Advanced changes to the code

This document gives some explanation of how to do advanced changes to the code. It is highly recommended you do not attempt this unless you know how to program in C++ (including object-oriented programming) and you understand how the code in this project works.

All guidance in this document is only guidance, not a procedure which is guaranteed to work. The changed mentioned in this document haven't been tested so the list of things to change might not be exhaustive (and errors will occur if there are things missing and you only do what is listed here).

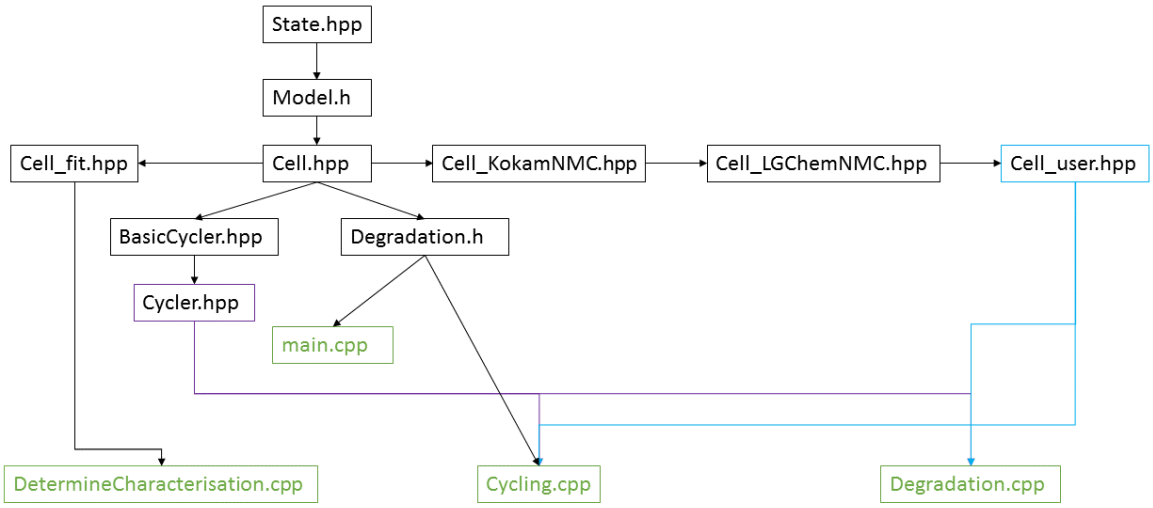
Adding a new cell-type as a new Class

There are in total 4 types of cells

- Cell_fit: this cell is used when fitting the characterisation parameters (see word doc '5B Using the code, parametrisation of the characterisation curves.docx')
- Cell_KokamNMC: this cell has parameters for a high-power Kokam NMC cell and is used for cycling and degradation simulations
- Cell_LGChem: this cell has parameters for a high-energy LG Chem NMC cell and is used for cycling and degradation simulations
- Cell_user: this cell is made for the user to set his/her own parameters for his/her own type of cell (in the released code, the parameters are the same as the ones for the Kokam cell.

Users can make their own sub-class of Cell to implement another cell type. This class has to inherit Cell so that it can be used by the rest of the code. Additionally, the header-file structure must be corrected: the new cell type must include Cell_user.hpp in its header file, and in Cycling.cpp and Degradation.cpp the header type of the new Cell-subclass must be include instead of Cell_user (to avoid double- including header files).

The diagram below shows how the header-files are currently structured (the colours are for clarity only). The new Cell-subclass must be inserted to the right of Cell_user (which it has to include), and the lines to Cycling.cpp and Degradation.cpp have to come from the header file of this new subclass instead of from Cell_user.hpp.



