeModel` interface. Cycle ageing is calculated based on a curve fit of the battery's number of cycles to failure N_f vs DoD curve.

2.3.1 Cycle life curve fits

Due to the fact that the cycle ageing can vary strongly between technologies, it can be difficult to find a good fit for the N_f vs DoD curve. In order to provide some flexibility, three different classes, each implementing the `curveFitInterface`, are provided for fitting such curves in this package: `woehlerFit`, `nrelcFit` and `deFit`. They only differ in their class names, the number of fit parameters and in the functions used for fitting. The function used in `woehlerFit` is based on a metal fatigue curve (also known as a "Wöhler curve") [11]

$$N_\mathrm{f}(DoD) = x_1 \cdot DoD^{-x_2} \tag{3}$$

with the fit parameters x_1 and x_2. The `nrelcFit` class bases it's fit method on an older model [12].

$$N_\mathrm{f}(DoD) = x_1 \cdot \frac{1}{DoD} \cdot exp\left(x_2 \cdot \left(1 - \frac{1}{DoD}\right)\right) \tag{4}$$

Finally, the `deFit` class uses a double exponential function that was originally developed for lead-acid batteries [13]. However, it also seems to provide decent results for lithium-ion

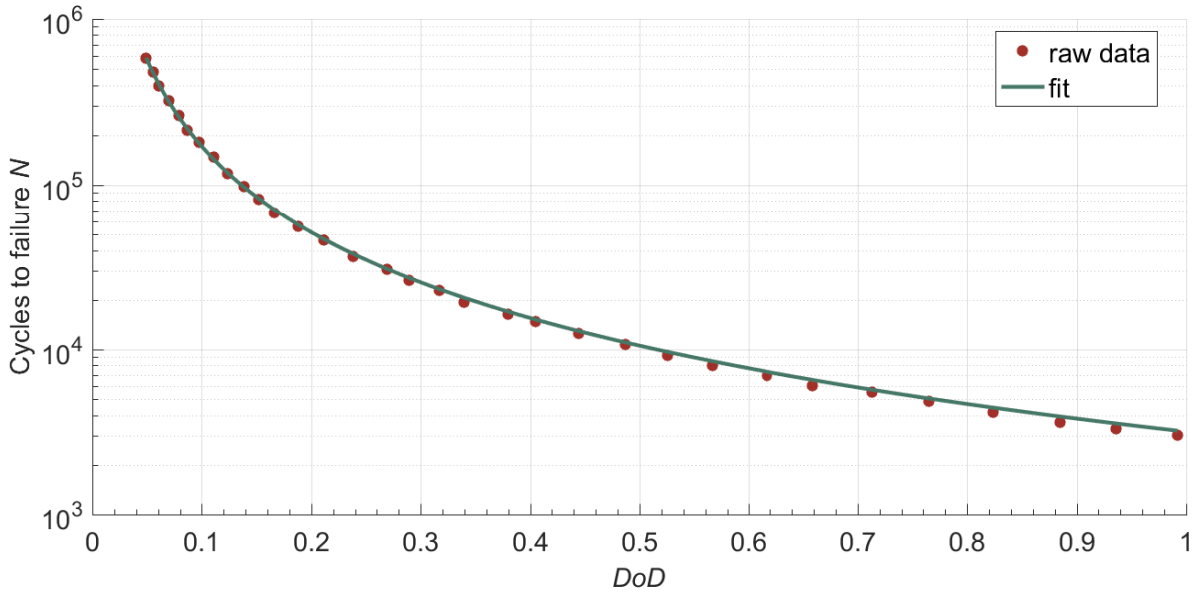

Figure 8: *Example for a lithium ion battery's N_f vs DoD cycle life curve fitted using the* `woehlerFit` *class.*

batteries in some cases.

$$N_f(DoD) = x_1 + x_2 \cdot exp(-x_3 \cdot DoD) + x_4 \cdot exp(-x_5 \cdot DoD) \tag{5}$$

An example for the fit results of a `woehlerFit` object is depicted in Figure 8. Due to a lithium-ion battery's extremely large amount of cycles to failure at low DoDs, large relative errors may occur, especially at DoDs close to 1. In order to reduce the $rmse$, as many raw data points as possible should be provided for fitting.

