Overview of the code

Goal of Slide

The main goal of Slide (Simulator of Lithium-Ion Degradation) is to allow fast simulation of degradation of li-ion batteries. Simulating 5000 full 1C cycles with a time resolution of 2 seconds (1C charge, 1C discharge) for one cell takes about 40 seconds. Including a CV phase on charge to a current limit of 0.05C increases the calculation time to 85 seconds.

The project uses the single particle model to simulate the main reaction in the battery. Various physics-based degradation models from literature have been implemented, and the user can select which degradation models to use. The parameters of the battery model, and of the different degradation models can be changed by the user.

Some degradation procedures have already been implemented, such as calendar ageing (resting a cell for a long time), cyclic ageing (continuously charging and discharging) and drive cycle ageing (continuously repeating a given current profile). The settings of these basic procedures can be easily changed (e.g. the user can choose whether to do a CC or CCCV charge and the voltage window to be used).
The user can program his/her own degradation procedures similarly to the way a battery tester is programmed. The code has functions implemented to do a CC or CV (dis)charge (or a combination of both) where the parameters such as the current, or the set voltage can be determined by the user. By calling these functions with different parameters, the user can specify his/her own degradation experiment. Similarly, functions have been implemented to measure the (remaining) cell capacity, the OCV curves, do a pulse discharge test, etc., which the user can call do to regular check-ups during the degradation simulation.

The C++ code writes its results in csv files. The results of the check-ups during the degradation experiments (e.g. capacity measurements) are written in separate csv files. When a cell is being cycled, the user can choose to record the cell current, voltage and temperature at a constant time interval (e.g. 5 seconds) in which case this data is also stored in csv files.
Matlab functions to read these results have been implemented as well. E.g. to read the results of the pre-defined 'calendar ageing' function, the user has to run 'readCalendarAgeing.m', which will plot the outcomes.

Speed of calculation & data collection

The main advantage this code is that it fast and relatively flexible to use. The calculation speed depends on what you are simulating. Below are a few tips.

- Calculating a CV takes much longer than a CC. As reference, calculating 100 1C cycles with only CC takes about 0.9 second while calculating 100 1C cycles with also a CV phase on charge (with a current limit of 0.05C) takes 1.9 second. This depends of course on how long the CV phase takes (a lower diffusion constant will give a longer CV and therefore increase calculation time).
- The code can record periodic values of current, voltage and temperature. This slows down the code and creates a lot of data, especially when you simulate long term degradation. Writing the data to csv files takes long, with the exact time depending on the speed of your hard disk. As a reference, the same 100 1C cycles (CC only) as before but with a 5s time recording interval takes 18 seconds, with the vast majority of the extra time spent on writing 17MB of data to the hard disk. Simulating the 100 cycles with an additional CV charge and 5s data collection takes 22 seconds and 21MB is written. When simulating degradation experiments, gigabytes

of data will be generated and it can take up to hours to write all this data (even though the actual calculations take only a couple of minutes). Especially for profile ageing (drive cycles), the amount of data generated (and time to write this data) is huge. But it can be used to compare the simulation with lab data.

- Some degradation models significantly increase the calculation time. Most models only add about 10% to the calculation time but some models have more impact. E.g. The degradation models with the stress model from Dai (LAM model 1 or CS model 2) double the calculation time.

Mathematical background

Slide builds upon the spectral single particle model implemented by Adrien Bizeray, Jorn Reniers and David Howey, which is available on GitHub (https://github.com/davidhowey/Spectral_li-ion_SPM). The reader is recommended to first familiarise with the spectral SPM before turning attention to Slide, which extends the spectral SPM with various degradation mechanisms.

There is one main mathematical difference. The spectral SPM used the 'transformed concentration' *u*, which was obtained by multiplying the radius with the lithium concentration at that radius. The final equation of the state space model for solid diffusion of the spectral SPM was:

$$\frac{\partial \boldsymbol{u}_{2:N}}{\partial t} = D_{s,i}(T) A \boldsymbol{u}_{2:N} + B j_i(t)$$

Where $\boldsymbol{u}$ is the transformed concentration at the inner nodes, $D_{s,i}(T)$ is the solid diffusion coefficient of electrode *i* at temperature *T*, $j_i(t)$ is the current density on electrode *i* at time *t* and *A* and *B* are the state space matrices. Although the accuracy of this model is very high even for a low number of discretisation nodes (N), the matrix *A* is full. This project adds one extra transformation to the eigenspace in order to obtain a sparse (diagonal) state space matrix.