Adding a degradation model

If you find a new degradation model for an existing mechanism (e.g. a new SEI-growth model) and want to implement it, follow these steps:

- Go to the struct with the parameters for that mechanism in Cell.hpp (e.g. *SEIparam*).
- Add the definitions of the fitting parameters you want for that mechanism (e.g. *sei4k* with a rate constant). Set the value in the constructors of all the child classes of Cell.hpp.
- If you need new physical parameters (e.g. the volume fraction of the SEI layer), it is recommended you code them in the Cell-class (e.g. *e_sei*). You can also do this in the struct if you want but it is recommended to put the physical constant in the Cell class directly. Add the parameter definition in the class definition in Cell.hpp and give its value in the constructors of all the child classes of Cell.hpp.
- Go to the relevant function in Cell.cpp (e.g. *SEI*)
- Add a new case in the switch deciding which model to use (based on *deg_id.SEI_id[i]*). There you can code your new model (e.g. as case 4), using the newly defined fitting parameters (e.g. *seiparam.sei4k*) and the newly defined physical cell constants (e.g. *e_sei*). The battery states are in the field *s* of the Cell (so e.g. the cell's temperature can be accessed by *s.getT()*). If this model affects a battery state which so far was not affected by the degradation mechanism, this extra state must become an output parameter, do the following:
    - Adapt the function definition in the hpp and cpp file, e.g. add *double* thickn* as output if you want to decrease the electrode thickness due to SEI growth.
    - You then have to set the value of this new output variable in your new model, and set it to 0 for all other models.
    - In *dstate*, when the degradation mechanism is called, add the extra output variable, e.g. *double dthickn_sei;* and *SEI(OCVnt,…,&dthickn_sei)*
    - Then at the last code block of *dstate* (where the time derivatives are calculated), you have to add the effects from the new degradation models (e.g. *dstates[2*nch + 4] = dthickn_sei + dthickn;)*.
- It is a good idea to update the comments for the degradation models (the DEG_ID struct is defined in Cell.hpp and its model identifiers are repeated in *main* in main.cpp).

- Then you can simply select this new model in the *main* function in main.cpp (e.g. *deg.SEI_id[0] = 4*) and the code will use this new degradation model.

Adding a degradation mechanism

There are currently 4 mechanisms implemented (SEI growth, surface cracking, LAM and lithium plating). This is a purely 'logical' split (i.e. you can put a model for SEI growth in the function which normally does LAM growth), so in theory you can add your new model for the new mechanism under one of the existing mechanisms (see the section 'adding a degradation model'). This is however a bit messy, so it is better to add a fully new degradation mechanism (e.g. binder decomposition or BD). To do that, follow these steps:

- In Cell.hpp, go to the definition of the struct DEG_ID. Add the identifiers for the models for this mechanism (e.g. *int BD[10]* which will allow you to use 10 models for this new mechanism at the same time and *int BD_n*).
- In *main* in main.cpp add values for the new fields in DEG_ID you have made (e.g. *deg.BD[0] = 1;* and *deg.BD[1] = 4;* and *deg.BD_n = 2;* to use models 1 and 4 for binder decomposition)
- In the Cell-class you will have to define a new function which will implement the various models for this new mechanism (e.g. *BD*). You have to add this function definition in the header file and implement it in the cpp file. You can give the function the inputs you need. The outputs must be the time derivatives of the battery states on which this degradation model has effect (e.g. *double* den, double * dep* if you want to change the volume fraction of active material in both the anode and cathode).
- Implement the models for this new mechanism. You can follow the same structure as the existing function. E.g. using a loop for each model to use (*from 0 to deg_id.BD_n*) and a switch (on *deg_id.BD[i]*) to implement the various models.
- In the function *dstate* in Cell.cpp you have to call the new degradation function (e.g. *double dep_bd, den_bd;* and *BD(&den_bd, &dep_bd)*. Then at the last code block of *dstate* (where the time derivatives are calculated), you have to add the effects from the new degradation models (e.g. *dstates[2*nch + 5] = dep + dep_bd;* and *dstates[2*nch + 6] = den + den)sei + den_bd;)*.

Adding a new state-variable

This is a very complicated procedure, please only do this if you understand the code.

A battery has a certain number of states, which are grouped in the class State. State-variables are all independent parameters of Cell which vary over time (things like the voltage and stress are not independent, they can be calculated if the battery states are known). If you want to implement a new model or a new mechanism, you might have to add new state-variables. If it are totally new variables, you can add them directly. In most cases, the variables will be constants in the Cell-class and they have to be removed there first.

