Using the code, parametrisation of the characterisation curves

This chapter explains how to use the code implemented for finding the parameters of the model such that it can fit your data. The process is very similar to the parametrisation of the OCV curve so it is recommended you first read that document.

It assumes you have installed a C++ editor such as Eclipse, and that you have imported the code of Slide into your editor. All c++ code is in the 'src' subfolder of this project. See the '1 getting started' document on how to do this.

It also assumes you have the files from the Matlab setup in the folder of the project (either because you downloaded them, or because you ran modelSetup.m with the same parameters as in the C++ code). See the chapter on 'Matlab setup before running the C++ code' in the document '2 Overview of the code'.

Overview of the parameters in the model

The main goal of the code is to enable fast simulation of battery degradation using the SPM. Two sets of parameters for the SPM (for two different cells) have been provided. If users want to simulate different cells, they have to provide the parameters of the SPM. The parameters can be grouped in three classes:

- OCV parameters: these include all parameters needed to produce an OCV curve. This includes the half-cell OCV curves, initial lithium concentrations, size of the electrodes, etc.
- Characterisation parameters: these include all parameters needed to calculate the voltage while the cell is cycling (constant current and/or constant voltage). This includes the diffusion constants, rate constants and resistance, etc.
- Degradation parameters: these are the parameters of the different degradation models which will determine how a cell degrades over its life time.

For the first two groups, some basic code is provided to aid users with parametrisation. The degradation parameters have to be found using manual trial and error by the user.

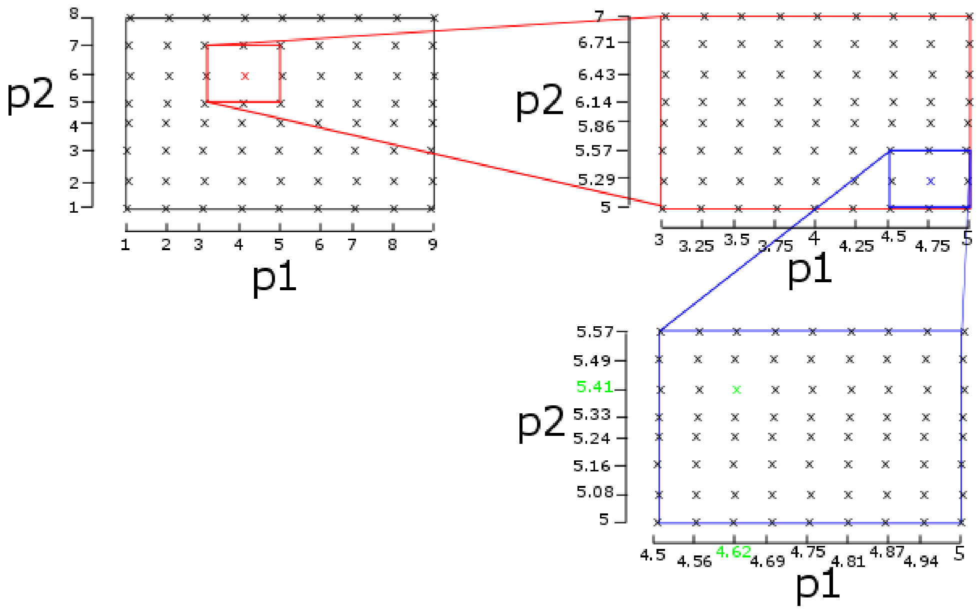
This chapter explains the code provided for the characterisation of the cell, and as said this code is very similar to the parametrisation of the OCV curve.

A) what is implemented & where can you find it?

If the user can supply the information of the OCV curve (see the document on the parametrisation of the OCV curve) and some voltage measurements of a cell which is cycling at a few C rates (potentially with also a CV phase), the code implemented will try to find values of the rate and diffusion constants of each electrodes as well as the DC resistance of the overall battery. This is done for one temperature, but the user can change the temperature to get the parameters at a different temperature (see the last section of this document). All the code is implemented in *determineCharacterisation.cpp* (and the corresponding header file).

For a given combination of the parameters, the code simulates the full CC (or CCCV) charges and discharges on the cell. The (simulated) voltage of the cell is recorded and these voltage curves are compared to the measured voltage curves which were supplied by the user. The difference between both indicates how well this combination of parameters fits the data. The code implemented tries to find the combination which minimises the error.

