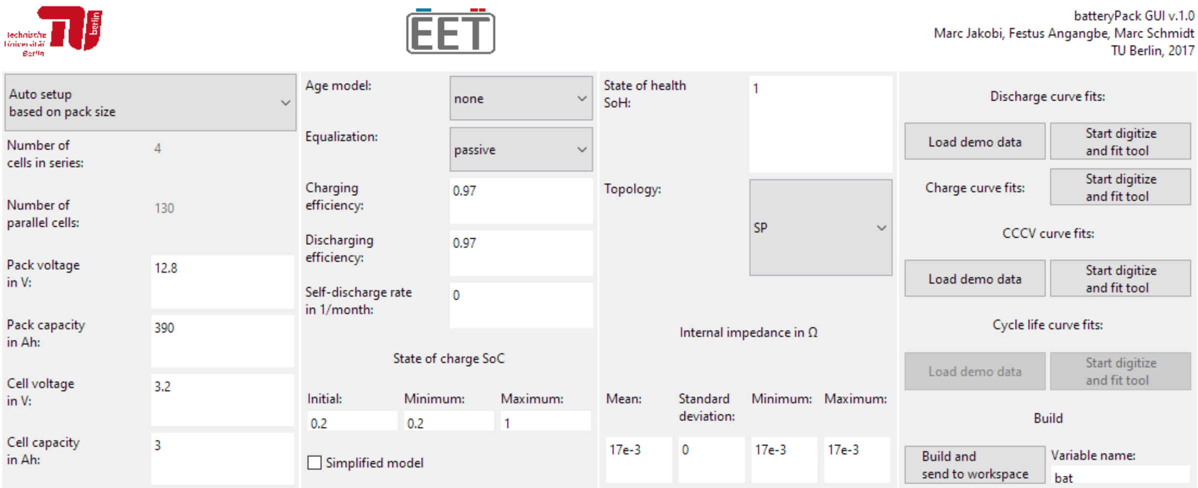

Cell Resolved Matlab® OOP Model of a Lithium Iron Phosphate Battery Pack

Marc Jakobi, Festus Anyangbe, Marc Schmitdt

March 14, 2017

HTW Berlin

Supervision:

M.Sc. Steven Neupert

TU Berlin

Contents

1 Discharge curves **1**

1.1 Single discharge curve . . . 1

1.1.1 Creation of a `dischargeFit` object . . . 1

1.1.2 Visual validation . . . 2

1.1.3 Object properties . . . 3

1.1.4 Usage of a `dischargeFit` object . . . 4

1.2 Collection of discharge curves . . . 5

1.2.1 Creation of a `dischargeCurves` object . . . 5

1.2.2 Interpolation between curves . . . 6

1.2.3 Usage of a `dischargeCurves` object . . . 7

2 Age model **9**

2.1 Overview . . . 9

2.2 Cycle counting . . . 10

2.2.1 The `rainflowCounter` class . . . 11

2.2.2 The `dambrowskiCounter` class . . . 11

2.2.3 Comparison of the cycle counters . . . 12

2.3 Event oriented ageing model . . . 13

2.3.1 Cycle life curve fits . . . 13

2.3.2 Cycle ageing . . . 15

2.3.3 Age model initialization . . . 15

2.3.4 Creating a user-defined age model . . . 16

2.3.5 Calendar ageing . . . 17

3 Battery Composition **18**

3.1 Overview . . . 18

3.2 Method delegation . . . 19

3.3 Battery Interface . . . 21

3.3.1 Battery object initialization . . . 21

3.3.2 Battery charging and discharging . . . 23

3.3.3 Age model level . . . 28

3.4 CCCV charging and BMS . . . 28

3.4.1 CCCV curve fits . . . 29

3.4.2 Cell monitoring and communication with the charger . . . 30

References **32**

List of Figures

1 Fit results of the dischargeFit class using the fit methods lsqcurvefit and fminsearch, respectively . . . 2
2 Fit results of the dischargeFit class using the fit mode 'both' with the default parameter initialization and with a custom parameter initialization . . 3
3 Comparison of the dischargeCurves results using linear interpolation and spline interpolation, respectively . . . 6
4 Result of the interp() method for a current below the lowest measured current without output limitation and with output limitation . . . 7
5 Overview of the Observer implementation of the age model with communication flows and inheritance links . . . 10
6 Qualitative visualization of pre-cycle counting: Two pre-cycles of prior equality and a pre-cycle of subsequent equality . . . 12
7 Comparison of the counted cycles and their DoD between two simulations using the rainflowCounter and the dambrowskiCounter using the same SoC profile . . . 13
8 Example for a lithium ion battery's N_{f} vs DoD cycle life curve fitted using the woehlerFit class . . . 14
9 Visualization of the possible battery topology compositions . . . 18
10 Class diagram of the battery composition with communication flows and inheritance links . . . 19
11 Example of a method being delegated across a battery pack composition . . 20
12 Flow chart of the powerRequest() and currentRequest() methods . . 23
13 Flow chart of the iteratePower() method . . . 24
14 Flow chart of the iterateCurrent() method . . . 25
15 Comparison of battery pack charging simulations using a constant current, a linearly increasing current and a random current, respectively - Voltage vs. simulation time . . . 26
16 Comparison of battery pack charging simulations using a constant current, a linearly increasing current and a random current, respectively - Voltage vs. SoC 27
17 Simulation of battery charging with random currents using the mdischargeCurves class for voltage calculation . . . 27
18 Qualitative example of a CCCV charging curve . . . 29
19 Linear curve fits of $I_{\mathrm{max}}(SoC)$ during the CV phase for two different battery cells 30
20 Schematic visualization of the BMS combined with CCCV charging . . . 31

List of Tables

1 Accessible properties of the dischargeFit class . . . 4

Acronyms

BMS battery management system

CCCV constant current / constant voltage

EOL end of life

MEX Matlab® executable

OO object oriented

OOP object oriented programming

SP strings of parallel elements

PS parallel strings

List of Symbols

Symbol	Unit	Description
A_{c}	-	battery's age
A_{cal}	-	calendar ageing
$A_{\mathsf{c,eol}}$	-	age at which the end of life is reached
A_{cyc}	-	cycle ageing
A_{tot}	-	total ageing stress
C	Ah	battery capacity
C_{bu}	Ah	usable capacity
C_{dis}	Ah	discharge capacity
$cDoC$	%	cycle-depth-of-cycle
C_{n}	Ah	nominal capacity
DoD	-	depth of discharge
F	As/mol	Faraday constant
I	A	current
I_{max}	A	battery's maximum current
L_{cal}	a	calendar life
N_{f}	-	number of cycles to failure
P	W	power
P_{sd}	W	self-discharge
p_{z}	-	impedance proportionality factor
SoC	%	state of charge
SoC_{max}	%	maximum SoC
$SoC_{\mathsf{max,l}}$	%	local maximum in an SoC profile
SoC_{min}	%	minimum SoC
SoH	%	state of health
R	J/(mol · K)	universal gas constant
$rmse$	*various*	root mean squared error
T	K or °C	temperature
t_{s}	s	simulation time step
V	V	voltage
V_{n}	V	nominal voltage
Z_{i}	Ω	internal impedance
z_{Li}	-	ionic charge number of lithium
Δt_{s}	s	simulation time step size

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.

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

where $x_1, \; .., \; x_8$ are the fit parameters, $R = 8.3144598$ J/(mol · K) is the universal gas constant, $z_{\mathrm{Li}} = 1$ 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.

```
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:

```
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.

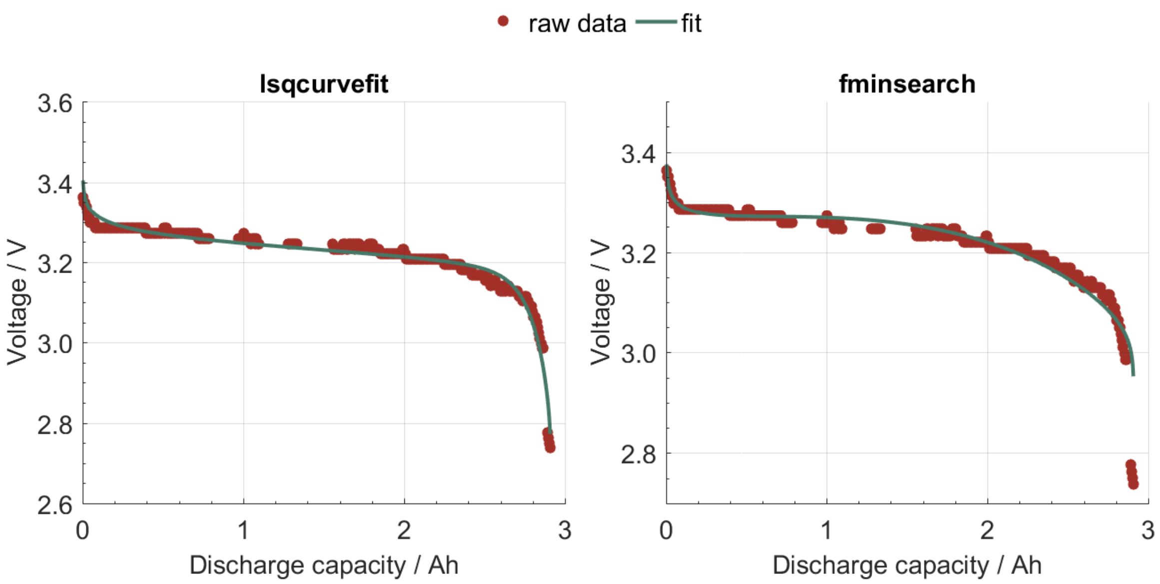

Figure 1: *Fit results of the `dischargeFit` class using the fit methods `lsqcurvefit` and `fminsearch`, respectively. The raw data was extracted from [3].*