Using the eigenvalue decomposition of *A* we can write:

$$\frac{\partial \boldsymbol{u}_{2:N}}{\partial t} = D_{s,i}(T) V^{-1} \Lambda V \boldsymbol{u}_{2:N} + B j_i(t)$$

Where V is a matrix with the eigenvectors and Λ is a diagonal matrix with the eigenvalues. We can simplify this equation to:

$$\frac{\partial (V \boldsymbol{u}_{2:N})}{\partial t} = D_{s,i}(T) \Lambda \, (V \boldsymbol{u}_{2:N}) + V B j_i(t)$$

Using $\boldsymbol{z}_{2:N} = V \boldsymbol{u}_{2:N}$ and $\tilde{B} = VB$ we can write:

$$\frac{\partial \boldsymbol{z}_{2:N}}{\partial t} = D_{s,i}(T) \Lambda \, \boldsymbol{z}_{2:N} + \tilde{B} j_i(t)$$

This equation has the same format as the original equation, but the matrix Λ is diagonal, which increases calculation speed and reduces numerical errors.

This transformation has one additional advantage. One of the eigenvectors represents a uniform concentration. This means that if we start with a uniform concentration, only one value of $\boldsymbol{z}$ will be non-zero. This also means that the corresponding eigenvalue in Λ is 0 (the time derivative of a uniform concentration must be 0 if there is no external current $j_i(t)$) such that the amount of lithium is conserved.

Matlab setup before running the C++ code

Chebyshev spectral methods are used to discretise the diffusion PDE in space. The result of the mathematical derivation are some matrices which are needed to calculate the derivatives, actual concentrations etc.

$$\frac{\partial \boldsymbol{z}_{i,2:N}}{\partial t} = D_{s,i}(T)\widetilde{A}_\iota \, \boldsymbol{z}_{i,2:N} + \widetilde{B}_\iota j_i(t)$$

$$c_i = \begin{bmatrix} \widetilde{C}_\iota \boldsymbol{z}_{i,2:N} + \dfrac{\widetilde{D}_\iota j_i(t)}{D_{s,i}(T)} \\ \widetilde{C_{\iota,c}} \left(\widetilde{C}_\iota \boldsymbol{z}_{i,2:N} + \dfrac{\widetilde{D}_\iota j_i(t)}{D_{s,i}(T)} \right) + \dfrac{j_i(t) D_{s,i}(T)}{\widetilde{D}_i} \end{bmatrix}$$

$\boldsymbol{z}_{i,2:N}$	transformed concentration at the (positive) inner nodes in electrode i
$D_{s,i}(T)$	solid diffusion constant of electrode *i* at temperature *T*.
$\widetilde{A}_\iota$	Diagonal matrix of the state space model for electrode *i* (same matrix as Λ)
$\widetilde{B}_\iota$	Column matrix of the state space model for electrode *i*
$j_i(t)$	current density at time *t* on electrode *i*.
c_i	concentration at the inner and centre nodes of electrode *i* [mol / m3]
$\widetilde{C}_\iota$	Matrix of the state space model linking the actual and transformed concentrations for electrode *i*
$\widetilde{D}_\iota$	Martix of the state space model linking the actual concentration and the current density for electrode *i*
$\widetilde{C_{\iota,c}}$	Matrix of the state space model linking the actual concentration at the centre node to the actual concentration of the inner nodes.

These matrices are calculated by the Matlab function 'modelSetup.m', which also writes their values in csv files. In the C++ code, the values of these matrices are read from these csv files and stored in a Struct.

However, the values of the matrices depend on 3 parameters that the user can change: the radius of each particle and the number of discretisation nodes. Of course, the same values must be used by the Matlab code as by the C++ code (else you are spatially discretising for a radius r1, while you are calculating things for a radius r2, which obviously produces wrong results). When the C++ code reads the matrices, it checks that the parameters are identical, and throws an error if they are not. In that case, the user has to change the parameters in the Matlab code and re-run it to re-calculate the matrices for the new parameters.

