Developer Manual of BatPar

Contents

0. Introduction

0.1 Checklist before you start

Targeted prospective contributors of BatPar, this manual mainly addresses methodologies of previous development and slightly involves suggestions for further development. Below is a checklist for readers of this manual, which should be fulfilled before attempting to understand the underlying codes of BatPar; otherwise, you may find great difficulty in further development.

☒ I have installed *MATLAB 2021a* (or above) with *Signal Processing Toolbox* and *Optimization Toolbox*

☒ I have used BatPar and successfully got parameterisation results from it (at least once).

☒ I have experience developing MATLAB codes and understand MATLAB data types, expressions, functions, GUI, optimisation algorithms, etc.

Don’t worry if you cannot fulfil the above checklist –

– if you don’t have too much experience with MATLAB, here is a recommended book of MATLAB basics, “*Programming and Engineering Computing with MATLAB (2022)*”. Essential chapters are those EXcluding Chapters 6, 9-13.

– if you haven’t used BatPat yet, there are slides and a presentation recording introducing BatPar. Moreover, below is a step-by-step demo of using BatPar.

Step 1 – Make sure that you have below the files in place, as shown in *Fig. 1*. Fig. 1 will be referenced throughout this manual, so bear in mind the location of Fig. 1.

Step 2 – Run *User_Inputs.m* in MATLAB, and a Graphical User Interface (GUI) will appear as below in *Fig. 2*, where the editable places are numbered from ① to ㊵.

Fig. 2 will be referenced throughout this manual, so bear in mind the location of Fig. 2.

Step 3 – Open */ InputData/ Demo GITT 25degC 2C-rate.csv/* and have a quick look - this spreadsheet stores the GITT test data of a sample cell (4.4Ah, 2.7V~4.2V). Let’s say you want to parameterise the sample cell with the GITT test data. **The folder */InputData* is where battery test data should be placed**.

(If your spreadsheet is saved in the format of ‘.xlsx’, don’t forget to close your spreadsheet before proceeding. This is because MATLAB cannot read any opened ‘.xlsx’, as the operating system locks the read access of opened ‘.xlsx’. Spreadsheets saved in ‘.csv’ do not have this problem)

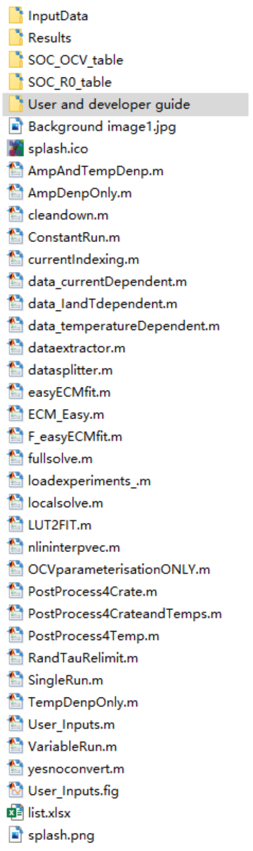
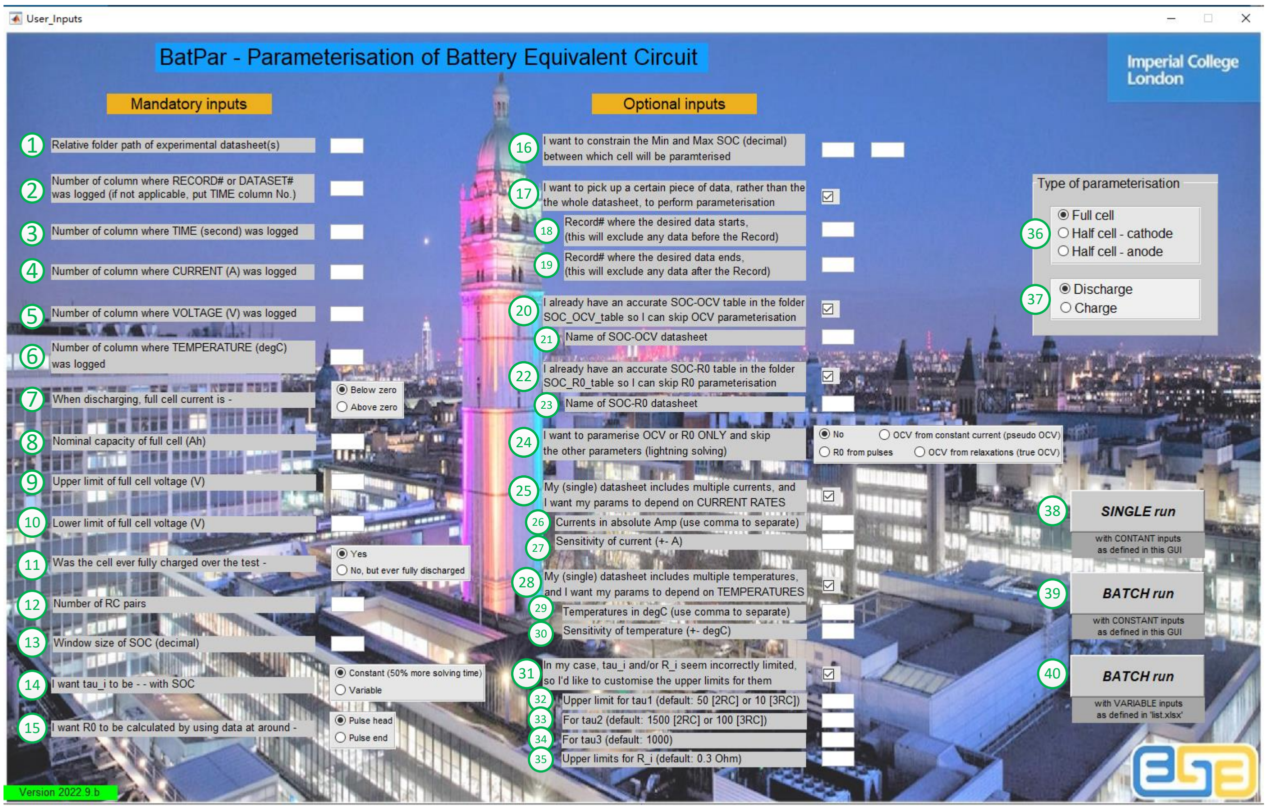