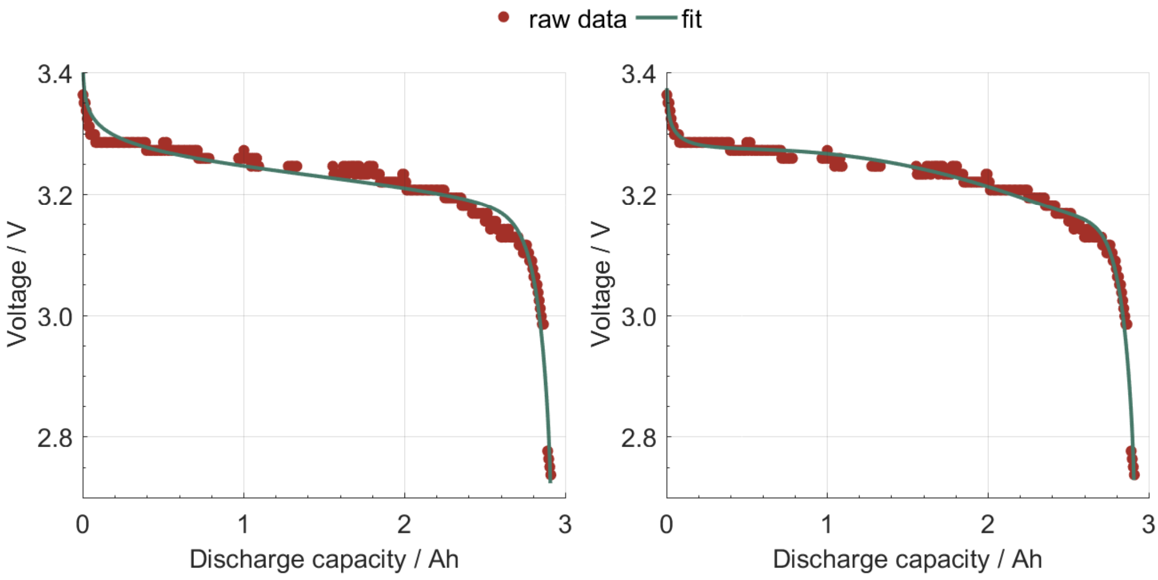

Figure 2: *Fit results of the `dischargeFit` class using the fit mode `'both'` with the default parameter initialization (left) and with a custom parameter initialization (right). The raw data was extracted from [3].*

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.

```
d = dischargeFit(V, C_dis, I, T, 'mode', 'fmin');
Cd = 1.5; % Discharge capacity in Ah
V = d(Cd); % Voltage in V
Cd_vect = linspace(0, 3, 1000); % Vector of discharge capacities in Ah
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 3.3) using it's `addcurves()` method.

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:

```
V = dC.interp(I, Cd);
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

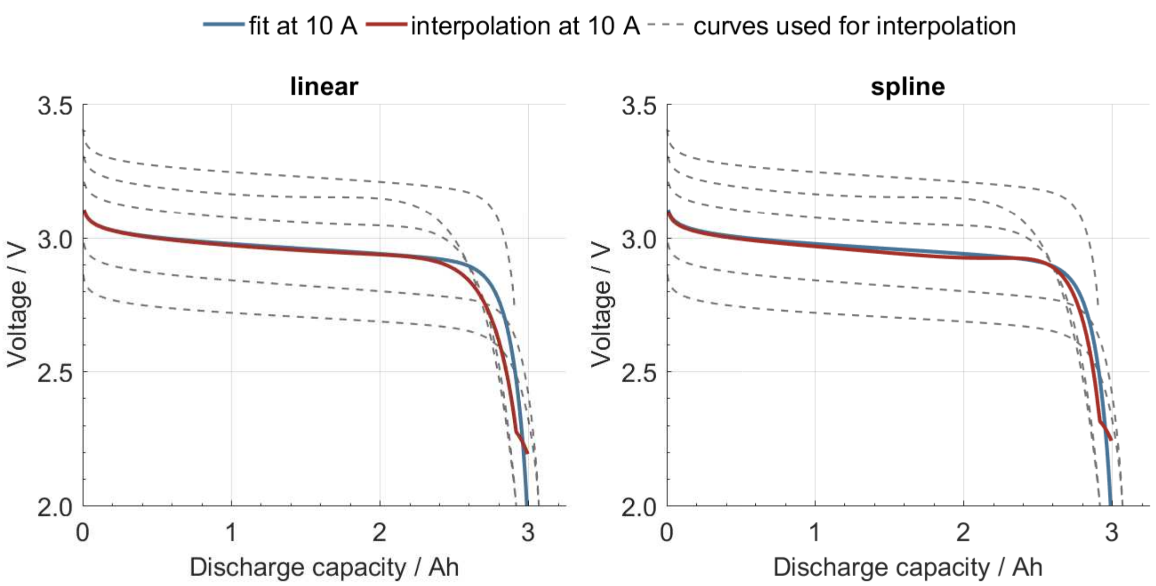

Figure 3: *Comparison of the `dischargeCurves` results using linear interpolation and spline interpolation, respectively. The raw data was extracted from [3].*

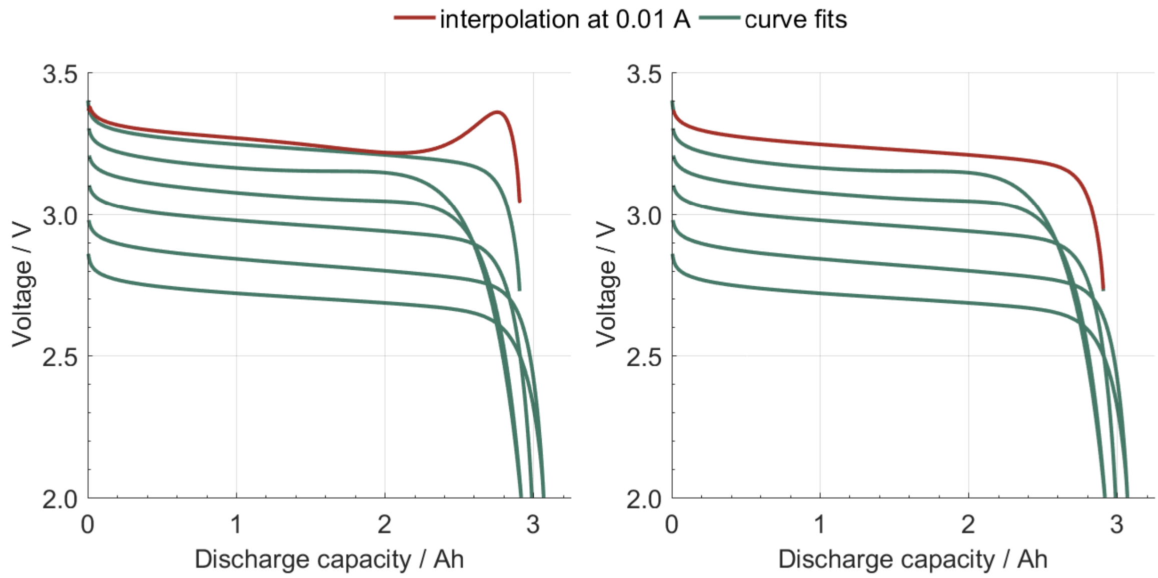

Figure 4: *Result of the* `interp()` *method for a current below the lowest measured current without output limitation (left) and with output limitation (right). The raw data was extracted from [3].*

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 3.3) using it's `addCurves()` method.

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)[ii]. 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

[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.
[ii] see section3.3.3.

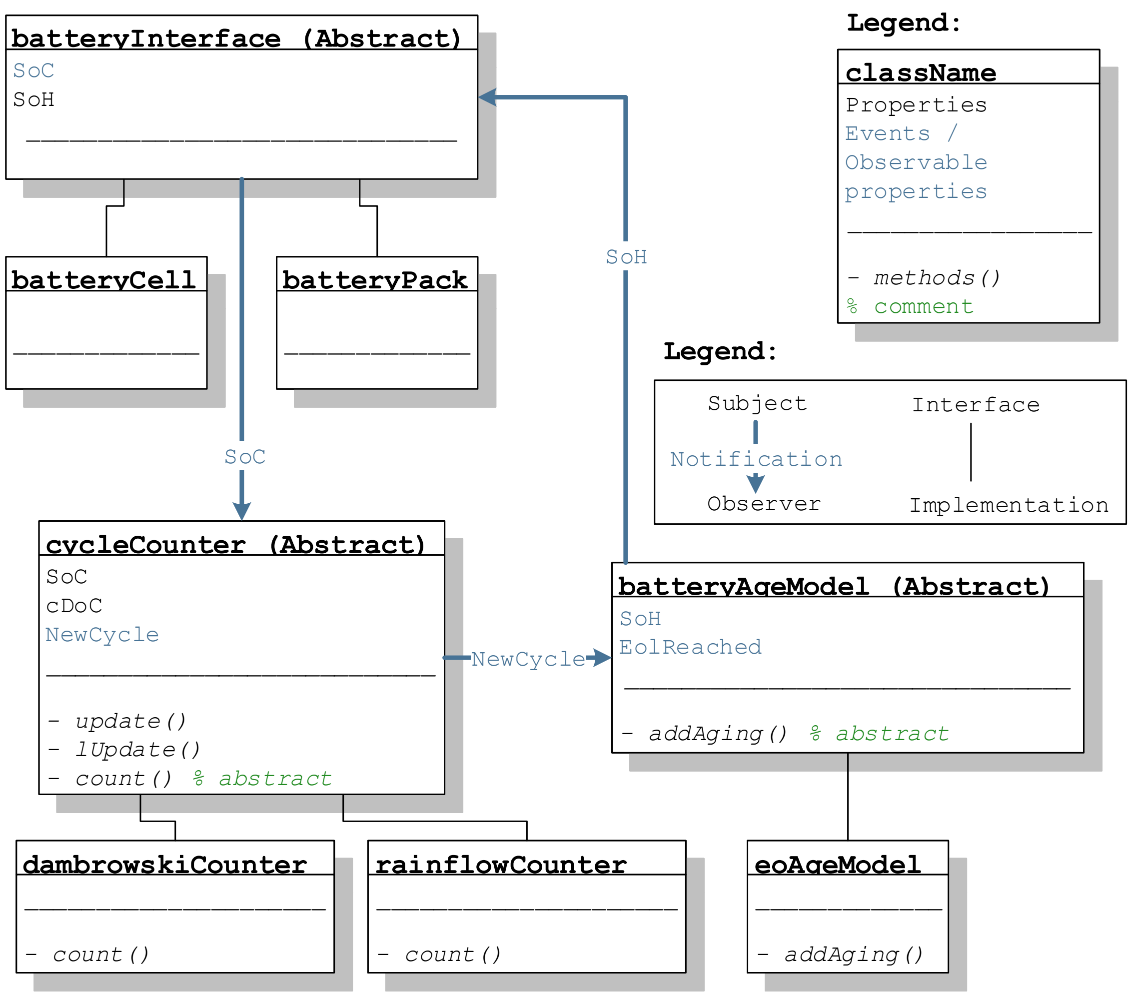

Figure 5: *Overview of the Observer implementation of the age model with communication flows and inheritance links.*

at runtime or by including a `dummyCycleCounter` and a `dummyAgeModel`. These classes 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:

