Appendices

Debugging & common errors

This section gives some advice when errors appear in the code.

The code is written with a defensive programming style. This means that exceptions are thrown in case of unexpected events (the 'throw' statements in the code). When calling functions which might throw errors, the code is usually wrapped in a try-catch block (*try {…} catch(int e){…}*) to handle the errors. Some errors can be corrected by the calling function, while others can't and are thrown on to the higher functions. When an error reaches the top level (i.e. *main*), the simulation crashes.

Normally, there are print statements every time is thrown or caught. Users can choose from the 'verbose' levels which determine how many of these statements are actually printed to the terminal. The value of *verbose* is set in the *main*-function (in main.cpp)

- 0: almost no error messages are printed. Only illegal inputs are reported (e.g. negative voltages). If you choose this level of verbosity, the code might crash without printing a message.
- 1: only 'fatal' errors are printed. I.e. if we think this error is going to make the code crash, an error message is printed. If we think the error can be corrected by higher functions, no message is printed. It isn't always known at the lowest level if an error is going to be fatal or not, so this is not exact. I.e. the code might crash without printing a message and you might see an error message, but the code can recover from this error. It is recommended you use this setting as standard.
- 2: all error messages are printed. Every time an error is thrown, a message is printed irrespectively of whether the error can be corrected or not. This setting might produce a lot of output in certain cases. If the code crashes and you don't know why, it is recommended you change *verbosity* to 2 so you can see what the last error was (and this error will cause the code to crash).
- 3: Apart from printing all the error messages, also a message is printed every time a function in the Cycler or BasicCycler is started and terminated. This will allow you to find in which function the error happens. Note that this generates a lot of print statements, especially from the function *findCVcurrent* and *findCVcurrent_recursive* in BasicCycler.cpp (this function solves the nonlinear equation to find the current required to keep a given voltage).
- 4: On top of the previous output, also some information about the high-level flow of the program is printed by the functions in the Cycler, of the type "in function xxx we are going to charge the cell". This might help to logically understand why an error occurs and where exactly it is (e.g. only after 1500 cycles into the degradation profile). As said before, there will be a lot of output from *findCVcurrent* and *findCVcurrent_recursive* so it might be a good idea commenting out the two print statements in that function (put a double back slash in front of the print statements at the start and end of *findCVcurrent* and *findCVcurrent_recursive* in BasicCycler.cpp).
- 5: On top of the previous output, detailed information is printed by the functions in the BasicCycler, of the type 'in time step xx the cell voltage is yy V and the cell current is zz A'.
- 6: On top of the previous output, *findCVcurrent_recursive* prints details of the iterations when it is trying to find the current required to reach a given voltage.
- 7: On top of the previous output, a message is printed every time a function in the Cell is started. This will help locate the error if it happens in a function from Cell.

If you are calculating with multiple threads, error messages of different threads might interrupt each other. I.e. you get first some words from one message from thread x, then some words for an error message from thread y, etc. If you are debugging, it is recommended you isolate the thread which causes the problem (try them one by one if you are not sure which thread is causing the error). Comment-out all other threads by adding a double backslash in front of the line defining them (*thread x (..)*) and in front of the line joining them (*x.join()*), where x is the name of the thread. Then you can clearly see the error messages from the one thread which causes the error. This is especially the case if you set *verbosity* to 2 or higher.

All the errors also carry an identification code (i.e. a number). The excel document 'Error IDs.xlsx' lists the numbers, the location where this error originates from and a very short description of the error. This information should already be in the error message which is printed but in case of confusion you can look the codes up.

To locate the error, it is recommended you slowly increase the value of *verbose* in the *main* function. Start with 2, if you can't see the problem set it to 3, then to 4, etc. A lot of output will be generated from 3 onward (due to the print statements in *findCVcurrent_recursive*), too much to follow what is going on. Higher values will produce even more output, so they should only be used to exactly locate an error and understand why this is happening.
Due to the high output (from 3 or higher), the simulation will go significantly slower. Especially from 5 onward, the simulation slows down by orders of magnitude. So only set these higher levels when debugging.

Common errors and how to handle them

Below is a list of some errors which might appear in the code if you change certain settings.