InputData
Results
SOC_OCV_table
SOC_R0_table
User and developer guide
Background image1.jpg
splash.ico
AmpAndTempDenp.m
AmpDenpOnly.m
cleandown.m
ConstantRun.m
currentIndexing.m
data_currentDependent.m
data_IandTdependent.m
data_temperatureDependent.m
dataextractor.m
datasplitter.m
easyECMfit.m
ECM_Easy.m
F_easyECMfit.m
fullsolve.m
loadexperiments_.m
localsolve.m
LUT2FIT.m
nlininterpvec.m
OCVparameterisationONLY.m
PostProcess4Crate.m
PostProcess4CrateandTemps.m
PostProcess4Temp.m
RandTauRelimit.m
SingleRun.m
TempDenpOnly.m
User_Inputs.m
VariableRun.m
yesnoconvert.m
User_Inputs.fig
list.xlsx
splash.png

Fig. 1. Make sure that you have the files in place.


User_Inputs

BatPar - Parameterisation of Battery Equivalent Circuit

Imperial College London

Mandatory inputs

1 Relative folder path of experimental datasheet(s)
2 Number of column where RECORD# or DATASET# was logged (if not applicable, put TIME column No.)
3 Number of column where TIME (second) was logged
4 Number of column where CURRENT (A) was logged
5 Number of column where VOLTAGE (V) was logged
6 Number of column where TEMPERATURE (degC) was logged
7 When discharging, full cell current is - [✓ Below zero] Above zero
8 Nominal capacity of full cell (Ah)
9 Upper limit of full cell voltage (V)
10 Lower limit of full cell voltage (V)
11 Was the cell ever fully charged over the test - [✓ Yes] No, but ever fully discharged
12 Number of RC pairs
13 Window size of SOC (decimal)
14 I want tau_i to be - - with SOC [✓ Constant (50% more solving time)] Variable
15 I want R0 to be calculated by using data at around - [✓ Pulse head] Pulse end

Optional inputs

16 I want to constrain the Min and Max SOC (decimal) between which cell will be paramterised
17 I want to pick up a certain piece of data, rather than the the whole datasheet, to perform parameterisation ✓
18 Record# where the desired data starts, (this will exclude any data before the Record)
19 Record# where the desired data ends, (this will exclude any data after the Record)
20 I already have an accurate SOC-OCV table in the folder SOC_OCV_table so I can skip OCV parameterisation ✓
21 Name of SOC-OCV datasheet
22 I already have an accurate SOC-R0 table in the folder SOC_R0_table so I can skip R0 parameterisation ✓
23 Name of SOC-R0 datasheet
24 I want to paramerise OCV or R0 ONLY and skip the other parameters (lightning solving) [✓ No] R0 from pulses OCV from constant current (pseudo OCV) OCV from relaxations (true OCV)
25 My (single) datasheet includes multiple currents, and I want my params to depend on CURRENT RATES ✓
26 Currents in absolute Amp (use comma to separate)
27 Sensitivity of current (+- A)
28 My (single) datasheet includes multiple temperatures, and I want my params to depend on TEMPERATURES ✓
29 Temperatures in degC (use comma to separate)
30 Sensitivity of temperature (+- degC)
31 In my case, tau_i and/or R_i seem incorrectly limited, so I'd like to customise the upper limits for them ✓
32 Upper limit for tau1 (default: 50 [2RC] or 10 [3RC])
33 For tau2 (default: 1500 [2RC] or 100 [3RC])
34 For tau3 (default: 1000)
35 Upper limits for R_i (default: 0.3 Ohm)

Type of parameterisation
36 [✓ Full cell] Half cell - cathode Half cell - anode
37 [✓ Discharge] Charge

38 SINGLE run with CONTANT inputs as defined in this GUI
39 BATCH run with CONSTANT inputs as defined in this GUI
40 BATCH run with VARIABLE inputs as defined in 'list.xlsx'

Version 2022.9.b

Fig. 2. The GUI and the numbering of editable places.

Step 4 – Use the GUI to configure the inputs of parameterisation. Details follow:

① *InputData*

② *1*

③ *4*

④ *8*

⑤ *9*

⑥ *10*

⑦ *Below zero*

⑧ *4.4*

⑨ *4.2*

⑩ *2.7*

⑪ *Yes*

⑫ *2*

⑬ *0.05*

⑭ *Constant*

⑮ *Pulse head*

⑰ *(tick ON the box)*

⑱ *260*

⑲ *293322*

㉔ *No*

⑳ ㉒ ㉕ ㉘ ㉛ *(tick OFF the boxes)*

㊱ *Full cell*

㊲ *Discharge*

Step 5 – Click '*SINGLE run*'. Then you can find messages being updated in MATLAB command window (hopefully you will not see any error message).