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

```
% 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` (see section 3.3).

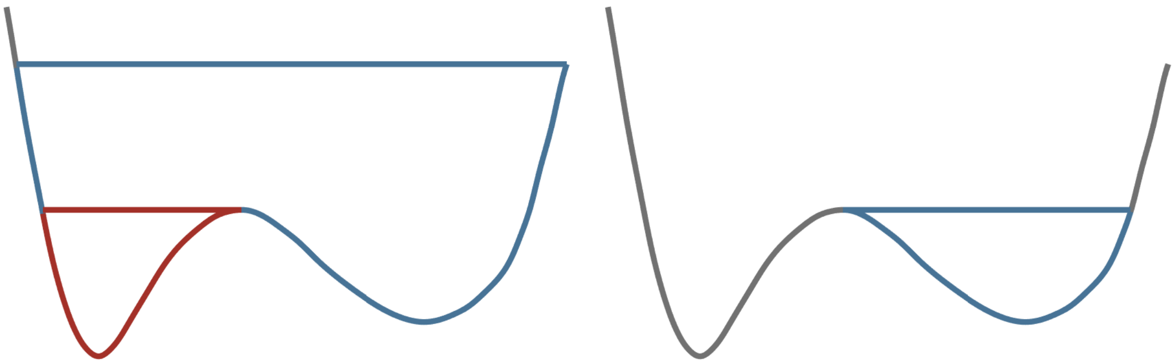

Figure 6: *Qualitative visualization of pre-cycle counting according to [7]: Two pre-cycles of prior equality (left) and a pre-cycle of subsequent equality (right).*

within an interval between the respective $SoC_{\text{max,l}}$ and the subsequent point at which the SoC is equal to $SoC_{\text{max,l}}$. 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.

```
bat.initAgeModel('ageModel', 'EO')
% Automatically initializes a dabrowskiCounter object with init_soc
% 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].

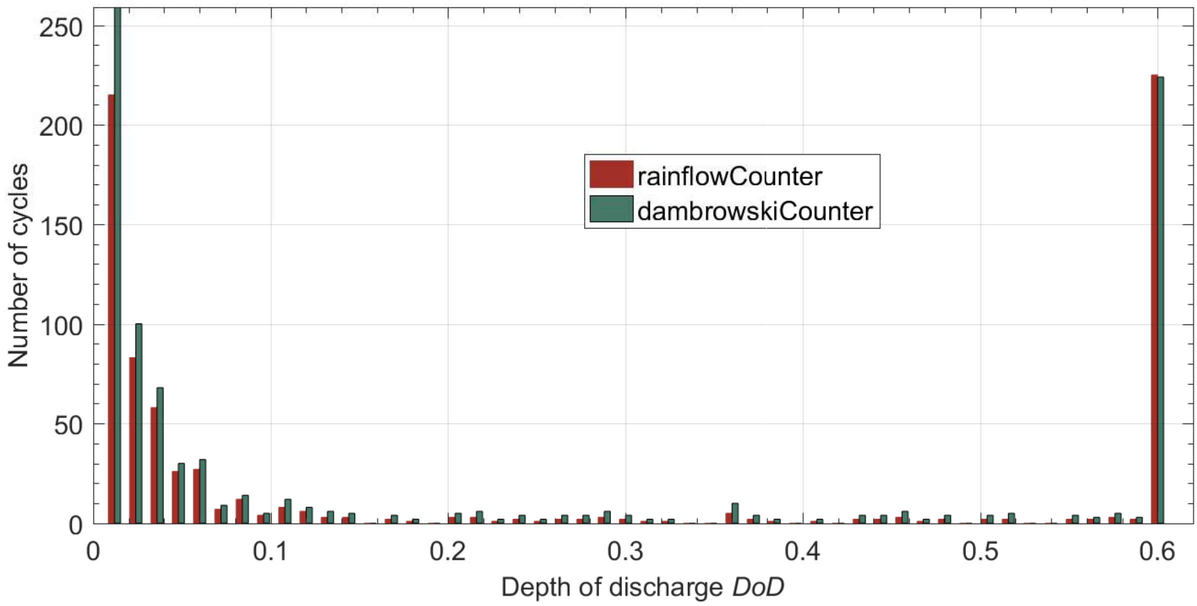

Figure 7: *Comparison of the counted cycles and their DoD between two simulations using the `rainflowCounter` and the `dambrowskiCounter` using the same SoC profile.*

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]

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

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

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

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

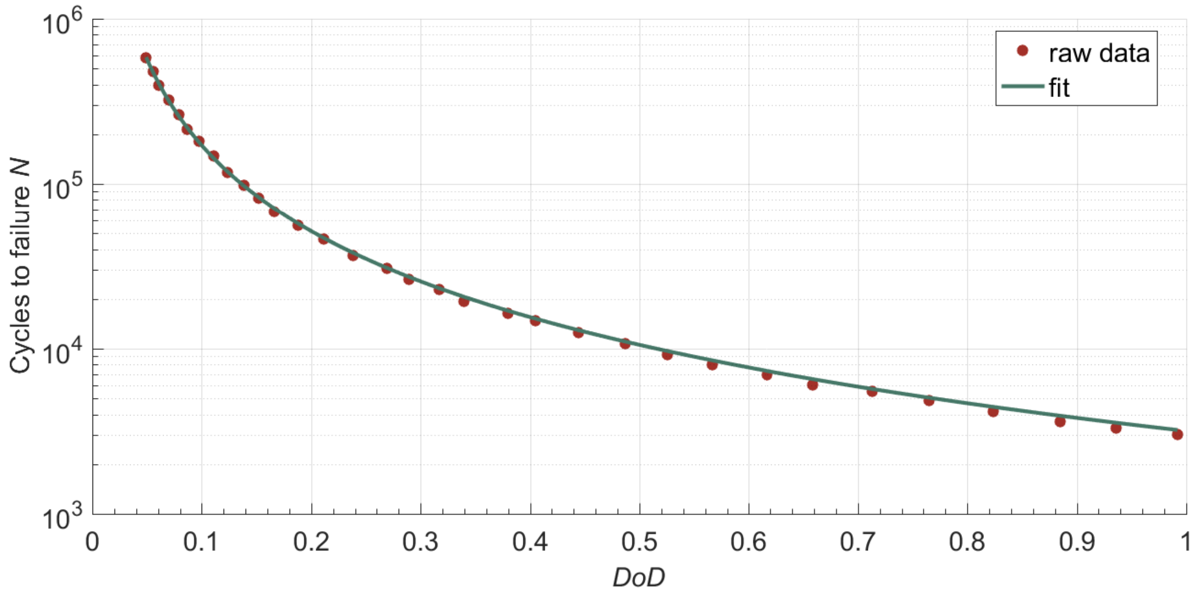

Figure 8: *Example for a lithium ion battery's N_f vs DoD cycle life curve fitted using the `woehlerFit` class.*

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.

A cycle life curve fit object is initialized with the raw data and the optional name-value pairs that the `dischargeFit` and every other subclass of the `curveFitInterface` accepts (see section 1.1.1). It can then be passed on to an age model via it's constructor or by setting it's `wFit` property.

```
% DoD = depth of discharge, N = number of cycles to failure
fit1 = woehlerFit(DoD, N);
% Plot results to new figure window
fit1.plotResults;
% Example using lsqcurvefit
fit2 = deFit(DoD, N, 'mode', 'lsq');
% Example using fminsearch and with custom initial params
fit3 = nrelcFit(DoD, N, 'mode', 'fmin', 'x0', [0.5; 1]);
% Pass cycle counter cy and fit1 to age model via it's constructor
am = eoAgeModel(cy, fit1);
% Replace fit1 with fit2 in age model
am.wFit = fit2;
```

Alternatively, a cycle life curve fit can be passed directly to a battery object `bat` via it's `addcurves()` method.

```
fit = woehlerFit(DoD, N);
bat.initAgeModel('ageModel', 'EO')
bat.addcurves(fit, 'cycleLife')
```

If the age model has not been initialized when the curve fit is added, it is stored for later use.

```
bat.addcurves(fit, 'cycleLife')
bat.initAgeModel('ageModel', 'EO') % fit is automatically passed
% to the age model
```

As well as curve fit objects, function handles to functions of one variable are accepted. However, anonymous functions are not recommended, due to their significant performance penalty.

```
fit = @(x)(3000 * x.^(-1.73));
bat.addcurves(fit, 'cycleLife')
```

2.3.2 Cycle ageing

A battery's age A_{c} is the opposite of the SoH, which is the usable capacity C_{bu} divided by the nominal capacity C_{n}.

$$A_{\mathsf{c}} = 1 - SoH = 1 - \frac{C_{\mathsf{bu}}}{C_{\mathsf{n}}} \tag{6}$$

In the `eoAgeModel` class, the ageing due to cycling stress A_{cyc} with n cycles and their respective DoD is determined from the curve fit $N_{\mathsf{f}}(DoD)$.

$$A_{\mathsf{cyc}} = \sum_{i=1}^{n} \frac{1}{N_{\mathsf{f}}(DoD_i)} \tag{7}$$

Every time a set of cycles is counted, A_{cyc} is determined and added to A_{c}.