In the Matlab code, these parameters are defined at the top of the function 'modelSetup.m'

- nch: the number of inner Chebyshev nodes in the positive domain. The value of this parameter must be the same value as the one used by the C++ code (which is defined on top of State.hpp as nch)
- Rp: The radius of the positive particle. The value must be the same as the value used by the C++ code, which is defined in the constructors of the child-classes of Cell: Cell_Fit.cpp, Cell_KokamNMC.cpp, Cell_LGChemNMC.cpp and Cell_user.cpp
- Rn: The radius of the negative particle. The value must be the same as the value used by the C++ code, which is defined in the constructors of the child-classes of Cell: Cell_Fit.cpp, Cell_KokamNMC.cpp, Cell_LGChemNMC.cpp and Cell_user.cpp

The matlab code calls some Matlab functions which implement the mathematical formulas described in the 'Spectral SPM' and the eigenvalue transformation described above. It writes the following csv files:

- Cheb_input: contains the number of inner nodes, and the radius. This file is read by the C++ code to verify that Matlab used the same values of nch, Rp and Rn as the C++ code
- Cheb_nodes: contains the nondimentional location of the inner nodes of the positive Chebyshev domain. The file is read by the C++ code to fill in the variable xch in the Model-structure
- Cheb_Ap: diagonal elements of the matrix Ap (the other elements are 0)
- Cheb_An: diagonal elements of the matrix An (the other elements are 0)
- Cheb_Bp: elements of the column matrix Bp
- Cheb_Bn: elements of the column matrix Bn
- Cheb_Cp: elements of the matrix Cp
- Cheb_Cn: elements of the matrix Cn
- Cheb_Cc: elements of the column matrix to get the concentration at the centre node
- Cheb_Dp: elements of the column matrix Dp
- Cheb_Dn: elements of the column matrix Dn
- Cheb_Q: elements of the Chebyshev integration matrix Q
- Cheb_Vp: inverse of the eigenvector of the positive electrode
- Cheb_Vn: inverse of the eigenvectors of the negative electrode

As long as the input parameters (N or nch, Rp and Rn) are not changed, the initial Matlab setup can be skipped because the matrices don't change.

The functions used to calculate the updated matrices come from:

- A Matlab Differentiation Matrix Suite by JAC Weideman and SC Reddy (https://uk.mathworks.com/matlabcentral/fileexchange/29-dmsuite and http://appliedmaths.sun.ac.za/~weideman/research/differ.html). The details are published in JAC Weideman, SC Reddy, A MATLAB differentiation matrix suite, ACM Transactions of Mathematical Software, Vol 26, pp 465-519 (2000). (DOI 10.1145/365723.365727).
- Chebfun V4 by the Chebfun Team (https://uk.mathworks.com/matlabcentral/fileexchange/23972-chebfun-v4-old-version-please-download-current-version-instead?focused=5592790&tab=function). License:

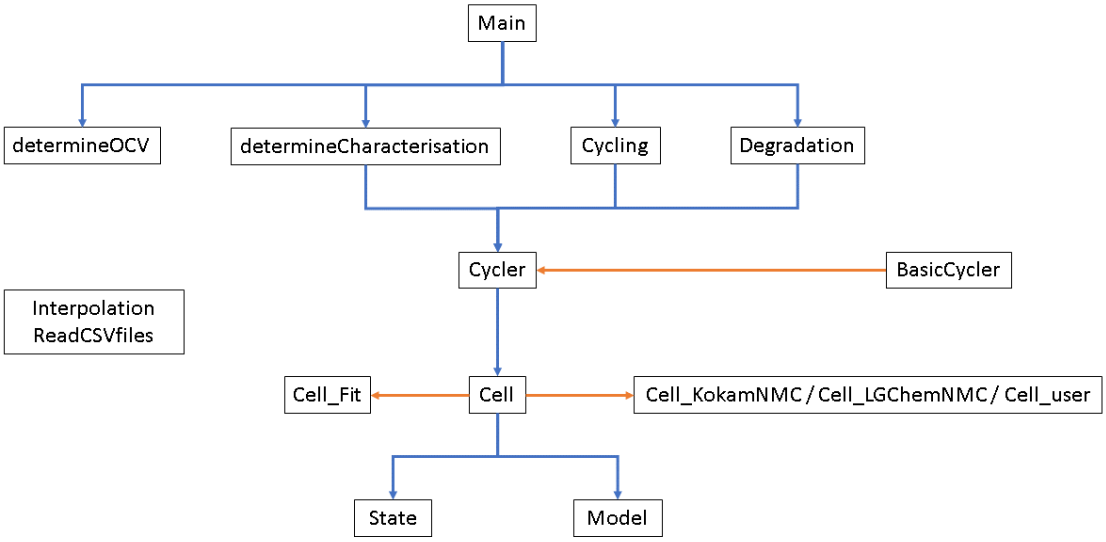
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The C++ code

Code hierarchy

The various cpp files form a structured hierarchy as indicated on the diagram below. Blue lines mean that the functions from the 'top' files uses functions from the 'bottom' files. Orange lines mean an inheritance, where the arrow points from the (file implementing the) parent to the (file implementing the) child class(es).