Step 6 – Allow 15 minutes or so for BatPar to complete running. After completion, **you will find parameterisation results saved in */Results/***, as shown in *Fig. 3*. Have a quick check on the result files. All done!

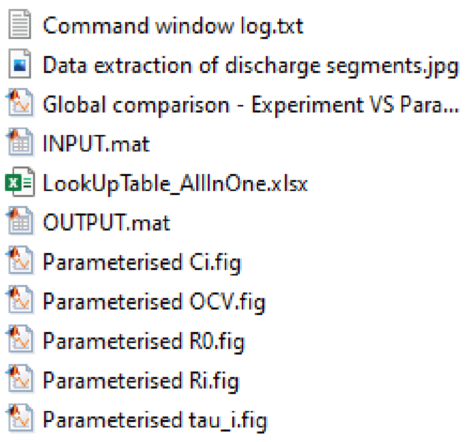

Command window log.txt
Data extraction of discharge segments.jpg
Global comparison - Experiment VS Para...
INPUT.mat
LookUpTable_AllInOne.xlsx
OUTPUT.mat
Parameterised Ci.fig
Parameterised OCV.fig
Parameterised R0.fig
Parameterised Ri.fig
Parameterised tau_i.fig

Fig. 3. Parameterisation results are saved in */Results/*.

0.2 Principles of BatPar

BatPar is a MATLAB program that extract equivalent circuit parameters of batteries from experimental datasheets.

1> The parameters are defined as **open-circuit voltage (OCV)**, **resistances (Rs) and capacitances (Cs)** in an equivalent circuit, as *Fig. 4*.

2> The extracted parameters are **SOC** dependent and can also be **temperature (T)** and/or **current (I)** dependent.

3> The experimental data, saved in the format of ***.csv*** or ***.xlsx***, should contain **time, current and voltage** at least, while **temperature** is optional.

4> Users of BatPar need to know **battery capacity** and **voltage range** as inputs. Besides, users also need to decide on the **number of RC pairs** and the **SOC window size** as inputs.

5> A **GUI** is made as the entry to BatPar, as *Fig. 2*. In the GUI, user inputs are divided into **mandatory** and **optional** ones. Optional ones can be left blank.

6> BatPar can accept data from **any standard test procedure** like GITT, AMPP, HPPC, or **arbitrary tests** without pulses (prescribed R0 needed as inputs). BatPar can also parameterise **multiple datasheets** in a row.

7> After running, BatPar can **automatically** sort out the parameters and plots and save them in the result folder.

8> More about the solving methods of BatPar: the parameters are treated as **unknowns** to be solved by a **fitting algorithm**. The fitting algorithm, combined with the **governing equations** below, aims to find parameters that can best **match the fitting model with experimental data**. Best match is evaluated by **the gap (RMSE) between experimental voltage and fitting model voltage**, if the fitting model is given the same current as the experiment.

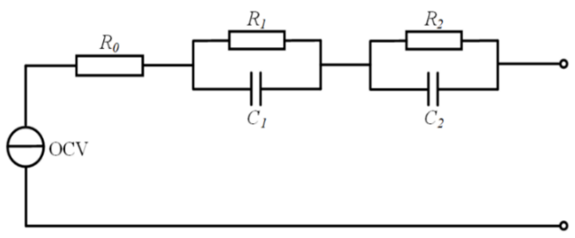


Fig. 4. A typical battery equivalent circuit with two RC pairs.

$$V = OCV - IR_0 - V_{r1} - V_{r2}$$

$$\dot{V}_{r1} = -\frac{V_{r1}}{R_1C_1} + \frac{I}{C_1} \xrightarrow{discretisation} V_{r1}(t+1) - V_{r1}(t) = -\frac{V_{r1}(t)}{R_1C_1} + \frac{I(t)}{C_1}$$

$$\dot{V}_{r2} = -\frac{V_{r2}}{R_2C_2} + \frac{I}{C_2} \xrightarrow{discretisation} V_{r2}(t+1) - V_{r2}(t) = -\frac{V_{r2}(t)}{R_2C_2} + \frac{I(t)}{C_2}$$

9> More about the functionalities of BatPar: please look at the functionality matrix below.

$$\begin{bmatrix}\text{full cell}\\ \text{anode}\\ \text{cathode}\end{bmatrix} \times \begin{bmatrix}\text{discharge}\\ \text{charge}\end{bmatrix} \times \begin{bmatrix}\text{1 RC}\\ \text{2 RC}\\ \text{3 RC}\end{bmatrix} \times \begin{bmatrix}\text{OCV, Rs, Cs}\\ \text{Rs, Cs}\\ \text{OCV only}\\ \text{R0 only}\end{bmatrix} \times \begin{bmatrix}\text{variable tau}\\ \text{constant tau}\end{bmatrix} \times \begin{bmatrix}\text{SOC dependence}\\ \text{SOC + I dependences}\\ \text{SOC + T dependences}\\ \text{SOC + I\&T dependences}\end{bmatrix} \times \begin{bmatrix}\text{Single processing}\\ \text{Batch processing}\end{bmatrix}$$

All set with the introduction? **The remaining manual will go through every underlying file of BatPar (as shown in Fig. 1), with emphasis on: <1> the functionality and structure of each file; <2> explanations of core codes and hard-to-read codes <3> tips for future improvements, if any.** The appearance order of files follows their execution sequence and importance degree; for example, the file *User_inputs.m* is the first file to be executed when using BatPar and also an important file of BatPar, so it is the first file to be reviewed in the following. One last thing before proceeding to the following chapters – please have a look at the **FLOWCHART** in *Fig. 5* – by this chart you will know how scripts/functions interact with each other. One last thing for sure – if you want to know which of the scripts/functions are the most important ones as **the 'soul' of BatPar** – they are ECM_easy.m, loadexperiments_.m and easyECMfit.m.