Illegal state (error 15)

This error is thrown if an invalid battery state is detected (by the function *validState* in State.cpp. It is treated like a fatal error and the simulation stops. This is done because illegal states might lead to unexpected errors later in the code. E.g. if the temperature becomes too high, the numerical time integration might become unstable and you get infinite or NaN (see the last section of document '5B Using the code, parametrisation of the characterisation code.docx' or the section on numerical problems below). This error will also be thrown if a cell has degraded too much (e.g. the electrodes are too thin). This check is done to avoid infinite loops, numerical problems etc. which will happen if the electrodes get very thin (or some other state has degraded too much).

However, an illegal state does not immediately lead to an error, and the boundaries of what is 'legal' are arbitrarily set, apart from the non-negativity of most variables (negative values will lead to errors).

If you get this error but want to continue simulating, you can increase the allowed range of the variables (e.g. increase the maximum temperature to 75 degrees if you want to simulate a cell of 60 degrees). You can also disable the throw-statement (i.e. add a double back slash in front of the statement '*throw 15*') but this is not recommended.

Illegal voltages or currents with the wrong signs, (errors 1004, 1005, 1006 and 1007)

These errors are the most difficult to solve. Users can define two ranges.

- The minimum and maximum allowed voltage of the cell (*Vmax* and *Vmin*), defined in the constructors of the Cell-subclasses
- The upper and lower voltage between which you want to load the cell (*Vupp* and *Vlow)*. E.g. doing a CC discharge to a *Vlow* > *Vmin* or following a drive cycle current profile while staying

between certain voltage limits. This variable is an input to the functions in the BasicCycler (and is therefore determined by the various functions in the Cycler).

Usually the problem is that in one way or another, the cell has gotten to a voltage which is just below the lower voltage limit, or just above. When the error message is printed, it gives the actual voltage and the limit which is exceeded. Often, these numbers might look identical but that is due to rounding. In reality, the actual voltage might be a few micro-Volts above the maximum or below the minimum (so if you print up to the mV range, the numbers look the same).

There is no easy solution for this problem. Simply increasing the allowed range typically doesn't solve it because the limits are shared for all functions (e.g. you first charge to *Vupp*, then do something which causes the voltage to go just above *Vupp* (e.g. changing the cell temperature or current) and then you try to charge again: this last charge leads to an error in *CC_t_V* (error 1005) and increasing *Vupp* won't solve it (because then you also charge a bit more). Some functions can recover from this error (e.g. *CC_t_CV_t* which does both a CC and CV will try to skip the CC phase and go straight to the CV phase which sometimes is possible but not always). Other functions will just throw on the error and the code crashes.
In this case, the only solution is to manually find how the cell got outside of the allowed range and then correcting that line. Isolate the thread which causes the error (i.e. comment out all others by adding a double backslash in front of the line defining and joining them) and increase the verbosity.

File could not be opened (error 1001)
The code tries to create a folder with the user-defined name, consisting of the prefix (in *main*) and the identifier of the Cycler (in the functions in Cycling.cpp and Degradation.cpp). This means that the values of those strings must be valid names for folders. If they are not, the subfolder can't be created and you get an error when you try to open a file in this folder.

Try to simplify the name and omit symbols and spaces (e.g. backslashes or forward slashes are not usually allowed, underscored should be fine).

Arrays have the wrong length (errors 10, 100, 107, 111, 1002, 10001)
In C++, you have to define how long the array is when it is created. When you later fill in values in the array, you have to be very careful. The compiler doesn't check the length of an array, neither statically when compiling nor dynamically when running the code. This means the code would just continue running until at some point you get a very strange error because a value had been overwritten with the extra value of the array.

To avoid this, the code throws an error to notify you in locations where there is a danger of arrays being too small (i.e. you try to fill more values than there are in the array). You can simply increase the size of the array where it is defined to accommodate the extra values.

You will have to find the location where the array is defined, and its length (which might be in a separate variable). This might be in a higher-level function (e.g. if an array is used to provide output for a function, the higher-level function which calls this function has to define the array used for output, so it is in this higher-level function that you have to increase the size). In these cases, the error message should show the trace of the error (i.e. where it originated, where it was caught, etc.) so you can look at the function above the one which threw the error.

Out of memory
In Eclipse, you have to specify the amount of memory you will allocate to your program. Allocating too much might swamp the RAM of your computer (especially when you don't need it), allocating too

little leads to a very strange error: you don't get any error message at all but suddenly the program crashes. This typically happens when you create a new variable or when you call a new function (which would exceed the allowed memory space).