2.3.3 Age model initialization

The default constructor syntax of a `batteryAgeModel` is as follows:

```
am = batteryAgeModel; % Replace batteryAgeModel with the subclass name
% eoAgeModel requires at least one input (a cycle counter cy)
am = eoAgeModel(cy);
am = eoAgeModel(cy, cfit); % adds a cycle life curve fit
% Specify the SoH at which the end of life is reached (default: 0.2)
am = eoAgeModel(cy, cfit, soh_eol);
am = eoAgeModel(cy, cfit, soh_eol, soc_ini); % Set the initial SoC
```

To initialize the age model directly from a battery `bat`, use it's `initAgeModel()` method.

```
bat.initAgeModel('ageModel', 'EO') % default eoAgeModel
```

2.3.4 Creating a user-defined age model

The `eoAgeModel` class does not take into account calendar ageing, which may be needed in some cases. It was left out of the default class because in many cases, a weighted degradation factor may be sufficient. The following source code snippet provides an example of how the `eoAgeModel` could be subclassed to extend it with a simple calendar ageing model. A property which holds the battery's calendar life is added and initialized in the constructor, taking into account the end of life age (typically 0.2). Finally, an `addCalAge()` method is added, that calculates calendar ageing as a linear function of the time step size. This method can be called from within the main simulation. To create a completely different age model (i.e. one that takes thermal influences into account), subclass the `batteryAgeModel` class instead.

```
classdef myCalendarAgeModel < lfpBattery.eoAgeModel
%MYCUSTOMAGEMODEL: An example for a user-defined age model.
%Combines the event oriented age model with a linear calendar age model.

    properties
        L_cal; % calendar life in s
    end

    methods
        function obj = myCalendarAgeModel(l, varargin)
            % l = calendar life in years
            % varargin = input args of eoAgeModel constructor
            %% call superclass constructor
            obj = obj@lfpBattery.eoAgeModel(varargin{:});
            obj.L_cal = l * 525600  / obj.eolAc; % set L_cal in
            % seconds taking end of life age into account
        end
        function addCalAge(obj, dt)
            % Adds to the battery's age using the simulation time
            % step size.
            % dt = simulation time step size in s
            % obj.Ac = 1 - obj.SoH
            obj.Ac = obj.Ac + dt / obj.L_cal; % increment age
        end
     end
end
```

A user-defined age model can be added to a battery `bat` using it's `initAgeModel()` method.

```
L_cal = 20; % calendar life in years
am = myCalendarAgeModel(L_cal, cy); % user-defined age model
bat.initAgeModel('ageModel', am)
```

The above is meant as a rough example for how a user-defined age model could be defined. However, it adds calendar ageing on top of the cycle stress every time the `addCalAge` method is called. This could lead to an unwanted acceleration of the simulated ageing process. A better solution is included in this package.

2.3.5 Calendar ageing

The `eoCalAgeModel` class is an extension of `eoAgemodel` that adds the possibility of calendar ageing. It is similar to the example in section 2.3.4. However, calendar ageing is not simply added on top of cycle stress, but set against it. An `eoCalAgeModel` object is created with the same input arguments as an `eoAgemodel` object, with the addition of the battery's calendar life L_{cal} in years as the first argument.

```
am = eoCalAgeModel(L_cal, cy, __);
```

As within the example in section 2.3.4, the calendar ageing A_{cal} is modelled as a linear function of the simulation time step size Δt_{s}

$$A_{\mathsf{cal}} = \frac{\Delta t_{\mathsf{s}} \cdot A_{\mathsf{c,eol}}}{L_{\mathsf{cal}}} \tag{8}$$

where $A_{\mathsf{c,eol}}$ is the age at which the end of life EOL is reached. The `addCalAge()` method must be called manually from within the main simulation at the end of each time step (after cycling the battery). The total stress for each simulation time step t_{s} is the maximum of cycle and calendar ageing.

$$A_{\mathsf{tot}}(t_{\mathsf{s}}) = \max(A_{\mathsf{cyc}}(t_{\mathsf{s}}), A_{\mathsf{cal}}(t_{\mathsf{s}})) \tag{9}$$

3 Battery Composition

The battery pack is modelled using a variation of the Composite design pattern with multiple composite classes[i]. This way, cells can be combined flexibly in various different topologies.

3.1 Overview

The `batteryInterface` is the abstract component that defines the interface for all objects in the composition. It is subclassed by all other battery elements. The `batteryCell` objects are the "leaves" and a composite can be one of the following classes:

- `parallelElement`: A set of components in parallel.
- `seriesElementAE`: A set of components in series with active equalization.
- `seriesElementPE`: A set of components in series with passive equalization.

Each component can either be another composite object or a leaf. Figure 9 provides a visual overview of the topologies that are possible using different compositions. Using this variation

[i]The basic Composite design pattern has one component interface, one composite class and one leaf class.

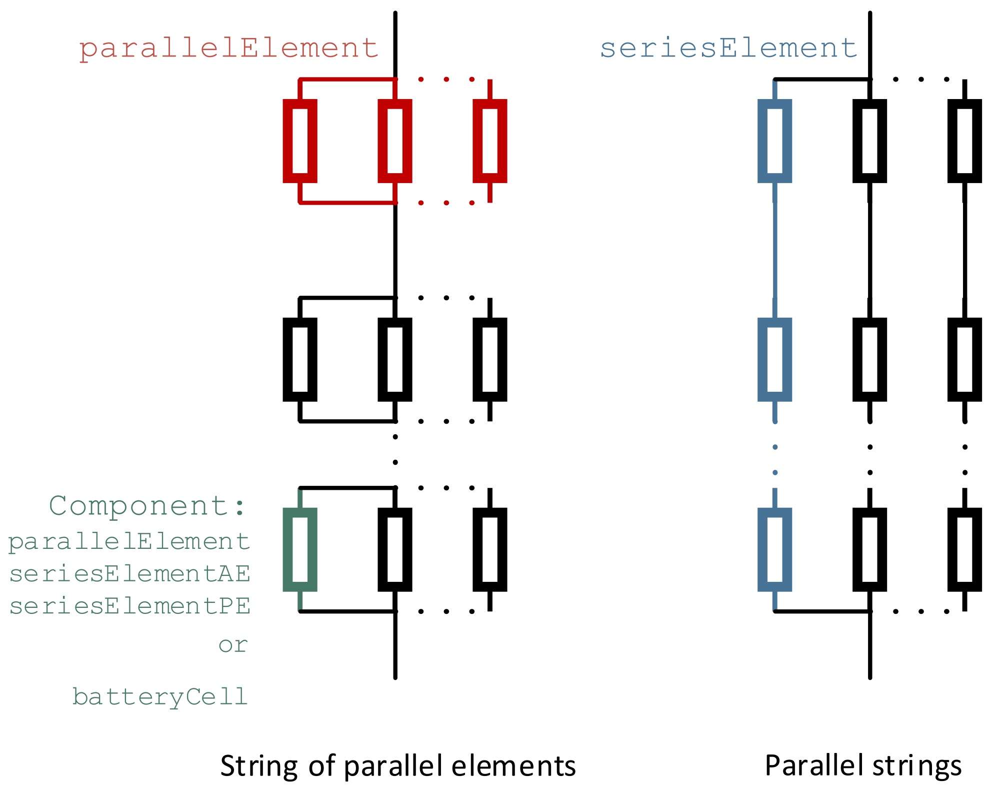

Figure 9: *Visualization of the possible battery topology compositions.*

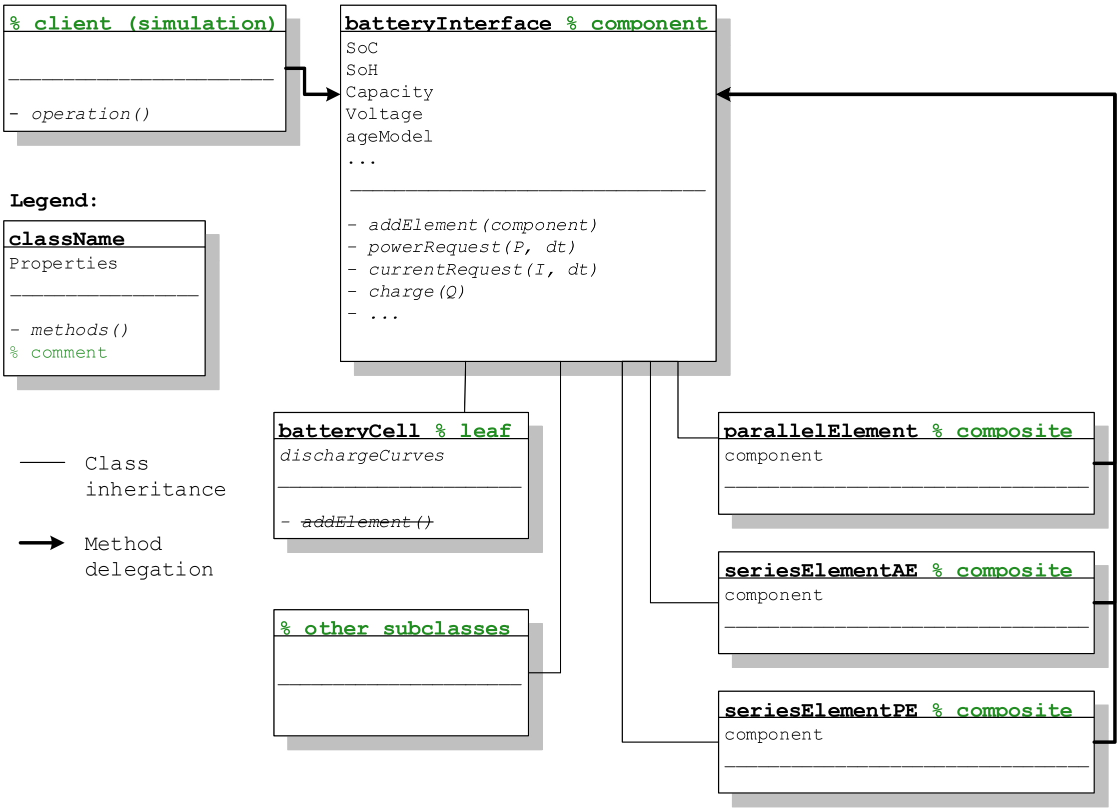

Figure 10: *Class diagram of the battery composition with communication flows and inheritance links.*

of the Composite design pattern, the components can be combined in any possible way at runtime. The most common battery topologies are strings of parallel elements (SP) and parallel strings of cells (PS) [14]. In Figure 9, these would be the case if the composition's leaf nodes (cells) were all in the second layer (marked green). However, since every component can be either a cell or another composite, more complicated topologies are made possible in this package.