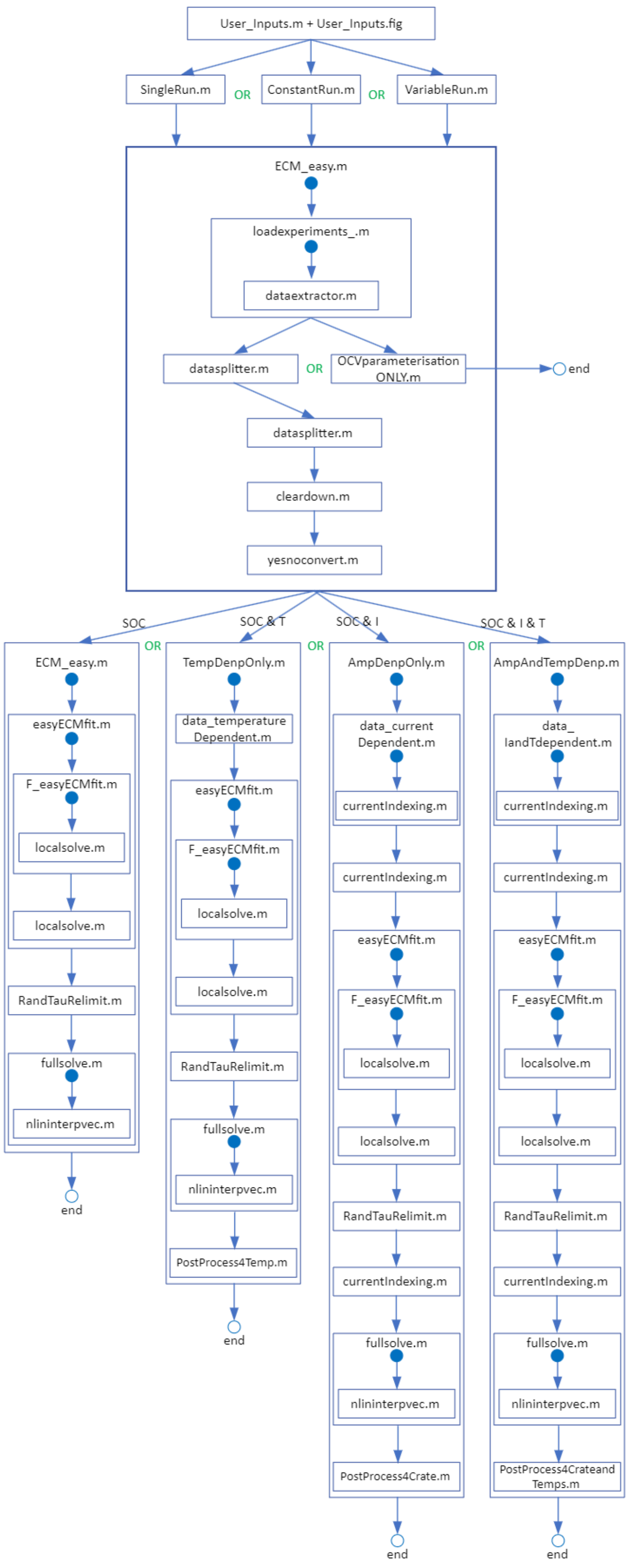


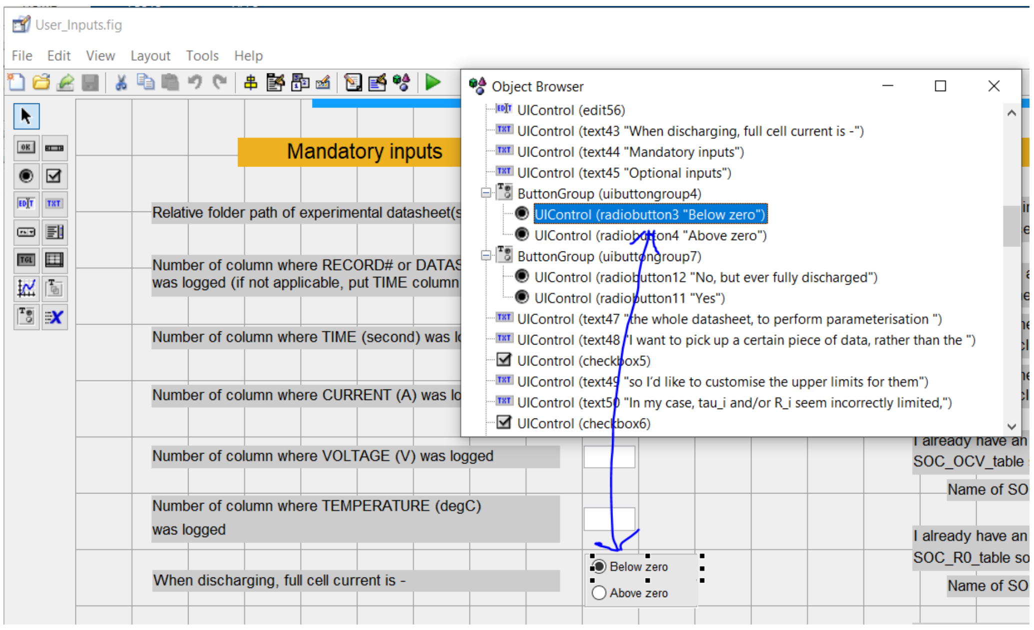
Fig. 5. Flowchart of BatPar and interactions between scripts/functions.