- Main.cpp is the 'top file' where users select what to simulate.
- There are 4 cpp files implementing the various things we can simulate: determineOCV.cpp, determineCharacterisation.cpp, Cycling.cpp and Degradation.cpp.
- Three of these use a Cycler (defined in Cycler.cpp), which extends the class BasicCycler (defined in BasicCycler.cpp).
- A (Basic)Cycler uses a Cell from Cell.cpp. There are four classes extending Cell, defined in Cell_fit.cpp (used by the functions in determineCharacterisation.cpp for fitting the cell parameters), Cell_KokamNMC.cpp, Cell_ LGChemNMC.cpp, and Cell_user (which are all three used for the functions in Cycling.cpp and Degradation.cpp)
- A Cell uses a State from State.cpp and a Model from Model.cpp
- The functions for interpolating (Interpolation.cpp) and reading csv files (ReadCSVfiles.cpp) are used throughout the code [arrows have not been drawn to avoid confusing the reader]

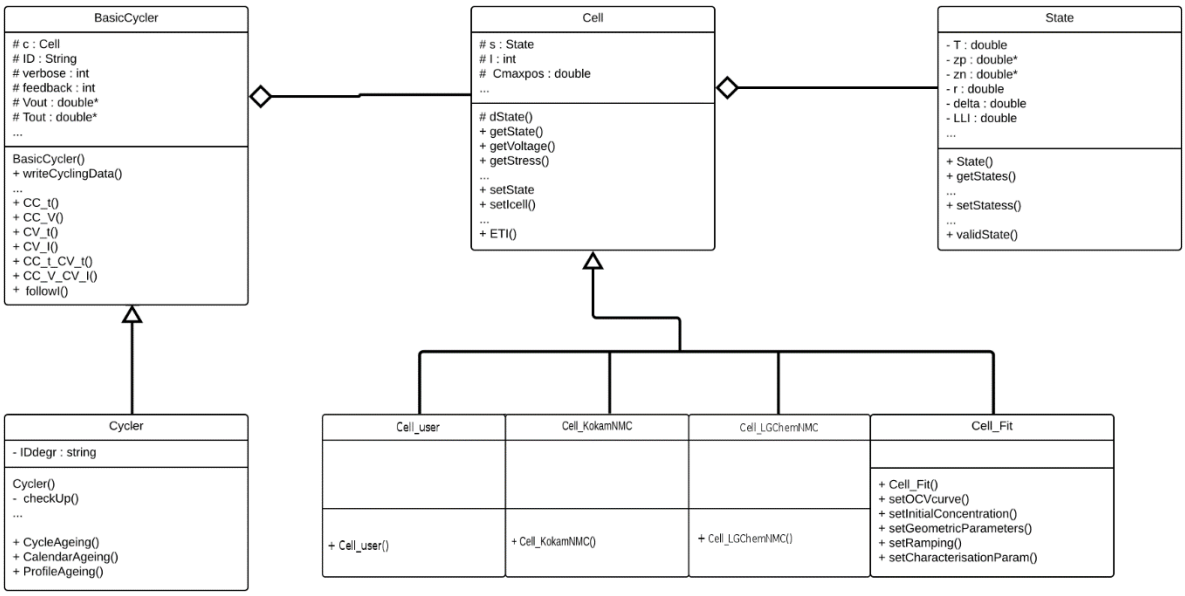


Classes

The C++ code is object-oriented. The class diagram is on the next page.

The 'BasicCycler' class represents a battery tester. It implements functions to load the cell with constant current (CC), constant voltage (CV) or a mix of both. It also stores periodic data about the cell (such as the cell voltage and temperature), which can be written to csv files. A 'basiccycler' has a Cell-attribute representing the cell which should be loaded (i.e the cell connected to the battery test channel).

The 'Cycler' class inherits from the 'BasicCycler' class, and implements the degradation procedures such as cycle ageing, as well as the check-up procedures, such as a capacity measurement.



The ‘Cell’ class is the main class from the model. It implements the battery model in a state-space formulation, where the states are represented by a State-attribute. The function ‘dstate’ calculates the time derivative of all state variables and the function ‘ETI’ performs time integration (using a forward Euler integration scheme). It has private functions which implement the various degradation models, grouped per physical degradation mechanism (e.g. there is a function ‘SEI’ which implements all models for simulating the growth of the SEI layer). It has other getters (e.g. to get the cell voltage) and setters (e.g. to set the environmental temperature). Finally, a cell has an attribute ‘I’ representing the current which is running through the cell. There is a setter to change this current, which can be done in small incremental steps (rather than instantaneously changing the cell current).