You can allocate the amount of memory in the settings of your project in Eclipse.

- Right click on the project folder (in eclipse) and select 'properties' all the way at the bottom.
- In the pop-up window on the left, extend 'C/C++ Build' (click on the white triangle) and in the drop-down menu 'Settings'. On the tab 'tool settings' check the following:
- MinGW C++ Linker -> Miscellaneous: in the box called 'Linker flags' type the following text (without the quotation marks): ' -Wl,--stack,8000000000 '(this option increases the amount of RAM that the simulation can use. If your computer doesn't have much RAM you might have to reduce this number; if the simulation runs out of RAM, you will have to increase this number. There are also settings to dynamically allocate RAM but they don't always work.

Numerical problems

Changing certain parameters (the radius, the diffusion constant, and all the thermal parameters (cooling, density, heat capacity, etc.) or using extreme states (too high temperature, too much degraded cells, etc.) might lead to errors in the code. There are two potential reasons

Numerical stability

** this section is the same as the last section in the word document '5B Using the code, parametrisation of the characterisation curves.docx' **

In this case, the problem 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.

Discretisation problems (e.g. error 101)

In this case, the problem is that because you take discrete time steps, the states vary in discrete steps too. This might mean that one state is valid, while the state in the next time step is illegal.

Consider a case where the cell is being discharged to the minimum voltage (e.g. 2.7V). Over the time steps, the lithium concentration in the anode increases (and it decreases in the cathode), which will cause the voltage to decrease. Ideally this goes in small steps (the lithium fractions change by a few percentages and the voltage by a few millivolts). In this case you can stop when you reached a step just below the limit (e.g. 2.69V).

But if the current or time step is very large, or the amount of active material or the diffusion constant is very small, the changes in surface concentration (and cell voltage) over one time step become very large. Suppose that the voltage at time t is 2.71V with a corresponding lithium fraction in the anode of 0.96, so the code takes another time step to further discharge the cell. But due to the large change in surface concentration, the lithium fraction in the anode now becomes 1.01, which is illegal (the lithium fraction has to be smaller than 1). This will lead to error 101.

To solve this, try the following.

- First check the parameters you are using (e.g. you got the wrong units and are now trying to run a 1000C current instead of a 1C). Idem for the diffusion constant, electrode thickness, etc.
- Secondly you can decrease the time step, which will decrease the change per time step.
- Another thing you can try is to decrease the rate of change in the cell's current (*dIcell* in the constructors of the Cell-subclasses). This will solve the case where the problem happened when you were suddenly changing the applied current which lead to a crazy surface concentration (but not the case where you were just running a large current and then the error happened)
- Finally, you can be more conservative with your limits. In the case above, increasing the minimum voltage to 2.75V will solve the problem because you stop one time step earlier when the concentration is still in the allowed range.

Debugging for other problems

For other problems, the error message hopefully explains what is wrong, which allows you to correct the error (e.g. you specified a negative current limit for the CV phase). To debug, try the following:

- Isolate the thread which produced the error. Comment-out all other threads by adding a double backslash in front of the line defining them (*thread x (..)*) and in front of the line joining them (*x.join()*), where x is the name of the thread.
- Increase the verbosity (in the functions in Cycling.cpp and Degradation.cpp which cause the problem) with the variable *verbosity*.
- Find where in the code the error happened, and why it happened.
- Try to solve it
- If you have no clue, try disabling the throw-statement. Certain errors are not really a problem, just things which should not happen or things which might cause problems later in the code. This is the case for the 'illegal state'-error (15), the 'initial state'-error (14) and a whole bunch of other errors. If the code runs fine after you disabled the error, then ignore the error. If you get all sorts or other errors, you will have to correct the original error.

The basics of c++

C++ is similar to C, Java and Python (and probably many others). If you can code in one of those languages, you should be able to understand the written code. There are some syntax differences, but nothing out of the ordinary. You can find good documentation, help with the syntax and example code fragments on www.cplusplus.com.