1. *User_Inputs.m* & *User_Inputs.fig*

1.1 Functionality and structure

User_Inputs.m & User_Inputs.fig are the two files to enable the GUI (Fig. 2.).

User_Inputs.fig claims objects (e.g., radiobutton) of the GUI. If you want to add/delete objects to/from the GUI, User_Inputs.fig is the file to edit – please **use 'GUIDE'** to open and edit User_Inputs.fig. Once opened with GUIDE, all the objects will be editable, as Fig. 6., and you can use Object Browser to match the ID with location of each existing object.


User_Inputs.fig
File Edit View Layout Tools Help
Mandatory inputs
Relative folder path of experimental datasheet(s
Number of column where RECORD# or DATAS
was logged (if not applicable, put TIME column
Number of column where TIME (second) was l
Number of column where CURRENT (A) was lo
Number of column where VOLTAGE (V) was logged
Number of column where TEMPERATURE (degC)
was logged
When discharging, full cell current is -
Below zero
Above zero
I already have an
SOC_OCV_table
Name of SO
I already have an
SOC_R0_table so
Name of SO
Object Browser

UIControl (edit56)
UIControl (text43 "When discharging, full cell current is -")
UIControl (text44 "Mandatory inputs")
UIControl (text45 "Optional inputs")
ButtonGroup (uibuttongroup4)
UIControl (radiobutton3 "Below zero")
UIControl (radiobutton4 "Above zero")
ButtonGroup (uibuttongroup7)
UIControl (radiobutton12 "No, but ever fully discharged")
UIControl (radiobutton11 "Yes")
UIControl (text47 "the whole datasheet, to perform parameterisation ")
UIControl (text48 "I want to pick up a certain piece of data, rather than the ")
UIControl (checkbox5)
UIControl (text49 "so I'd like to customise the upper limits for them")
UIControl (text50 "In my case, tau_i and/or R_i seem incorrectly limited,")
UIControl (checkbox6)

Fig. 6. Use *GUIDE* to open and edit *User_Inputs.fig*.

User_Inputs.m defines the callback functions of objects. When the user is operating the GUI, (e.g., clicking one of the radiobuttons), the callback functions will take actions (e.g., change the value of one input variable from 0 to 1) based on their underwriting codes.

If you want to know more about GUI development, please refer to the book recommended above.

1.2 Core codes and hard-to-read codes

User_Inputs.m

```
ii=imread('Background image1.jpg');
```

In User_Inputs.m, you may use this line of codes to customise the background image of GUI. Also, put your image file in the root folder of BatPar.

User_Inputs.m

```
if get(handles.radiobutton17,'Value')==1
    assignin('base','Mode_anode_tf','yes');
else
    assignin('base','Mode_anode_tf','no');
end
```

Throughout the codes within User_Inputs.m, you can find lots of ‘assignin’ commands. Those commands pass user’s inputs from the GUI to **MATLAB workspace (named as `'base'`)**, so that the main program of BatPar (ECM_easy.m) can read inputs from the workspace and perform further calculations. The codes above assign the value of ‘`Mode_anode_tf`’ based on the user’s choice with ‘`radiobutton17`’. If the user clicks ‘㊱ Half cell – anode’, then the value of `radiobutton17` will be 1, so the string `'yes'` will be assigned to variable `Mode_anode_tf` in the workspace.

User_Inputs.m

```
function pushbutton3_Callback(hObject, eventdata, handles)
…

…
run("SingleRun.m")
```

Function pushbutton3_Callback defines the following actions to take, once the user pushes the button ‘㊳ Singe Run ’. The codes above indicate that `"SingleRun.m"` will be launched.