3.2 Method delegation

A pattern diagram of the classes used for the topology composition is depicted in Figure 10. Every composite element holds a reference to a component and delegates the methods called on it to said component. The delegated methods are wrapped with the rules of the respective topology in a similar fashion as is done with the Decorator design pattern. An example of the method delegation for a PS configuration - a `parallelElement` that holds a set of `seriesElement` objects, each in turn holding a set of `batteryCell` objects - is visualized in Figure 11. In this example, a current `I` and the simulation time step size is passed to the `parallelElement` via a `getVoltage()` method. The `parallelElement` determines

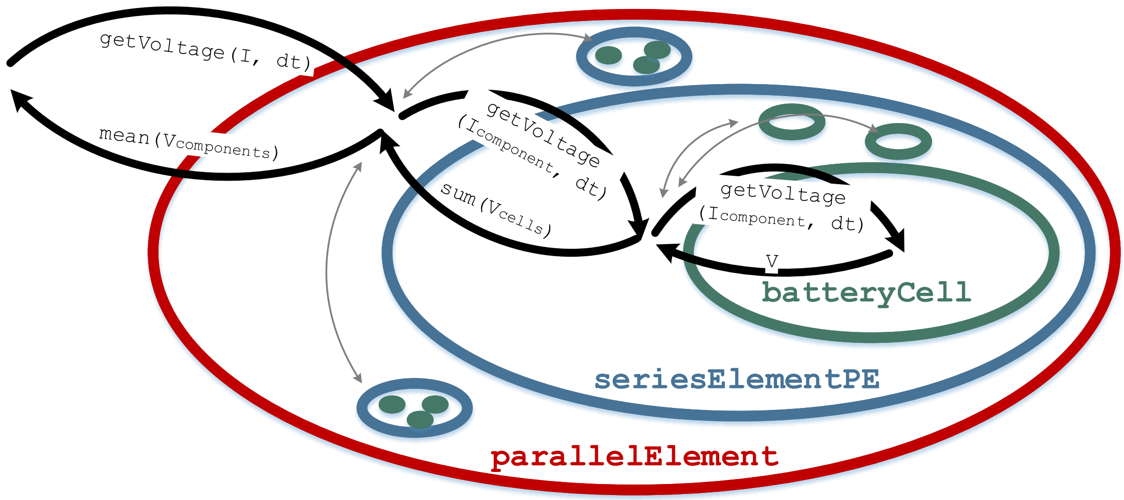

Figure 11: *Example of a method being delegated across a battery pack composition.*

which portion of `I` to send to each of it's components and delegates the method. Each `seriesElement` does the same and delegates the method to it's `batteryCell` objects. These return their voltages back to the `seriesElement` objects, which sum up the results received from their cells and pass the sum back to the `parallelElement`. Finally, the `parallelElement` calculates the mean of all the summed up voltages it received and passes the end-result back to the client. The following operations are delegated by an object that implements the `batteryInterface`:

- Determination of the new voltage after charging or discharging a with a certain current and time step size. This is delegated to each `batteryCell` object's `dischargeCurves` reference.
- Charging or discharging the battery.
- Determining the state of the battery if it were to be charged or discharged.
- Determining the maximum charging or discharging current.
- Calculating the pack's SoH.
- Getters and setters for the component's voltage and capacity properties.
- Getter for the component's internal impedance.

With the number of subcomponents n, a component's voltage is determined as

$$V_{\text{component}} = \begin{cases} \frac{\sum_{i=1}^{n} V_{\text{subcomponent},i}}{n} & \text{for a parallel element} \\ \sum_{i=1}^{n} V_{\text{subcomponent},i} & \text{for a series element} \end{cases} \tag{10}$$

And a component's capacity is

$$C_{\text{component}} = \begin{cases} \sum_{i=1}^{n} C_{\text{subcomponent},i} & \text{for a parallel element} \\ \min_{i=1}^{n} C_{\text{subcomponent},i} & \text{for a series element with passive equalization} \\ \frac{\sum_{i=1}^{n} C_{\text{subcomponent},i}}{n} & \text{for a series element with active equalization} \end{cases} \tag{11}$$

Since the SoH is derived directly from the capacity (see Equation 6), a component's SoH can be determined in the same fashion.

$$SoH_{\text{component}} = \begin{cases} \sum_{i=1}^{n} SoH_{\text{subcomponent},i} & \text{for a parallel element} \\ \min_{i=1}^{n} SoH_{\text{subcomponent},i} & \text{for a series element with passive equalization} \\ \frac{\sum_{i=1}^{n} SoH_{\text{subcomponent},i}}{n} & \text{for a series element with active equalization} \end{cases} \tag{12}$$

Due to the fact that the model is based on curve fits, the internal impedance Z_{i} property is not used for modelling the charging behaviour directly. It does however, determine how the voltages and currents are distributed across the subcomponents when charging or discharging. The impedance proportionality factor p_{z} of a subcomponent with index j is the component's Z_{i} divided by the sum of all subcomponents' Z_{i}.

$$p_{\text{z},j} = \frac{Z_{\text{i},j}}{\sum_{i=1}^{n} Z_{\text{i},i}} \tag{13}$$

When charging, a series element with active equalization will distribute it's voltage equally across all of it's subcomponents to account for balancing, while a series element with passive equalization will distribute it's voltage according to $p_{\text{z},j}$. For a parallel element, the current is distributed in such a way that the subcomponent j with the lowest Z_{i} receives the highest current.

$$I_{\text{subcomponent},j} = \frac{\frac{1}{p_{\text{z},j}}}{\sum_{i=1}^{n} \frac{1}{p_{\text{z},i}}} \cdot I_{\text{component}} \tag{14}$$

3.3 Battery Interface

The battery interface is described in the following subsections. Every component implements the `batteryInterface`, so the methods described in this section can be called on `batteryCell` objects and on the composites.

3.3.1 Battery object initialization

To initialize a battery object at runtime, the nominal capacity C_{n} in Ah and the nominal voltage V_{n} in V must be passed to a `batteryCell` constructor. A composite can be initialized as an "empty" circuit element and the cell (or other composites) can be added to it via it's `addElements()` method[i].

[i] Here, "empty" is referred to in the sense of not holding any cells, not in the sense of an empty Matlab® variable.

```
% Initialize an "empty" parallel element
bat = parallelElement;
Cn = 3; % Nominal cell capacity in Ah
Vn = 3.2; % Nominal cell voltage in V
% Initialize 3 battery cells and add them to bat
for i = 1:3
    b = batteryCell(Cn, Vn);
    bat.addElements(b);
end
```

The `addElements()` method also accepts component arrays...

```
for i = 1:3
    b(i) = batteryCell(Cn, Vn);
end
bat.addElements(b);
```

...and multiple inputs:

```
b1 = batteryCell(Cn, Vn);
b2 = batteryCell(Cn, Vn);
b3 = batteryCell(Cn, Vn);
bat.addElements(b1, b2, b3);
```

To create a composition like the example in Figure 11 (see also Figure 9, right), the following syntax could be used:

```
% Initialize "empty" parallel element
bat = parallelElement;
% Initialize 3 "empty" series elements each holding 3 cells
for i = 1:3
    se = seriesElementPE; % passive equalization
    for j = 1:3
        se.addElements(batteryCell(Cn, Vn))
    end
    % Add series elements to bat
    bat.addElements(se)
end
% Further initialization operations, e.g. bat.addcurves() here...
```

3.3.2 Battery charging and discharging

Battery charging [i] is handled by the methods `powerRequest()` and `currentRequest()`. Both functions are called in a similar manner. The syntax is as follows:

```
[P, V, I] = bat.powerRequest(P, dt);
[P, V, I] = powerRequest(bat, P, dt); % equivalent
[P, V, I] = bat.currentRequest(I, dt);
[P, V, I] = currentRequest(b, I, dt); % equivalent
```

Where `P` is the requested power P in W, `I` is the requested current I in A and `dt` is the simulation time step size Δt_s in s. The methods return the actual power throughput in W, the battery's voltage V at the end of the time step and the actual current throughput in A. The returned power and current is limited by the SoC or the cells' maximum currents, among other factors. Figure 12 contains a flow chart of the charging process. The client sends a request to the battery. If the requested power is not equal to zero and the battery's SoC is not already at it's upper or lower limit, a charge iteration is performed (the `iteratePower()` and

[i]Discharging will also be referred to as charging (with a negative current) in this documentation.

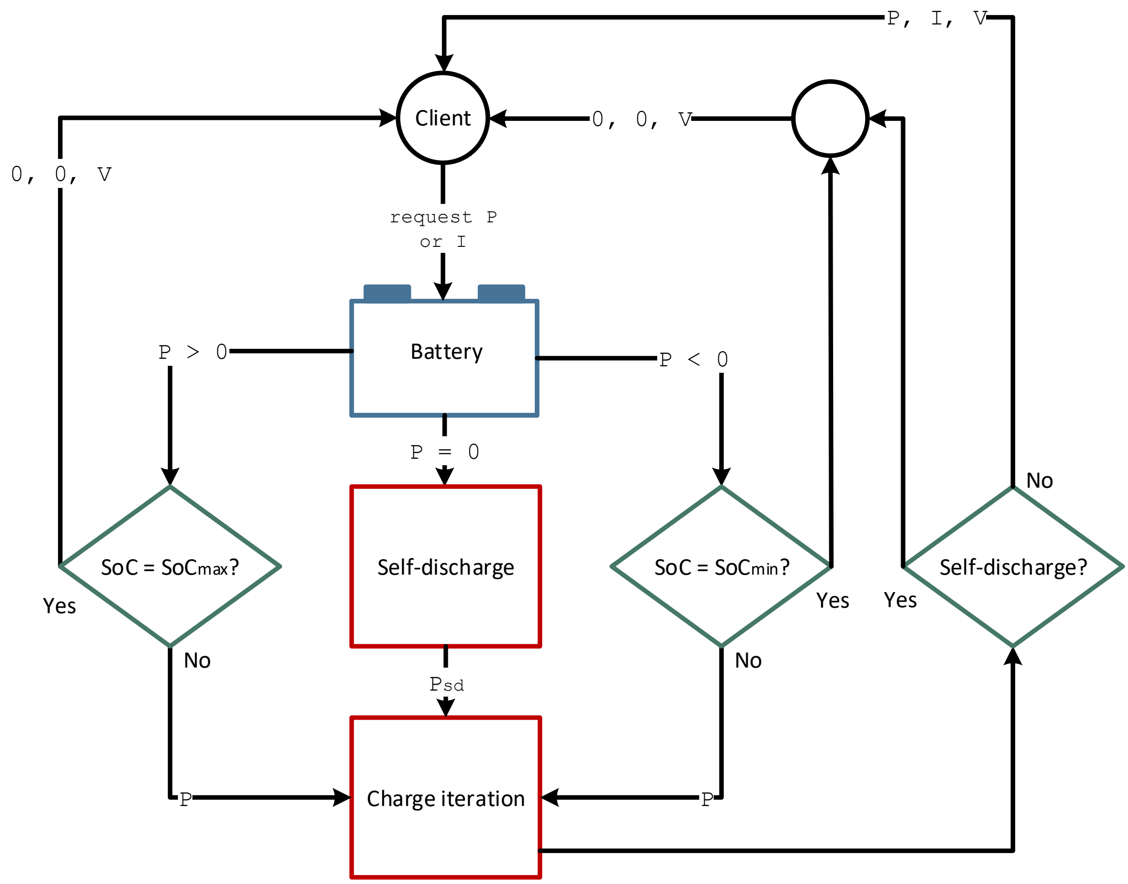

Figure 12: *Flow chart of the* `powerRequest()` *and* `currentRequest()` *methods.*