There are significant differences with more applied programming languages such as Matlab. I recommend to first have a look at some basic tutorials before making significant changes in the code. There are loads of good user guides, tutorials, etc. available for free on the internet. Personally I like the tutorials from cplusplus.com (http://www.cplusplus.com/doc/tutorial/)

The basics of object-oriented programming

The code is written in an object-oriented programming style. Below is a very short introduction to the basics of object-oriented programming. It should be enough to understand the code from this project, but you can find much more information on the internet if things are not clear.

Classes and objects

The basic elements are classes and objects. They are best explained with a real-world example. Think of a car. A car has certain properties (e.g. a colour, a brand, a year in which the car was built, etc.), a given state (e.g. the speed, location, etc.). A class is an abstract representation of what 'a car' is (i.e. a 'template' for a class). The properties and states are represented as so-called 'class variables'. They are defined somewhere in the class (without giving values for those variables, the class is just the template defining what properties a car has). In pseudo-code, our Car-class looks like:

```
Class Car {
  string brand
  string colour
  int buildYear
  double speed
  …
}
```

Up to here, a class is very similar to a struct in Matlab, C or other programming languages. The main difference is that in the real-world, there are clearly defined 'things' a car can 'do' (e.g. accelerate, turn, etc.). While in Matlab or other non-object-oriented programming functions to 'do things' have to be defined externally, they are included in a class in object-oriented code. So a class groups both the variables AND the functions related to the 'thing' the class represents. Let's add a function to accelerate the car to our class. The function has one variable, the increase in speed *ds*. Our Car-class could for instance look like:

```
Class Car {
  string brand
  string colour
  int buildYear
  double speed
  ...

  accelerate (double ds) {
        Speed = speed + ds
  }
}
```

Note that you can access the class variables in functions as if they were local variables. The class variables are shared between all functions of the class, so if one function changes the value than the value will also be different in the other functions. In fact, they are some sort of 'local global' variables for all the functions inside our Car-class.

An Object is an 'instance' of our class, i.e. with concrete values for the class variables. E.g. we can make an object called *c1*, and set its variables as follows:

```
Car c1 = new Car
```

c1.brand = "Opel"
c1.colour = "green"
c1.buildYear = 2015
c1.speed = 0

In this case, we have the Object *c1* which is a green Opel car made in 2015 which is stationary. If we now want to accelerate our car by 1 m/s, we can call the function from the object:

c1.accelerate(1)

Now our green Opel car will be driving at 1m/s, i.e. *c1.speed* will be 1.

One of the advantages is that it significantly simplifies the book keeping if we have many cars in our simulation. E.g. if we want to make two more cars, *c2* and *c3*, we type the following:

Car c2 = new Car
c2.brand = "Ford"
c2.colour = "red"
c2.buildYear = 2016
c2.speed = 0

Car c3 = new Car
c3.brand = "Renault"
c3.colour = "yellow"
c3.buildYear = 2010
c3.speed = 0

Now we can use our three cars independently of each other and we don't have to remember the state and parameters of each car. E.g. if we type *c2.accelerate(2)*, the Ford is driving at 2m/s, while the Opel is still driving at 1m/s and the Renault is still stationary. This means we don't have to remember which speed corresponded to which car (because each car remembers its own speed).

Encapsulation

Encapsulation means that inside a class, you 'hide complexity' to the outside world. A class has 'private' features which are only accessible from within the class and 'public' features which can be accessed externally. This reduces complexity and decreases the potential for errors in the code. It is good coding practice to make all class variable private, and make public functions to get their values ('getters') and to set their values ('setters'). In the setters, you can then check if the values are realistic, which will avoid getting errors in the code. The user can only change the value of the variable with the setter, so every time we set a new value, it will be checked on validity, thus avoiding getting wrong values. E.g. if we only allow build years after 1800, our Car could look like:

```
Class Car {
  private string brand
  private string colour
  private int buildYear
  private double speed

  public getBrand() {
      return brand
  }
  public getColour() {
      return colour
  }
  public getbuildYear() {
      return buildYear
  }
  public getSpeed() {
      return speed
  }

  public setBrand(string newBrand) {
      brand = newBrand
  }
  public setColour(string newColour) {
      colour = newColour
  }
  public setbuildYear(int newYear) {
      if newYear > 1800 {
            buildYear = newYear
      }
      else{
             throw error
      }
  }
  public setSpeed(int newSpeed) {
      speed = newSpeed
  }

  Public accelerate (double ds) {
            speed = speed + ds
  }
}
```