The parameters are found using the same hierarchical brute force search as was used for the OCV fitting (see '5A Using the code, parametrisation of the OCV curve.docx'). It starts with a large search space and a large step and then iteratively refines the search space and step size while it converges on the combination with the lowest overall error. Have a look at the document about the parametrisation of the OCV curve to find out how it works. Because the problem is not convex, the algorithm might converge to a local minimum and the found parameters might not be very good.



The user should only interact with the function *estimateCharacterisation*. The first code block '*1 user input'* groups the parameters which users have to supply:

- The user can specify at which temperature the characterisation is done by setting the value of '*Tref*' (in Kelvin). It should be noted that the half-cell OCV curves are not changed for a different temperature (i.e. the entropic coefficient is 0), the user-supplied OCV curves are interpreted as the OCV curves at this temperature.
- You have to specify how many data files (and therefore how many half-cycles should be simulated) there are in the variable *nCCCV*. All other inputs will be in arrays with a length of *nCCCV* and of course they must be in the same order (e.g. the length of file i must be in *lengths[i]* and its name must be in *names[i]*).
- The names of the csv file and their length (number of rows) must be supplied to the C++ code in the variables *lengths* and *names*. The cycles must be 'full cycles', i.e. between the minimum and maximum cell voltage (state of charge 0 to 100% or 100% to 0%). The data of each half-cycle (charge or discharge) has to be in a different csv file.
  The csv files must have two columns: the first must give the charge throughput in Ah starting at 0 and strictly increasing. The second column must give the measured cell voltage at that point (increasing if the cell is being charged and decreasing if the cell is being discharged). Ideally you have this data for a few C rates, both for charge and discharge, and potentially with also a CV phase. The more data provided, the better the fit is going to be but the longer the calculation will take.

- Then the user has to tell the C++ code what sort of cycles the data gives by giving the Crate used (where a positive value is a discharge and a negative value is charge) and the cut-off currents used for the CV phase. If the cut-off current is larger than the current from the CC phase, no CV phase is done.
  E.g. if you have data for a CCCV charge at 1C with a current limit of 0.05C for the CV phase, you have to add the following things: in *Crates* -1 (for the 1C CC charge) and in *Ccuts* 0.05 (for the current limit of the CV phase).
  Or if you have data for a 2C CC discharge without CV phase, you have to add: in *Crates* 2 (for the 2C discharge) and in *Ccuts* 5 (or any number larger than 2 such that no CV phase is done).
- The user can allocate a different weight to the different curves by setting different values in the variable *weights*. The 'total' error of a parameter set is the weighted sum of the individual errors for each curve. By giving different weights to different curves, the outcome will have a lower error for the curves with a higher weight. It makes most sense if the sum of the individual weights is 1 (but this is not enforced). Don't put negative values (then the error is maximised)!
- Then the user has to give the name of the csv files in which the results will be written (if the file already exists, its content is overwritten). The files will contain the value of the fitted variables (name in the variable *nameparam*).
  Additionally, the simulated voltage curves with the best-fit parameters will be written to various csv files (one file per half-cycle). Their name will all start with the string in the variable *nameCCCVfit* and after that will be the name of the data file which had the corresponding measurements. So e.g. if there was a data file called 'fit_0.5C_charge.csv' and *nameCCCVfit* has as value 'characterisationFit_' then the output file with the simulated curve will be called 'characterisationFit_fit_0.5C_charge.csv'.
- Finally, the user has to give all the parameters from the OCV fitting. They have been written in an output csv file OCVfit_parameters.csv from *determineOCV.cpp*, see the document '5A Using the code, parametrisation of the OCV curve.docx'. The parameters have to be grouped in the struct *ocvfit*. Additionally, the user has to specify the csv files which have the half-cell OCV curves (and their length) as was the case when fitting the OCV curve.

In the second code block '*2 define the search space for the fitting parameters*', the user has to give a range in which the algorithm should search. For every parameter, a minimum and maximum value has to be given, as well as the step size (from this the number of steps for that parameter can be calculated). It is recommended to use a wide range with a large step size since this is a combinatorial problem (the total number of combinations is the product of the number of steps of every parameter, so if you take 10 steps for each of the 5 parameters, we have to check 100,000 combinations). The hierarchical search will refine the search space to a smaller range and smaller step size. In total, there are 5 parameters

- *Dp*: diffusion constant of the cathode
- *Dn*: diffusion constant of the anode
- *kp*: rate constant of the main li-insertion reaction at the cathode
- *kn:* rate constant of the main li-insertion reaction at the anode
- *rdc:* DC resistance of the battery

In code block 3, the user can specify the number of search levels to use, the more levels the better the fit will be (but the longer the search will take). Then the hierarchical search algorithm is called, which will find the best combination of the 5 parameters.