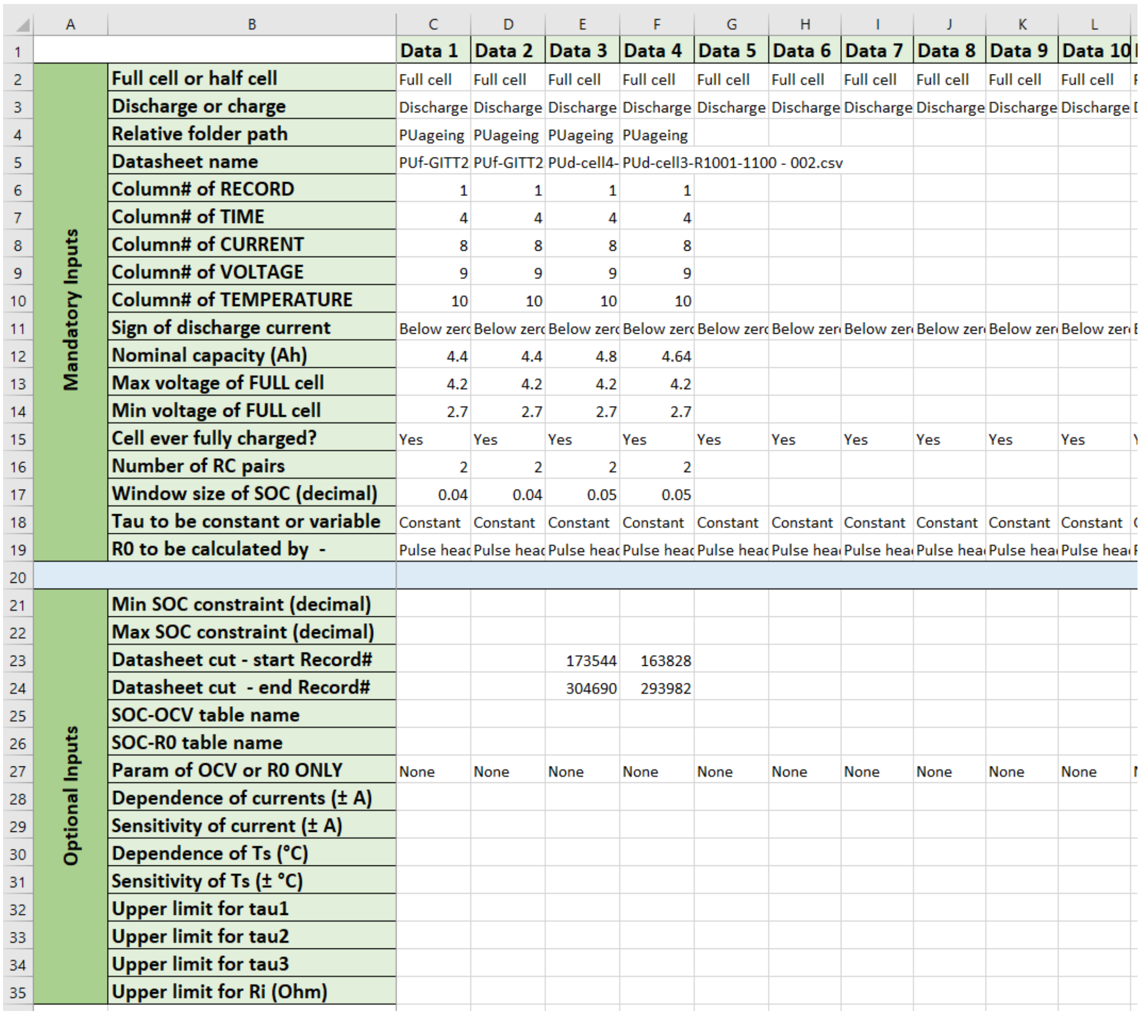
----

2. *SingleRun.m* & *ConstantRun.m* & *VariableRun.m* & *list.xlsx*

2.1 Functionality and structure

SingleRun.m is the launcher for singe-datasheet parameterisation, while ConstantRun.m and VariableRun.m are launchers for multiple-datasheet parameterisation (**batch-processing**).

ConstantRun.m takes inputs from the GUI – this means all the datasheets to be parameterised will adopt the same inputs as specified in the GUI. In contrast, VariableRun.m takes inputs from 'list.xlsx', as Fig. 7., and the user can put different inputs specific to each datasheet. In other words, VariableRun.m along with list.xlsx is developed to cater for users' demands of input-variant parameterisation, given that datasheets can be obtained from different cells/experimental set-ups/ test protocols and the user may have different requirements for parameters from different datasheets.



	A	B	C	D	E	F	G	H	I	J	K	L
1			**Data 1**	**Data 2**	**Data 3**	**Data 4**	**Data 5**	**Data 6**	**Data 7**	**Data 8**	**Data 9**	**Data 10**
2	**Mandatory Inputs**	**Full cell or half cell**	Full cell	Full cell	Full cell	Full cell	Full cell	Full cell	Full cell	Full cell	Full cell	Full cell
3		**Discharge or charge**	Discharge	Discharge	Discharge	Discharge	Discharge	Discharge	Discharge	Discharge	Discharge	Discharge
4		**Relative folder path**	PUageing	PUageing	PUageing	PUageing						
5		**Datasheet name**	PUf-GITT2	PUf-GITT2	PUd-cell4-	PUd-cell3-R1001-1100 - 002.csv						
6		**Column# of RECORD**	1	1	1	1						
7		**Column# of TIME**	4	4	4	4						
8		**Column# of CURRENT**	8	8	8	8						
9		**Column# of VOLTAGE**	9	9	9	9						
10		**Column# of TEMPERATURE**	10	10	10	10						
11		**Sign of discharge current**	Below zerc	Below zerc	Below zerc	Below zerc	Below zerc	Below zer	Below zer	Below zer	Below zer	Below zer
12		**Nominal capacity (Ah)**	4.4	4.4	4.8	4.64						
13		**Max voltage of FULL cell**	4.2	4.2	4.2	4.2						
14		**Min voltage of FULL cell**	2.7	2.7	2.7	2.7						
15		**Cell ever fully charged?**	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
16		**Number of RC pairs**	2	2	2	2						
17		**Window size of SOC (decimal)**	0.04	0.04	0.05	0.05						
18		**Tau to be constant or variable**	Constant	Constant	Constant	Constant	Constant	Constant	Constant	Constant	Constant	Constant
19		**R0 to be calculated by -**	Pulse heac	Pulse heac	Pulse heac	Pulse heac	Pulse heac	Pulse hea	Pulse hea	Pulse hea	Pulse hea	Pulse hea
20												
21	**Optional Inputs**	**Min SOC constraint (decimal)**										
22		**Max SOC constraint (decimal)**										
23		**Datasheet cut - start Record#**			173544	163828						
24		**Datasheet cut - end Record#**			304690	293982						
25		**SOC-OCV table name**										
26		**SOC-R0 table name**										
27		**Param of OCV or R0 ONLY**	None	None	None	None	None	None	None	None	None	None
28		**Dependence of currents (± A)**										
29		**Sensitivity of current (± A)**										
30		**Dependence of Ts (°C)**										
31		**Sensitivity of Ts (± °C)**										
32		**Upper limit for tau1**										
33		**Upper limit for tau2**										
34		**Upper limit for tau3**										
35		**Upper limit for Ri (Ohm)**										