Relations between classes

Another great feature of object-oriented classes is that they can be linked to each other. Suppose we have a class to represent a tire, e.g. with a radius and the air pressure in the tire:

```
Class wheel {
  private double radius
  private double pressure
  public getRadius {
        return radius
  }
  public getPressure {
        return pressure
  }
  public setRadius(int newRadius) {
      radius = newRadius
  }
  public setPressure(int newPressure) {
      pressure = newPressure
  }
}
```

Now we can add the wheels to our car. Class variables can be of any type, allowing this sort of relations.

```
Class Car {
  private string brand
  private string colour
  private int buildYear
  private double speed
  private Wheel leftFront
  private Wheel leftRear
  private Wheel rightFront
  private Wheel rightRear

  public getBrand() {
      return brand
  }
  public getColour() {
      return colour
  }
  public getbuildYear() {
      return buildYear
  }
  public getSpeed() {
      return speed
  }
  public getWheels() {
      return leftFront, leftRear, rightFront, rightRear
  }


  public setBrand(string newBrand) {
```

```
        brand = newBrand
    }
    public setColour(string newColour) {
        colour = newColour
    }
    public setbuildYear(int newYear) {
        if newYear > 1800 {
            buildYear = newYear
        }
        else{
            throw error
        }

    }
    public setSpeed(int newSpeed) {
        speed = newSpeed
    }
    public setWheels(Wheel newlf, Wheel newlr, Wheel newrf, Wheel newrr) {
        leftFront = newlf
        leftRear = newlr
        rightFront = newrf
        rightRear = newrr

    }

    Public accelerate (double ds) {
            speed = speed + ds
    }
}
```

And let's make 4 wheels with the same radius but different pressures

```
Wheel w1 = new Wheel
w1.setRadius(0.5)
w1.setPressure(10)

Wheel w2 = new Wheel
w2.setRadius(0.5)
w2.setPressure(11)

Wheel w3 = new Wheel
w3.setRadius(0.5)
w3.setPressure(9)

Wheel w3 = new Wheel
w3.setRadius(0.5)
w3.setPressure(12)

Wheel w4 = new Wheel
w4.setRadius(0.5)
w4.setPressure(10)
```

Now we can add the wheels to our cars:

```
c1.setWheels(w1, w2, w3, w4)
```

We could do the same with other wheels and other cars. As you can see, this will reduce the amount of bookkeeping we have to do to remember which wheel was part of which car (because the cars remember themselves).

Also keeping consistency between our objects becomes easier. Suppose that wheels also have a class-variable *rotationSpeed*, which gives the speed at which wheels are rotating (and we have implemented the standard getters and setters for the rotation speed in our Wheel-class). Of course, the speed of the car is related to the rotation speed of the wheels (the speed of the wheels is the speed of the car divided by the radius of the wheels). Every time we update the speed of the car, we have to update the speed of the wheels as well. The code below shows how the functions *setSpeed* would look like when we take advantage of these features. In our Car-class we can access the radius of the wheels using the getter from our Wheel class (*getRadius)*, and set the rotation speed of our wheels using the setter from Wheel (*setRotationSpeed(double newRotationSpeed)*).

```
public setSpeed(int newSpeed) {
    speed = newSpeed
    leftFront.setRotationSpeed (newspeed / leftFront.getRadius())
    leftRear.setRotationSpeed (newspeed / leftRear.getRadius())
    rightFront.setRotationSpeed (newspeed / rightFront.getRadius())
    rightRear.setRotationSpeed (newspeed / rightRear.getRadius())
}
```

Now in the function *accelerate*, we can simply call the setSpeed function such that we don't have to worry about the rotation speed of the wheels when accelerating the car:

```
Public accelerate (double ds) {
        setSpeed(getSpeed() + ds)
}
```

Our Car-class has fully 'hidden' the wheels to the outside world. When you change the speed of the car, you don't have to think about the wheels. This is a good example of the 'encapsulation' principle mentioned before.

And much more

Object-oriented programming has many other features, such as inheritance (if a child class 'inherits' from a parent class, it is as if the child class extends the parent class, i.e. it has all the features from the parent class plus any additional featured implemented in the child class).

There are again loads of tutorials online. A short introduction is given on
http://ee402.eeng.dcu.ie/introduction/chapter-1---introduction-to-object-oriented-programming