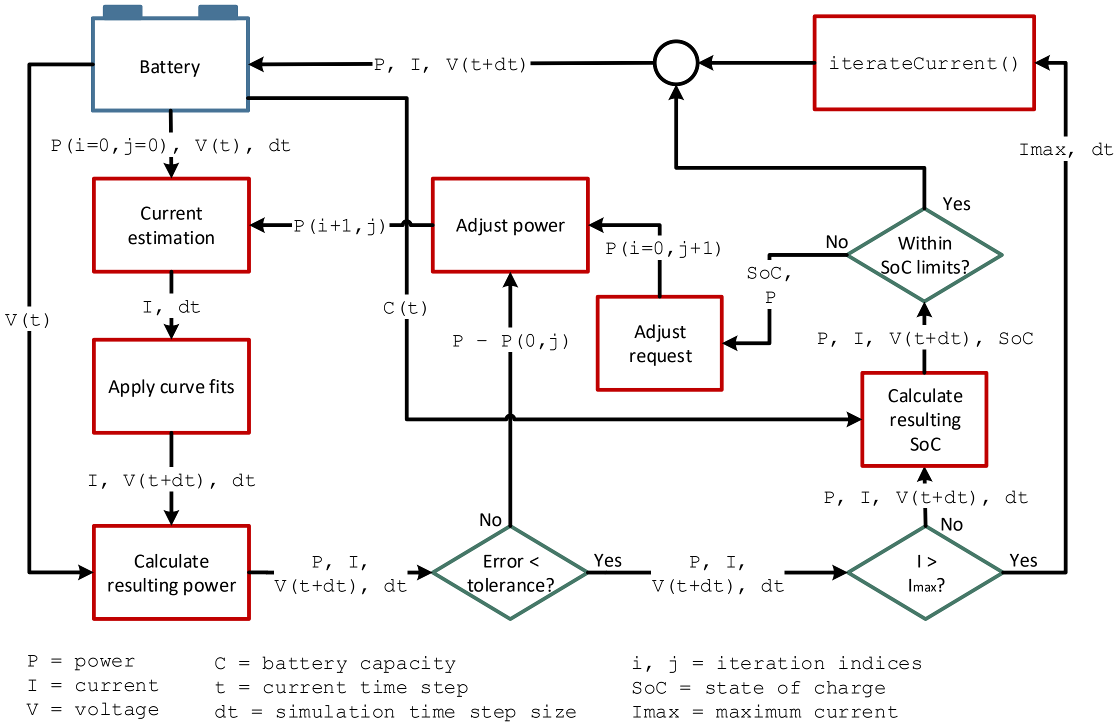

Figure 13: *Flow chart of the `iteratePower()` method.*

`iterateCurrent()` methods are called, respectively) and the resulting power, current and voltage are returned to the client. A positive input to the charge iteration indicates charging and a negative input specifies discharging. If the request is zero, signalling that the battery is in an idle state, a logical flag is set to true and the charge iteration is called with the battery's self-discharge P_{sd}. The logical flag is checked after every call to the charge iteration methods in order to return a power and current of zero to the client if it was set to true. If the SoC is either at it's upper or lower limit, the battery simply returns it's voltage along with a power and current of zero.

A flow chart of the `iteratePower()` method is depicted in Figure 13. First, a current is estimated from the requested power and the battery's voltage. The current and the time step size are then delegated to the battery cells' `dischargeCurves` objects, in order to determine the resulting voltage. An approximation of the power is determined from the mean of the returned voltage and the battery's old voltage. This is repeated through recursion until the difference between the iterated power and the originally requested power meets a certain tolerance. If the resulting current is greater than the battery's maximum current I_{max}, the `iterateCurrent()` method is called using I_{max} as an input. It's output current, the resulting power and voltage are returned. Otherwise, the SoC is determined and compared the battery's upper and lower limit. If the SoC is within the interval $[SoC_{\text{min}}, SoC_{\text{max}}]$, the power, current and voltage are returned. Otherwise, the requested power is adjusted according to the difference between the SoC and the respective limit that was exceeded, thus starting the iteration again.

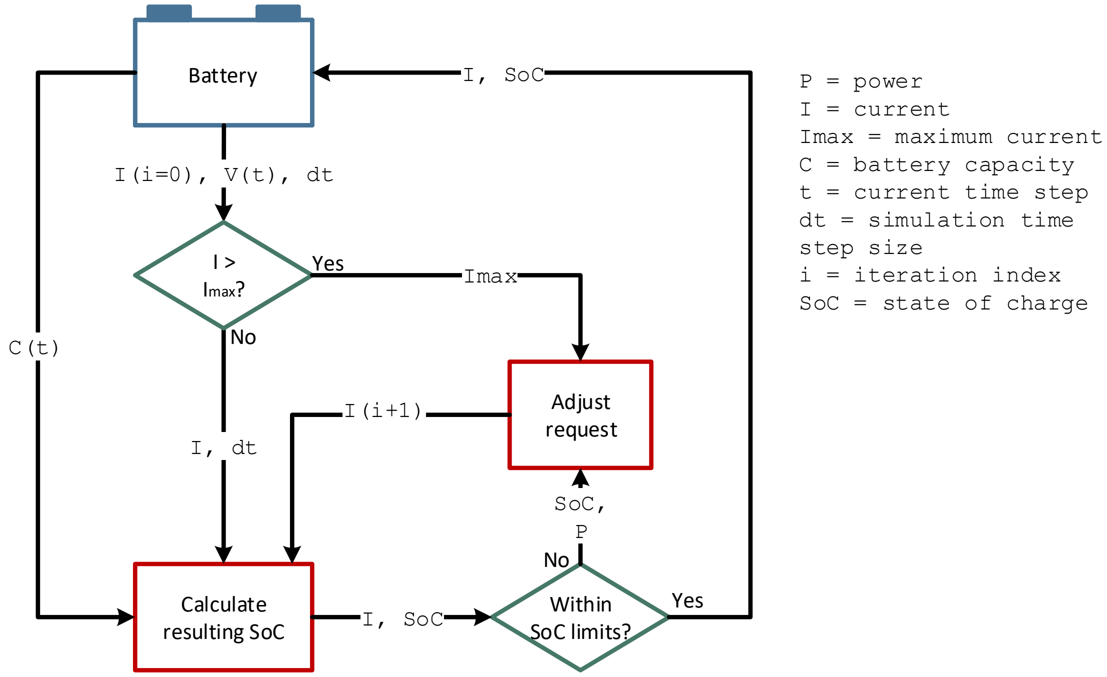

Figure 14: *Flow chart of the `iterateCurrent()` method.*

Figure 14 depicts a flow chart of the `iterateCurrent()` function. Using this method is a lot faster than using the `iteratePower()` function, due to it's comparative simplicity. However, the current may need to be determined separately in some cases. Before the iteration, the current is limited to I_{max}. Finally, the another limitation is performed if the SoC is not within the interval $[SoC_{\text{min}}, SoC_{\text{max}}]$. Normally, one or two iterations should suffice for returning the current and SoC. The voltage and power are not calculated and must be determined by calling the `getNewVoltage()` method if required[i].

The results of three charging simulations of a battery pack are depicted in Figure 15. Since the data sheet [3] that was used does not contain any voltage curves for charging currents, the discharge curves were used for charging, too[ii]. Every time, the empty pack was charged until an SoC of approx. 0.9 was reached. In the first simulation (on the left hand side), the battery was charged with a constant current I_{max}. The resulting voltage is an interpolation between two curve fits and appears to have perfect results upon first glance. For the second simulation (Figure 15, center), the current was linearly increased between 0 and I_{max}. The resulting voltage is a curve that interpolates all of the curve fits at different capacities. Here, the problems of using discharge curves for charging become apparent. The voltage is higher than in the first simulation - especially at lower currents. This is typical behaviour for discharging. However, a lower charging current should result in lower voltages. Due to this problem, it is advisable to add separate charging and discharging curve fits to the model. This can be done

[i]For example, this is done within the `currentRequest()` method, which does return the voltage and power.

[ii]This is the default behaviour if no charge curves are added.

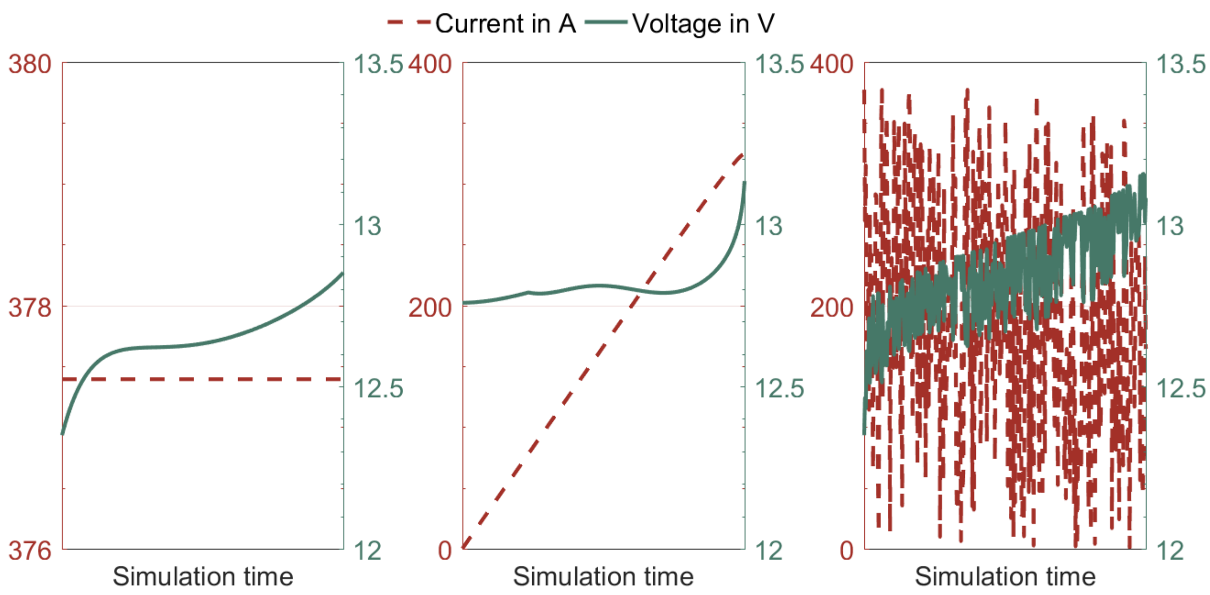

Figure 15: *Comparison of battery pack charging simulations using a constant current, a linearly increasing current and a random current, respectively - Voltage vs. simulation time. The pack's cells were modelled according to [3].*

via the `addcurves()` method:

```
bat.addcurves(chargeCurvefitObj, 'charge')}
bat.addcurves(dischargeCurvefitObj, 'discharge')}
```