In the example used below, you want to implement a new degradation model which will decrease the rate constant of the li-insertion reaction at the graphite (*kn*) over time (e.g. due to SEI growth).

- On the top of State.hpp, increase the value of *ns* to reflect the new number of state variables e.g. to *(2*nch+15)*
- In the class definition of State in State.hpp add the following
    - A field with the variable in the private part of State, e.g. *double kn*
    - A getter and setter in the public part of State, e.g. *double getKn();* and *void setKn();*
    - Add an extra input parameter in the function *iniStates*, e.g. *double kni*

- In the class implementation in State.cpp, add the following
    - In the constructor, *State()*, initialise the new state to 0
    - In the function *iniStates(..)* add the new input parameter in the function definition (*double kni*) and set the value of the state-variable (*kn = kni*)
    - In the function *getStates(int nin, double states[])* add the new variable in the output array, e.g. *states[2*nch+14] = kn*). Also update the assert-statement which checks the number of states (this is done such that you don't forget to add the state variable here), e.g. *assert(ns == 2*nch + 15)*
    - Add the getter and setter functions which returns / set the value of *kn*
    - In the function *setStates(int nin, double states[])*, add the new variable, e.g. *kn = states[2*nch+14]*. Also update the assert-statement which checks the number of states
    - In the function *setStates(State si)* add the new variable, e.g. *kn = si.getKn()* and update the assert-statement which checks the number of states
    - In the function *validState()* add a check on the allowed values, e.g. *bool knmin = kn < 0*, print an error message if the value is not allowed and update the condition when the error is thrown, e.g. *if (Tmin || .. || knmin)*
- Remove the definition of *kn* in the class definition of Cell, in Cell.hpp (at this point a lot of errors will appear in Cell.cpp). If your newly added state was not a class-variable you can skip this step
- Go to the implementation of the Cell class in Cell.cpp and do the following
    - Everywhere where the original class-variable *kn* was called, there is now an error. Replace the calls to *kn* with the getter for the new state *s.getKn()*. If your newly added state was not a class-variable you can skip this step
    - Add the degradation model/mechanism which affects the new state as explained in the previous chapters, e.g. add SEI growth model 4 with the model for the decrease in kn.
        - If the decrease is conditional on the amount of SEI growth predicted by the other models, you can add it as a new case of *deg_id.SEI_porosity*, e.g. else *if deg_id.SEI_porosity == 1 {*dkn = seiparam.sei_newParameter * *isei}* (and you can make a case ==2 with both cases 0 and 1 if needed, or you an add a new identifyer for the decrease in kn in DEG_ID).
        - You will have to add new output values for kn in the degradation mechanism which will decrease kn, see the section on 'Adding a new degradation model'.
    - In *dstate* you have to add the time derivatives of this new state at two locations
        - In the code block '*if we ignore degradation in this time step, we have calculated everything we need*' you have to set the time derivative of the new state to 0 (assuming the new state is only affected by degradation, if it is a cycling state then you have to give the time derivative here) , e.g. *states[2*nch+14] = 0*. Also update the assert-statement which checks the number of states
        - In the final code block titled '*time derivatives*' you have to give the value of the time derivative for the new state (if it was a cycling-state then you have to give the same value as before; if it is a degradation state then here you give the nonzero time derivative) , e.g. *states[2*nch+14] = dkn*. Also update the assert-statement which checks the number of states
- In the function *checkUp_batteryStates(..)* in Cycler.cpp the value of the new state will be automatically recorded. I.e. in all the output csv files, there will be an extra column with the evolution of the new state parameter, e.g. the one but last column in the csv file shows how *kn* decreases over time. You don't have to change anything here, it's mentioned so you know how the new state will be recorded.

- In the Matlab script readAgeing_BatteryState.m you have to account for this extra column (which is not going to be the last column, as you can see in *checkUp_batteryStates(..)*, the last column is the battery resistance while the new state variable is the one but last column). E.g. in Matlab, add *state{i}.kn = A(:,30);* and state{i}.R = A(:,30);. If you want to display the evolution of the new state variable, you have to add a subplot in the figure (or make a new figure)
- In *main* in main.cpp, call the degradation model/mechanism which will affect the new state and simulate it.