Fig. 7. Screenshot of 'list.xlsx', which is used together with VariableRun.m to perform input-variant, multiple-datasheet parameterisation.

2.2 Core codes and hard-to-read codes

ConstantRun.m

```
Filescsv = dir(strcat('*.csv'));
Filesxlsx = dir(strcat('*.xlsx'));
Files = [Filesxlsx;Filescsv];
```

BatPar can recognise datasheets in the format of .csv or .xlsx, while .txt can not be recognised. In ConstantRun.m, the codes will detect and parameterise every .csv and .xlsx successively.

ConstantRun.m

```
for tttttttt = 1:length(Files)
    clearvars -except tttttttt Files exp_path record_column time_column
    current_column voltage_column temp_column row_start row_end Nom_capacity
    voltage_up_limit voltage_low_limit OCVsheet_name RCpair_number SOC_window
    SOC_param_lowlimit SOC_param_uplimit tau1_limit tau2_limit tau3_limit Ri_limit
    Currents_4_Denp CurrentSensi_4_Denp Temps_4_Denp TempSensi_4_Denp
    Current_belwozero_tf MaxSOCis1 Mode_anode_tf Mode_charge_tf Mode_cathode_tf
    Pulse_head_tf fixedtau_realy_tf OCVtable_tf datasheet_cleanup_tf
    Ri_taui_relimit_tf CurrentDenp_tf TempDenp_tf
    ...
    ...
    fig1h = findall(0,'type','figure','Tag','figure1');
    figh = findall(0,'type','figure');
    other_figures = setdiff(figh, fig1h);
    delete(other_figures);  % close all figures except GUI
end
```

For multiple-datasheet parameterisation, **one key point to follow is to avoid the current parameterisation being interfered by the previous one**. To this end, '`clearvars`' is used to clear up the variables of previous parameterisation – those variables had already been saved with previous parameterisation but can only be disturbance to the current parameterisation. ConstantRun.m uses '`clearvars -except`' to remove some of the variables in the workspace but simultaneously keeps the other which are global inputs applicable to all the datasheets. You can find similar codes with the same nature in VariableRun.m.

Another practice implemented for multiple-datasheet parameterisation is about display of plots. In single-datasheet parameterisation, plots of results will automatically pop up after parameterisation finishes. However, **this feature can cause heavy loads to memory and GPU in case of multiple-datasheet parameterisation** – let's say you have 200 datasheets being parameterised in a row, and after all done, you will have more than 1,000 plots displayed on the monitor. To tackle this, after the current parameterisation done and results saved, the codes will '`close all figures except GUI`' before proceeding to the next parameterisation.

VariableRun.m

```
for iiiiii = 3: columns_
    if ismissing(todolist{5, iiiiii})
        column_dataend = iiiiii-1;    % the last datasheet specified
        break
    else
        column_dataend = columns_;
    end
end
```

As aforementioned, VariableRun.m takes inputs from ‘list.xlsx’, as Fig.7. To find out how many datasheets are there in list.xlsx and will be parameterised, the above codes detect the existence of the row ‘Datasheet name’ in each column of list.xlsx and thus confirm the number of datasheets to be parameterised.

SingleRun.m or *ConstantRun.m* or *VariableRun.m*

```
run ECM_Easy.m
```

In any of SingleRun.m, ConstantRun.m or VariableRun.m, you can find that the main program `‘ECM_Easy.m’` is called. It is called only once in SingleRun.m, but multiple times within a for loop in ConstantRun.m or VariableRun.m, depending on the number of datasheets.

----

3. *ECM_easy.m*

3.1 Functionality and structure

ECM_easy.m is the heart of BatPar – generally, all the other functions and scripts are developed to interact with ECM_easy.m. Therefore, understanding ECM_easy.m in full is vital for further development of BatPar. More details are given in Section 3.2.