A charge curve fit can be fitted using the `dischargeFit` class or the `dischargeFit()` method (see section 1.1) or a user-defined class that implements the `curveFitInterface`. A random distribution of currents within the interval $[0, I_{\text{max}}]$ was used for the third simulation (Figure 15, right). The main issue of this model's approach using curve fits is emphasised here. Voltage leaps occur if the current changes drastically from one time step to another. It is highly doubtful that a battery would behave like this in reality, since the discharge curves are actually measured by discharging with a certain current and then waiting for long periods of time (e.g. up to four hours) until the resting voltage stabilizes before taking measurements [15]. A possible solution in a simulation that charges and discharges with strongly fluctuating currents could be to smooth the returned voltages out with a running mean or to use a customized version of the `dischargeCurves` class (see section 1.2) that always returns the mean of all currents' voltages as a function of the SoC. The `mdischargeCurves` class was added to this package for that purpose. As is shown in Figure 16, plotting the voltage against the SoC causes the curves to follow more similar functions than when plotting them against the current. By using the `mdischargeCurves` class instead of the `dischargeCurves` class, the volatile curve with random currents can be flattened out to produce the results in Figure 17. It should be noted, however, that the mean charging and discharging currents of the simulation must be known so that the relevant curve fits can be added accordingly.

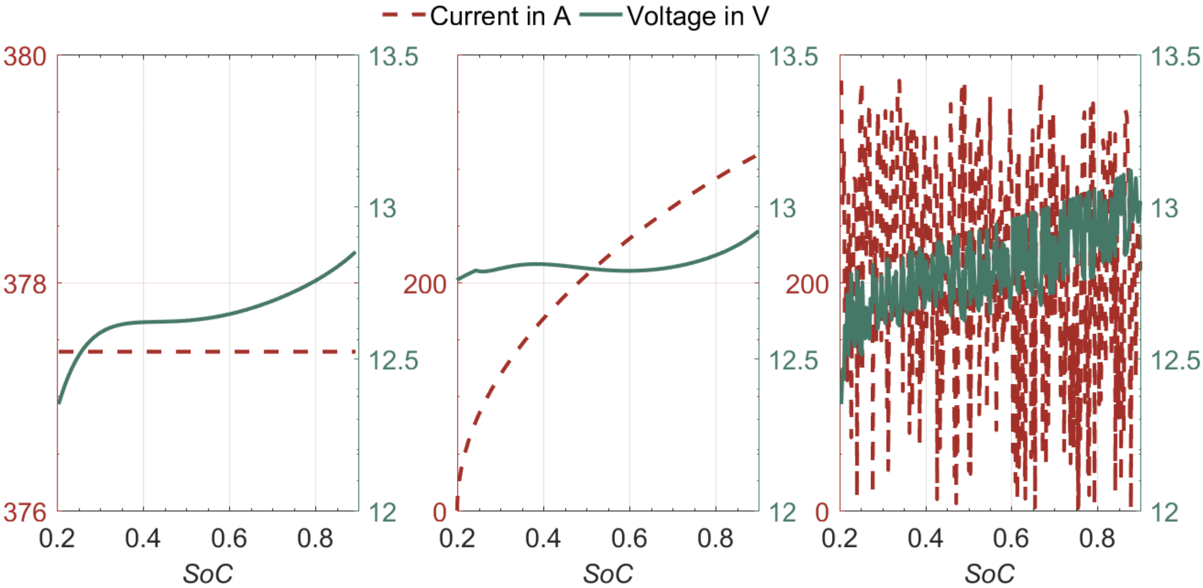

Figure 16: *Comparison of battery pack charging simulations using a constant current, a linearly increasing current and a random current, respectively - Voltage vs. SoC. The pack's cells were modelled according to [3].*

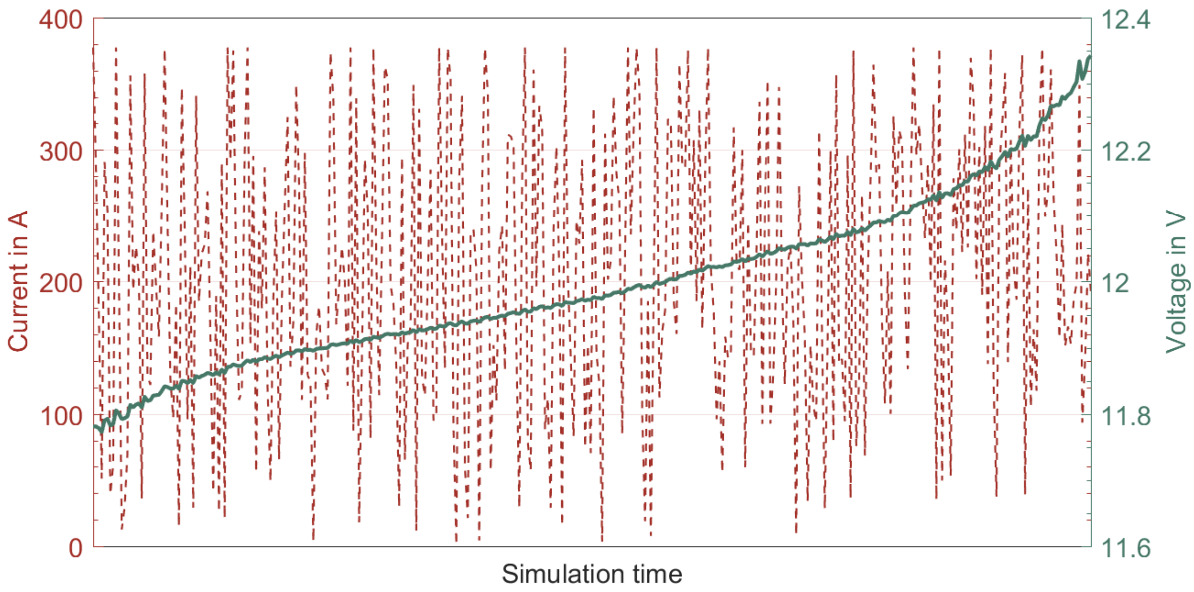

Figure 17: *Simulation of battery charging with random currents using the* `mdischargeCurves` *class for voltage calculation. The resulting voltage is a function of the battery's SoC.*

An `mdischargeCurves` object shares the exact same interface as a `dischargeCurves` object. To convert the classes into each other, use the `add()` method.

```
d2 = mdischargeCurves;
d2.add(d); % Adds all of the dischargeCurves' dischargeFit objects
           % to the mdischargeCurves object
% Works the other way around, too
d = dischargeCurves;
d.add(d2)
```

3.3.3 Age model level

The age model (see section 2) can be left out completely, added on the pack level or added on the cell level. Adding it on the pack level is done by calling `initAgeModel()`[i] on the outermost wrapper object, e.g. a `batteryPack` (see section), a `parallelElement`, a `seriesElementPE`, etc. Doing so will cause the battery pack's total SoC to be observed for cycle counting and the pack's SoH to be updated by the `batteryAgeModel` object. If the cells have varying properties (i.e. different internal impedances), their individual cycles may vary and it could make sense to add an age model to each cell individually. This can be done by calling `initAgeModel()` on every cell. The cells can be extracted using the `createIterator()` method, which returns a `batteryIterator` object that can be used to iterate through the cells. To indicate that the age model is set to the cell, level, the main battery pack object (outermost wrapper) must have it's age model set to `'LowerLevel'`, or the SoH will not be calculated correctly. The following code provides an example of extracting a battery pack `bat`'s cells and setting the age model.

ref section

```
it = bat.createIterator;
while it.hasNext % Iterate through cells
    b = it.next; % batteryCell object
    b.initAgeModel('ageModel', 'EO')
end
% Make sure 'LowerLevel' option is set on outermost wrapper
bat.initAgeModel('LowerLevel')
```

3.4 CCCV charging and BMS

Typically, a lithium-ion battery is charged using a constant current / constant voltage (CCCV) charging strategy. A qualitative example of the CCCV strategy is depicted in Figure 18. In the CC phase, a constant charging current causes the SoC to increase linearly over time while the voltage rises according to the respective charging curve (see section 1.2). When the voltage

[i] The `initAgeMoel()` method is described in sections 2.2 - 2.3.

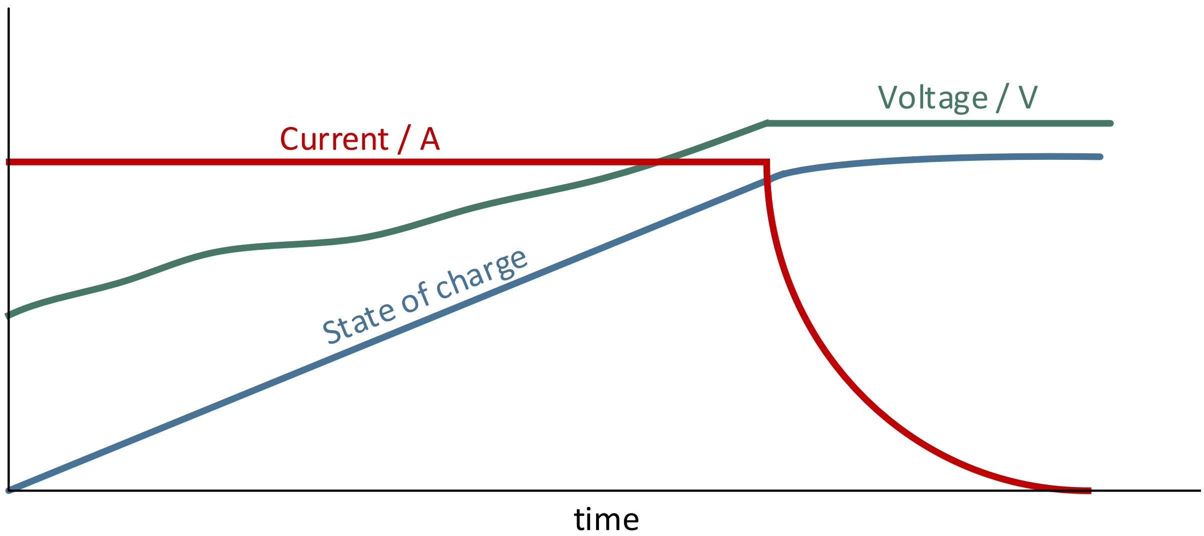

Figure 18: *Qualitative example of a CCCV charging curve.*

reaches a certain threshold, the current is reduced in order to stabilize the voltage during the CV phase. As a result, the SoC is no longer a linear function of time and increases at a slower pace. In practice, this charging strategy works well for individual cells and cells connected in parallel, since the voltage is distributed evenly across all cells. However, this is not the case for cells connected in series. A CCCV charger may limit the total voltage, but it has no knowledge of the distribution across the individual cells, which could result in the cells with the highest voltage getting damaged [16]. To prevent this, a battery management system (BMS) is required. The BMS monitors each cell individually and communicates with the charger. In the case of active equalization, the BMS rebalances the charge and voltages among the cells. This can be modelled by using the `seriesElementAE` class (see section 3.1).