In code block '4 *write outputs*' the results are processed. The value of the 5 parameters and the remaining error are written to a csv file. Then the characterisation curves of the cell are simulated (with these 5 parameters) and written to a csv file. The Matlab function 'readEstimateCharacterisation.m' will read these simulated curves as well as the (user-supplied) measured curves and plot both. The user can then check how well the simulation fits the data. If the outcome isn't great, you can try with a different initial search space (a slightly smaller step size) and the outcome might be better.

B) How do you run it?

In the *main*-function defined in 'Main.cpp', most variables (celltype, degradation model specification, etc.) are all ignored. The only thing left to do, is need to uncomment (remove the two backslashes in front of the lines) the line which calls *estimateCharacterisation* in the code block '*parametrisation function calls*'.

Ensure that all other function calls (all lines in the blocks under 'PARAMETRISATION FUNCTION CALLS', 'CYCLING FUNCTION CALLS', 'DEGRADATION FUNCTION CALLS') are commented. Every function which is not commented (i.e. the line doesn't start with //) will be executed so if multiple lines aren't, each of these things will get simulated.

Finally build the code and run it as indicated in the document '1 getting started'. With the settings in the code release, the code takes a couple of hours to simulate. This can increase quite significantly if you add a CV phase or if you add more data. It is recommended you let the code run over night.

c) How do you interpret the results?

Per voltage curve you simulated, a csv-file is written with the simulated values according to the fitted parameter set. Their names all start with the string from the variable *nameCCCVfit* (e.g. 'characterisationFit_'), which is followed by the name from the data file with the corresponding measured voltage curve. E.g. if there was a data file 'Characterisation_0.2C_CC_discharge.csv' then the file with the simulated curve will be called 'characterisationFit_Characterisation_0.2C_CC_discharge.csv'. The file has three columns (charge throughput, voltage, and temperature).
The Matlab script readEstimateCharacterisation.m reads the simulated and measured files and plots both. You have to give the names of the simulated and measured files in Matlab (by giving the same variables as in the C++ code, *names* with the names of the data files and the prefix *nameCCCVfit* appended before them to get the simulations).
You can then see how good the fit actually is. If you are not happy, you can rerun the search with different parameters or with a different metric for the error (see below under 'D how do you change something').

Secondly, a csv file with the values of the best-fit parameters is written. The name of the file is given by the variable *nameparam* (e.g. 'characterisationFit_parameters.csv'). You can check that the values are realistic. If you are not happy, you can rerun the search with different parameters or with a different metric for the error (see below under 'D how do you change something'). The file also contains the input parameters which were used to generate this result.

If you are happy with the values found, you have to transfer them to the constructor of one of the Cell-subclasses (Cell_KokamNMC, Cell_LGChem, or Cell_user). This has to be done at the following locations in the constructors:

- Rate constant of the cathode (*kp*): the variable *kp* in the constructors of the Cell-subclasses is the rate constant at the reference temperature for that cell (as given by *Tref* in the constructor, recommended to be 25°). If you did the characterisation at 25°, you can simply copy the value to the code block *'main Li reaction'* in the constructor. If you did the characterisation at a different temperature, you have to change both the value of *kp* and *Tref* (where the latter is in the code block '*thermal parameters*'). Note that the reference temperature is the same for all temperature-dependent variables (diffusion, rate, and OCV curve).

- Rate constant of the anode (*kn*): idem as for the cathode.
- Diffusion constant of the cathode (*Dp*): the variable *Dp* in the constructors of the Cell-subclasses is the diffusion constant at the reference temperature for that cell (as given by *Tref* in the constructor, recommended to be 25°). If you did the characterisation at 25°, you can simply copy the value to the code block '*Initialise state variables'* in the constructor. If you did the characterisation at a different temperature, you have to change both the value of *Dp* and *Tref* (where the latter is in the code block '*thermal parameters*'). Note that the reference temperature is the same for all temperature-dependent variables (diffusion, rate, and OCV curve).
- Diffusion constant of the anode (*Dn*): idem as for the cathode
- DC resistance of the cell (*rdc*): the variable *Rdc* in the constructors of the Cell-subclasses is the DC resistance of the cell (which is considered to be temperature-independent).

Don't forget to also update the parameters from the OCV fitting (see the word document '5A Using the code, parametrisation of the OCV curve.docx').

You can then use the cell-subclass in which you have updated the parameters for other simulations (cycling or degradation) by choosing the correct value of *cellType* in the main function (e.g. if you changed the values in Cell_user you have to use cellType = 2.