There are four classes which inherit from Cell: Cell_KokamNMC, Cell_LGChemNMC, Cell_user and Cell_Fit. The first three don’t implement anything extra on top of what they inherit. The only reason why these classes exist is because they have different values of the parameters, to simulate different types of cells. By having this in different classes, it is more transparent to the user which cell is used where in the code. The fourth child-class, Cell_Fit, does implement extra methods because this class is used when cell parameters are fitted (which is done by the code implemented in ‘determineCharacterision.cpp’). This class therefore lets the user change various parameters of the cell after the object is made. This class (Cell_Fit) should only be used for parameter fitting (by ‘determineCharacterision.cpp’) and not for degradation simulations.

The State-class groups the state-variables of a cell, such as the li-concentration at the spatial discretisation nodes, the cell temperature, the specific resistance, the thickness of the SEI layer, etc. It has getters and setters for most states, as well as a getter and setter which uses an array representation for the states (which is convenient if you want to change all states at once). State also has a function to check if a state-object has valid values. This function will detect if the temperature becomes too high or too low, or if a cell has been degraded too much. It is however not checking if the lithium-concentrations are valid (which is done by functions in ‘Cell’).

Other cpp files

Apart from the classes, there are various other functions implemented in various other cpp files (most of which have corresponding header files).

Main: This file implements the 'main' function of the project. In this main function, the user has to choose which degradation models to use in the battery model, which cell type to use (the Kokam, LG Chem, or user-defined cell) and the prefix which will be the first part of every name of the subfolders in which the csv files with the results will be written.
Then, the user has to specify what should be simulated. All the function-calls are implemented already, but they are commented out. If the user wants to simulate e.g. cycle ageing, he/she has to uncomment the line with the code 'CycleAgeing(M, pref, deg, cellType);'. (uncommenting means you have to remove the double backslash '\\' at the start of the line)
The functions called by the 'main' function are defined in four separate cpp files: Cycling, Degradation, determineOCV and determineCharacterisation.

Cycling: This file groups the functions used to simulate how a cell is cycled. It has 2 functions: CCCV and followCurrent. The function CCCV simulates a few CCCV cycles for a cell and stores the current, voltage and temperature values. This function can be used to check if the parameters of the cell correspond with measured voltage curves. In the function FollowCurrent, a cell has to follow a user-defined current profile. The user specifies in a csv file for how long each current should be maintained, and the cell tries to follow this current profile. Again, the currents, voltages and temperatures are stored every few seconds. This function can be used to simulate e.g. a drive cycle on the cell.

Degradation: This file groups the degradation procedures. It has 3 main functions: CalendarAgeing (where the various calendar ageing simulations are defined), CycleAgeing (where the various cycle ageing simulations are defined), and ProfileAgeing (where the various drive cycle simulations are defined). The functions use multi-threaded code to speed up the calculation even further (by simulating three different degradation experiments at the same time).
This is the file which the user should change to simulate different degradation procedures, e.g. to change the temperature at which calendar ageing is done, or to change the cycle ageing to have only a CC charge instead of a CCCV charge.

determineOCV: This file can be used to fit the model parameters related to the cell's OCV curve. The user has to supply the measured OCV curve of the cell, and the OCV curves of the separate electrodes. The code then fits the geometric parameters (size of the electrodes) and initial li-concentrations of each electrode. The function 'estimateOCVparameters' will call the underlying functions to find the parameters. The search algorithm used is a hierarchical brute-force algorithm. The search takes about one hour.

determineCharacterisation: This file can be used to fit the model parameters related to the characterisation of the cell. The user has to supply data for a few CCCV charges and/or discharges, and the function 'fitCharacterisationAtReferenceT' uses a hierarchical brute-force search algorithm to find the values of the DC resistance, cathode diffusion constant, anode diffusion constant, cathode rate constant and anode rate constants. The search takes a few hours.

Model: This file implements the Structure which stores the matrices used for the spatial discretisation of the solid diffusion PDE. It reads the values from the CSV files and stores them in the matrices which together from the struct.

Interpolation: This file implements two functions for linear interpolation. They only differ in the way the data-arguments are passed.

ReadCSVfiles: This file implements some functions to read CSV files to arrays. They throw errors if the csv files could not be opened.