2.4 Creating a user-defined age model

```
1  classdef myCalendarAgeModel < lfpBattery.eoAgeModel
2  %MYCUSTOMAGEMODEL: An example for a user-defined age model.
3  %Combines the event oriented age model with a linear calendar age model.
4
5      properties
6          L_cal; % calendar life in s
7      end
8
9      methods
10         function obj = myCalendarAgeModel(l, varargin)
11             % l = calendar life in years
12             % varargin = input args of eoAgeModel constructor
13             %% call superclass constructor
14             obj = obj@lfpBattery.eoAgeModel(varargin{:});
15             obj.L_cal = l * 525600  / obj.eolAc; % set L_cal in
```

```
16              % seconds taking end of life age into account
17          end
18          function addCalAge(obj, dt)
19              % Adds to the battery's age using the simulation time
20              % step size.
21              % dt = simulation time step size in s
22              % obj.Ac = 1 - obj.SoH
23              obj.Ac = obj.Ac + dt / obj.L_cal; % increment age
24          end
25      end
26  end
```

References

[1] Lijun Gao, Shengyi Liu, and R. Dougal, "Dynamic lithium-ion battery model for system simulation," *IEEE Transactions on Components and Packaging Technologies*, vol. 25, no. 3, pp. 495–505, Sep. 2002, ISSN: 1521-3331. DOI: 10.1109/TCAPT.2002.803653.

[2] F. V. Werder, "Entwicklung eines batteriemodells auf lithium basis," PhD thesis, HTW Berlin, Berlin, Sep. 30, 2014.

[3] "Data sheet: BM26650etc1 - high power lithium iron phosphate cell," Batterien-Montage-Zentrum GmbH (BMZ), Karlstein, 2010. [Online]. Available: www.bmz-gmbh.de.

[4] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, no. 4, pp. 679–688, Oct. 2006, ISSN: 01692070. DOI: 10.1016/j.ijforecast.2006.03.001.

[5] (). Iterator (java platform SE 7), [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html (visited on 02/27/2017).

[6] (). Overview events and listeners - MATLAB & simulink - MathWorks united kingdom, [Online]. Available: https://uk.mathworks.com/help/matlab/matlab_oop/learning-to-use-events-and-listeners.html?s_tid=gn_loc_drop (visited on 02/28/2017).

[7] J. Dambrowski, S. Pichlmaier, and A. Jossen, "Mathematical methods for classification of state-of-charge time series for cycle lifetime prediction," in *Advanced Automotive Battery Conference Europe*, Mainz, Jun. 2012.

[8] M. Matsuishi and T. Endo, "Fatigue of metals subjected to varying stress," in *Mech. Engineering*, Japan, 1968.

[9] R. Dufo-López and J. L. Bernal-Agustín, "Multi-objective design of PV–wind–diesel–hydrogen–battery systems," *Renewable Energy*, vol. 33, no. 12, pp. 2559–2572, Dec. 2008, ISSN: 09601481. DOI: 10.1016/j.renene.2008.02.027.

[10] (). Rainflow counting algorithm - file exchange - MATLAB central, [Online]. Available: https://uk.mathworks.com/matlabcentral/fileexchange/3026-rainflow-counting-algorithm (visited on 02/27/2017).

[11] M. Naumann, C. N. Truong, R. C. Karl, and A. Jossen, "Betriebsabhängige kostenrechnung von energiespeichern," in *13. Symposium Energieinnovation*, Graz, 2014.

[12] S. Drouilhet, B. Johnson, S. Drouilhet, and B. Johnson, "A battery life prediction method for hybrid power applications," in *35th Aerospace Sciences Meeting and Exhibit*, 1997, p. 948.

[13] H. Bindner, T. Cronin, P. Lundsager, Risø National Lab, and Roskilde (DK). Wind Energy Department, *Lifetime modelling of lead acid batteries*. 2005, OCLC: 826678302, ISBN: 978-87-550-3441-9.