3.4.1 CCCV curve fits

Due to the fact that simulations using variable currents are possible with this model, the charge limitation in the CV phase is done by limiting the maximum charging current I_{max} as a function of the SoC. Thus, lower currents than I_{max} are also possible during the CC phase. By default, I_{max} is limited according to the discharge curve with the largest current stored in the cell's `dischargeCurves` reference (see section 3.3.2). If a set of charge curve fits is added without adding a CCCV curve (using the `'charge'` option of the `addcurves()` method), I_{max} is set according to the charge curve with the greatest current. Finally, if a `cccvFit` object is added, I_{max} is dynamically set as a linear function of the SoC according to the CCCV curve fit.

$$I_{\text{max}}(SoC) = \frac{I_{\text{max,CC}}}{1 - SoC_{\text{CC/CV}}} \cdot (1 - SoC) \tag{15}$$

where $I_{\text{max,CC}}$ is the maximum current during the CC phase and $SoC_{\text{CC/CV}}$ is the SoC at the end of the CC phase. To add a CCCV curve fit to a battery object `bat`, the `addcurvers()` method can be called with the `'cccv'` option. The `cccvFit` class im-

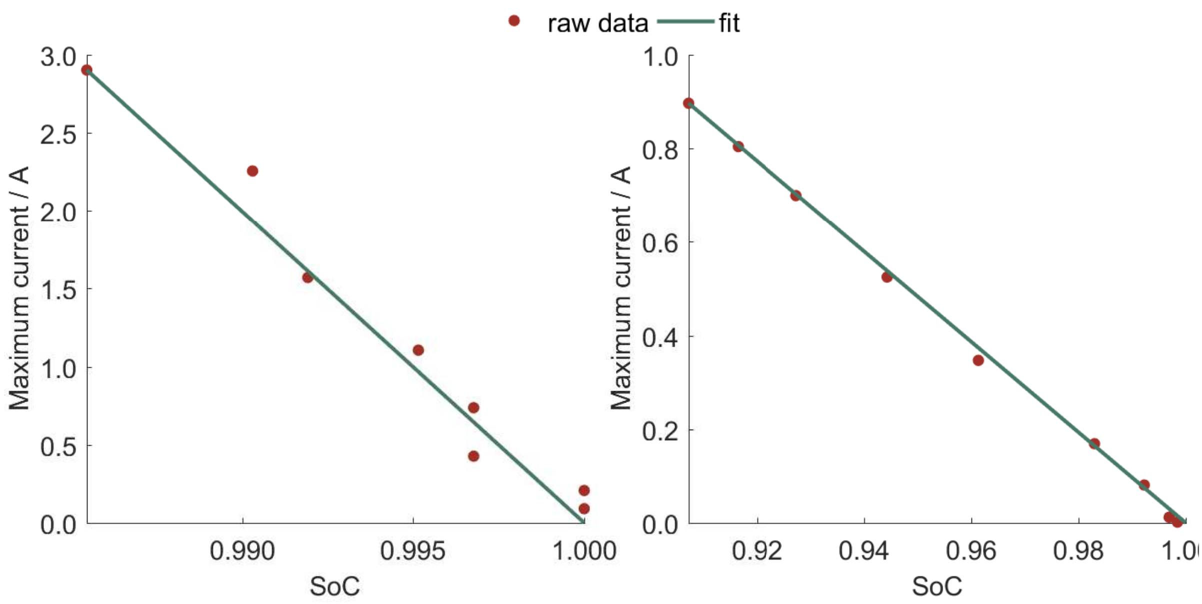

Figure 19: *Linear curve fits of $I_{\text{max}}(SoC)$ during the CV phase for two different battery cells.*

plements the `curveFitInterface` and is initialized with the raw data (SoC and I_{max} with $I_{\text{max}} = f(SoC)$) and the optional name-value pairs described in section 1.1. If for some reason the curve's maximum SoC is not 1, it should be passed to the constructor via a third argument SoC_{max}.

```
% CCCV curve fit
c = cccvFit(soc, iMax);
c2 = cccvFit(soc, iMax, socMax); % with SoC limitation
bat.addcurves(c, 'cccv') % add c to battery object
```

Two curve fits for two different battery cells' CV phases are depicted in Figure 19. For the cell fitted on the right hand side, the linear model provides a very good approximation. However, for cells with short CV phases resulting in small SoC gradients (fitted on the left hand side), the digitized raw data can appear more noisy due to the difficulty in digitizing the image.

3.4.2 Cell monitoring and communication with the charger

In this model, the cell monitoring and communication with the charger are implemented using the Observer design pattern. The cells take on the role of the subjects and are automatically registered with the pack they are added to via the `addElements()` method (see section 3.3). The communication flows between the BMS and the charger are visualized in Figure 20, with each communication path colour coded. Each battery composite object (e.g. a battery pack) contains a logical flag that indicates whether CV charging is active (true) or not (false). If a cell's individual SoC reaches the threshold at which CV charging is activated, the battery pack is notified, causing it's flag to be set true. This notification is also sent out if a cell that was in the CV phase moves back into the CC phase by being discharged. At the beginning

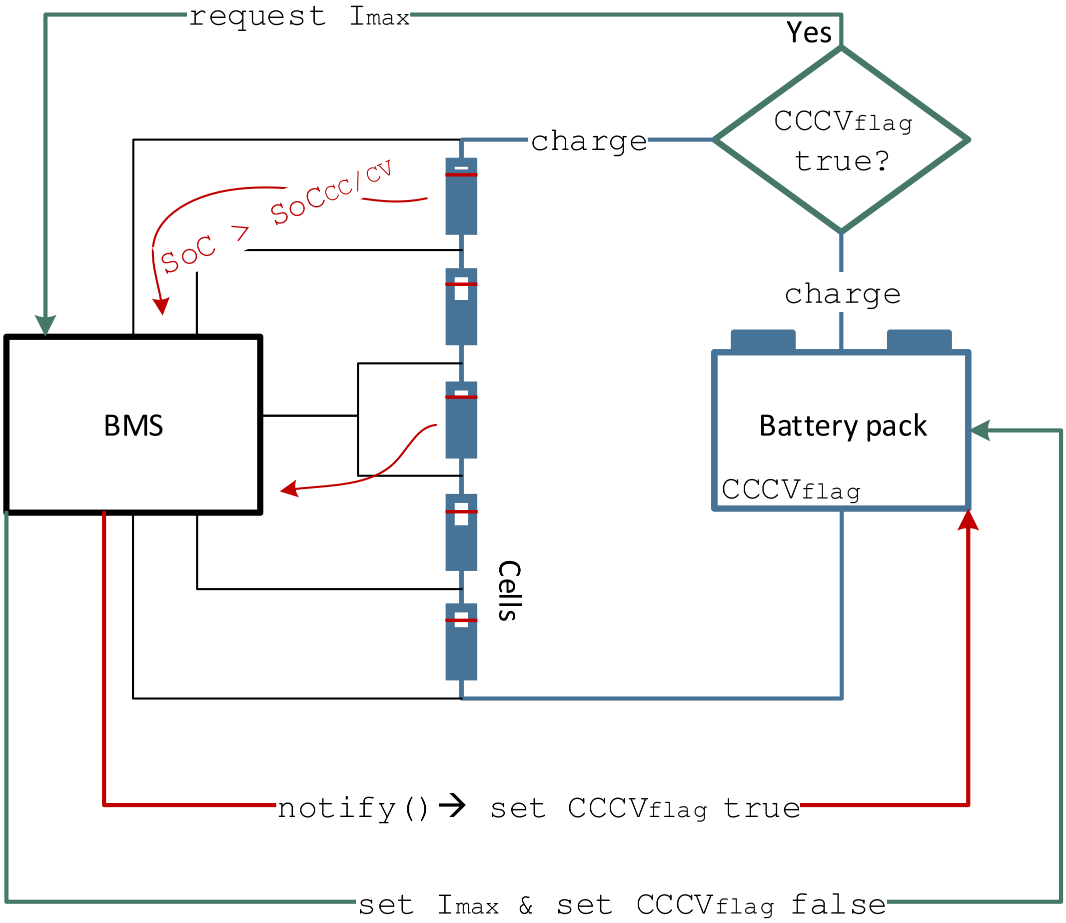

Figure 20: *Schematic visualization of the BMS combined with CCCV charging.*

of each time step, the maximum charging current is recalculated if the battery pack's CCCV flag is true. This also sets it false again to prevent unnecessary recalculations. If the flag is false, charging continues with the last cached I_{max}. Using this method significantly reduces simulation time compared to directly triggering a recalculation of I_{max} every time a cell's SoC threshold is reached.

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.

[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.

[14] A. Cordoba-Arenas, S. Onori, and G. Rizzoni, "A control-oriented lithium-ion battery pack model for plug-in hybrid electric vehicle cycle-life studies and system design with consideration of health management," *Journal of Power Sources*, vol. 279, pp. 791–808, Apr. 2015, ISSN: 03787753. DOI: 10.1016/j.jpowsour.2014.12.048.

[15] P. Keil and A. Jossen, "Aufbau und parametrierung von batteriemodellen," *19. DESIGN & ELEKTRONIK-Entwicklerforum Batterien & Ladekonzepte*, 2012.

[16] D. Andrea, *Battery management systems for large lithium-ion battery packs*. Boston: Artech House, 2010, 290 pp., ISBN: 978-1-60807-104-3.