D) How do you change something?

Change the half-cell OCV curves

The OCV curves have to be supplied in csv files which have to follow a specific format (see above). You can make new csv files with your new OCV curves and copy these files to the project folder. You then have to change the parameter in the c++ code which gives the name of the csv file with the respective OCV curve (ocvfit.*namexxx*) and the parameter which indicates the length of the OCV curve (the number of rows in the csv file, ocvfit.*nx*). This has to be done in code block '*1 user input*' in the function *estimateCharacterisation* in the file *determinecharacterisation.cpp*.
At the same location, you can also update the values of the maximum lithium concentration in each electrode.

Change the search algorithm

There are two main things you can change in the algorithm: the settings and the error calculation

- Settings: changing the initial search space and number of levels will lead to a different fit. Adding more levels in the search hierarchy (*hmax* in code block 3 in the function *estimateCharacterisation* in the file *determineCharacterisation.cpp*) will improve the fit by refining the values of the parameters a bit more. They should still be close to the original fit, just with more digits (because we converge to the same point but getting a bit 'closer' to the local minimum). Of course, this will increase the calculation time.
  You can also change the initial search space (minimum value, maximum value, step size) for every parameter individually. This might cause the algorithm to converge to a different point altogether (finding another local minimum) so the new best fit might be quite different. If more steps are taken (decreasing the step size or increasing the 'range'), the calculation time will increase as well.
- Another quick change is allocating different weights to different curves. This is done by changing the value of *weights* in code block 1 of *estimateCharacterisation*.

- Another important factor is how the error between the simulated and measured voltage curves is exactly calculated. This is done in the function *calculateError* in the file *determineOCV.cpp*. The currently implemented code uses the 'root mean square error' between both. You interpolate the simulated voltage curves at the measured points (to get the values at the same x-points), and take the difference between both y-values.
  One important factor is how you account for a difference in capacity (e.g. what if the measured voltage curve goes until 3Ah while the simulated voltage curve already reaches the minimum voltage at 2.7Ah?). The implemented code gives the user two options for 'extending' the simulated voltage curve: by either adding points at the lowest voltage or by adding points at 0V (i.e. you add points with x-values from 2.7 to 3 and as y-values Vmin or 0). The latter case penalises a difference in capacity quite a lot.
  Note that this also is 'asymmetric': if the simulated curve is extended, the 'errors in the extra region' are Vmeasured – V_added (where the latter is Vmin or 0); but if the simulated curve is 'longer' (e.g. the measured curve goes to 3Ah while the simulated curve goes to 3.2Ah), there is no need to extend either curve because we use the x-values of the measured curve (so the error is only calculated in the range 0 to 3Ah) and the points simulated from 3Ah to 3.2Ah are simply ignored.
  Users are free to implement their own error-metric which is of course going to affect the 'best' parameter fit.

Temperature dependency of the fitted parameters

Temperature dependency of the diffusion and rate constants is modelled through an Arrhenius relationship (the code assumes the DC resistance is constant for all temperatures). Therefore, the (full) code (as used by the functions in cycling.cpp and degradation.cpp) need the values at a reference temperature (e.g. 25°) and the activation energies. The fitting algorithm does not currently support fitting the activation energy directly.

Instead, you have to do the fitting (independently) at different temperatures. This will result in a set of values for each parameter (one value per temperature). You then have to fit an Arrhenius relation through these points and calculate the activation energies for each parameter (as well as the value at 25°).

The user can change the temperature at which the cell is characterised in code block 1 of *estimateCharacterisation()*. As said before, the user-supplied electrode OCV curves are interpreted as the OCV curves at this temperature. If the entropic coefficient can be neglected, you can keep using the same electrode OCV curve for all temperatures. Else you have to re-calculate them for the new temperature and update the csv file. You can then run the code again (make sure to change the name of the output CSV files so the old results are not overwritten).

Trying out your own parameters & manual parametrisation

In case you already have a set of parameters and want to check how well they fit data, or if the automatic parametrisation failed and you want to do it manually, you can use the CCCV-function from cycling.cpp. A full explanation is in the document '3 Using the code, cycling.docx' but here is a short summary:

- Put your parameters in one of the constructors of a Cell-subclass (e.g. Cell_user in Cell_user.cpp). You have to change the values of all 5 parameters mentioned above, they all affect the cycling behaviour (see the subsection 'C how do you interpret the results'). You also have to change all the OCV-related parameters (this includes linking to the csv files with your half-cell OCV curves), see word document '5A Using the code, parametrisation of the OCV curve.docx'. (Changing the diffusion constant by orders of magnitude might lead to numerical errors, see the section 'Note on numerical stability of …' at the end of this word document.)

- In the function *CCCV* in Cycling.cpp, you can choose the cycles you want to simulate. Set the temperature, Crate of the CC phase, limit current of the CV phase and voltage limits for the cycle(s) for which you have data.
- In the *main* function in Main.cpp, set the value of *cellType* to the cell for which you have set the parameters (e.g. 2 for Cell_user). Set the prefix to some value you will recognize (and to a new value to avoid overwriting data). It doesn't matter which degradation models are included in the simulation since we don't care about degradation at this point. Uncomment the line where you call the function *CCCV* (by removing the double backslash at the start of the line). Comment all other function calls (by adding two backslashes in front of all other lines in the code blocks with function calls).
- Build and run the code. Changing the diffusion constant by orders of magnitude might lead to numerical errors, see the section 'Note on numerical stability of …' at the end of this word document.
- The data of the simulation is written in the newly-created subfolder (called 'xx_yy_CCCV' where 'xx' is the prefix you had chosen and 'yy' the degradation models. There is a full description of the output csv files in the word document '3 Using the code, cycling.docx'.
- You can use the Matlab script readCCCV.m to plot the voltage, temperature, etc. of the cell (you will have to change the value of the prefix and degradation identifiers in the matlab script too). You will have to add your data to this plot yourself. The csv file also has the data about the electrode potentials if you need it.
- You can then change the values in the cell-constructor and simulate again (remember to change the prefix in *main* to avoid overwriting your original results). Then you can manually try to find the correct parameters yourself.

Changing other parameters than the ones mentioned

You can change other parameters of the model as well. There is no code to support those changes, but you can set different values in the constructor of the Cell-subclass you are using (see 'trying out your own parameters …'). Most things you can change without many problems (e.g. the density of the cell, the entropic coefficient, etc.). You only have to be careful with changing the radius of the particles.

It is not recommended to change the value of the radius because you can achieve the same effects by changing the other parameters in the model (e.g. for the diffusion behaviour, it is R/D which matters so you can change the diffusion constant instead; similarly for the amount of active material in which case you can change the thickness or the volume fraction instead). The reason why changing the radius is more difficult is because the spatial discretisation depends on the radius (i.e. the location of the nodes changes if the particle has a different size).

If you want to change the radius, you therefore have to re-compute the spatial discretisation. You can do this with the Matlab script 'modelSetup.m' as is explained in the chapter 'Matlab setup before running the C++ code' in the word document '2 Overview of the code.docx'. You have to change the value of the radius there, and run the scripts. This will write csv files with the new discretisation. If you then also change the values of the radius in the C++ code, it should automatically read these new values and the code should work.

Note that there can only be one spatial discretisation at any given time (the csv files are overwritten when a new discretisation is calculated). This means it is not allowed to use two cell types with a different radius at the same time (which is why both the high-power Kokam and the high-energy LG Chem cell have the same radius). So if you change the radius is one cell type (and therefore also in Matlab and in the written csv files with the discretisation), the code will throw an error if you try to use another cell type which still has the old radius.

Note on numerical stability of changing the radius, diffusion constant or thermal parameters

Sometimes, changing certain parameters by large amounts might lead to errors in the code. This is the case for the radius, the diffusion constant, and all the thermal parameters (cooling, density, heat capacity, etc.). The reason is that the time integration of the diffusion PDE or thermal ODE becomes unstable (i.e. numerical errors blow up to infinity and the value of parameters becomes inf or NaN (not a number). (note: the inf or NaN might appear somewhere else in the code, e.g. in the voltage). In this case, there are a couple of things you can try:

- Reduce the time step: the smaller the time step, the more stable the code. So the first thing to do if you get this problem is to reduce the time step. This will help both for the diffusion and the thermal problem
- Increase the number of discretisation nodes (this will only help if the diffusion equation is the problem; the thermal ODE is not affected). You can do this by increasing the value of *nch* defined on top of state.hpp. You will also have to rerun the Matlab scripts which calculate the spatial discretisation. See the document '2 Overview of the code.docx'.
- Change the time integration method. The code uses (first order) forward Euler time integration, which is known to be not very stable. There are more stable time integration schemes (e.g. Runge Kutta schemes) but they are not implemented. To change the time integration scheme, you have to adapt the function *ETI* in Cell.cpp. The comments clearly state where the time integration is done (first calculating the derivatives and then stepping forward in time using forward Euler), and these lines should be replaced with your new time integration scheme.