3.2 Core codes and hard-to-read codes

```
% naming of the result folder(s)
cathodeORnot = evalin('base','Mode_cathode_tf');
anodeORnot = evalin('base','Mode_anode_tf');
if strcmp(cathodeORnot,'yes')
    fileNAME = [fileNAME,'_Cathode'];
else
    if strcmp(anodeORnot,'yes')
        fileNAME = [fileNAME,'_Anode'];
    else
        fileNAME = [fileNAME,'_FullCell'];
    end
end
...
...
cd Results
mkdir(fileNAME)
cd(fileNAME)
diary 'Command window log.txt' % log file starts
cd ..
cd ..
```

The codes above are to create a new folder under '\Results', so that parameterisation results can be finally saved in the newly created folder. Folder naming follows 2 principles: (1) folder name starts with the name of datasheet. (2) folder name ends with suffixes that indicate important configurations of parameterisation, such as full-cell/cathode/anode, charge/discharge, I-dependence/T-dependence. E.g., let's say the datasheet being parameterised is named 'PUd-cell4-R601-700 – 001.csv', and you use the GUI to configure the parameterisation as full-cell and discharge, without considering any of I- or T-dependence. In this case, you will find parameterisation results under the folder as shown in Fig. 8.

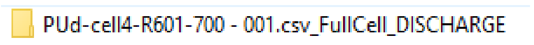

PUd-cell4-R601-700 - 001.csv_FullCell_DISCHARGE

Fig. 8. An example of folder naming based on datasheet name and configurations of parameterisation.

More in details of the codes, '`cathodeORnot`' is the variable indicating whether cathode parameterisation is configured – '`evalin`' is used to extract the configuration from another variable (`'Mode_cathode_tf'`) in the workspace (`'base'`). Throughout Easy_ECM.m, you can find lots of ‘`cd`’ and ‘`cd ..`’ commands – they are used to enter and exit different folders so that new files can be created under the folder path. ‘`diary 'Command window log.txt`’ means that anything to be displayed in MATLAB command will be logged in `'Command window log.txt`’. At the end of Easy_ECM.m, the line ‘`diary off`’ puts an end to logging.

```
%% Division of SOC windows
S_interval = str2num(evalin('base','SOC_window'));    % the SOC interval you want
in look-up tables;  >= 0.01
if strcmp(flags.MaxSOCis1.tf,'yes')  % 100% SOC is the baseline
S_lowerlimit = ceil (S_lowerbound/S_interval)*S_interval; % the lower limit of
DIVIDABLE SOC in look-up tables
S_j = S_lowerlimit:S_interval:1; % SOC points: from S_lowerlimit to 1, with a
fixed interval - S_interval
S_j = (vertcat(S_lowerbound,S_j'))'; % SOC points: from S_lowerbound to
S_lowerlimit to 1
[~,ind,~] = unique(S_j(1,:)); % ~:neglected  ind:row mubers of unique times in N
S_j = S_j(:,ind);                       % keep data at unique times
else   % 0% SOC is the baseline
S_upperlimit = 1- ceil ((1- S_upperbound)/S_interval)*S_interval; % the upper
limit of DIVIDABLE SOC in look-up tables
S_j = 0:S_interval:S_upperlimit; % SOC points: from 0 to S_upperlimit, with a
fixed interval - S_interval
S_j = (vertcat(S_j', S_upperbound))'; % SOC points: from 0 to S_upperlimit to
S_upperbound
[~,ind,~] = unique(S_j(1,:)); % ~:neglected  ind:row mubers of unique times in N
S_j = S_j(:,ind);
End
```

The codes above are to divide SOC windows based on the user’s inputs in ⑪ and ⑬. E.g., let’s say you put 0.04 in ⑬ and selected ‘yes’ in ⑪, then the SOC division (`S_j`) will be `[1, 0.96, 0.92, … , S_lowerlimit]`. If you selected ‘no’ in ⑪, then the SOC division (`S_j`) will be `[0, 0.04, 0.08, … , S_upperlimit]`.

```
% Specify the lower and upper limits for SOC
SOC_param_lowlimit = str2num(evalin('base','SOC_param_lowlimit'));
if isempty(SOC_param_lowlimit) == 0
    S_j = S_j(find(S_j > SOC_param_lowlimit));
    S_j = (vertcat(SOC_param_lowlimit,S_j'))';
    bounds.lowerbound = SOC_param_lowlimit;
end

SOC_param_uplimit = str2num(evalin('base','SOC_param_uplimit'));
if isempty(SOC_param_uplimit) == 0
    S_j = S_j(find(S_j < SOC_param_uplimit));
    S_j = (vertcat(S_j',SOC_param_uplimit))';
    bounds.upperbound = SOC_param_uplimit;
end
```

The codes above are to crop the `S_j` based on the user’s in ⑯.

```
flags.R0temp.val = [0,100];  % read off temperatures for R0
```

The above line of codes came from an earlier version and was used for T-dependent parameterisation. However, in the latest version T-dependent parameterisation was realised in another way so this line has **no effect** on anything of parametrisation. It is kept here because the declared '`flags.R0temp.val`' still exists in fullsolve.m [in other words, the developer is too lazy to remove this variable].

```
%% Switch - If current and/or temperature dependence(s) needed, codes will be
redirected
if flags.AmpDependence.tf == 1 && flags.TempDependence.tf == 0
    AmpDenpOnly
else
    if flags.AmpDependence.tf == 0 && flags.TempDependence.tf == 1
        TempDenpOnly
    else
        if flags.AmpDependence.tf == 1 && flags.TempDependence.tf == 1
            AmpAndTempDenp
        else
%% From here down, the codes only work for the case where neither current nor
temperature dependence is considered
...
...
```

Apart from SOC-dependent parameterisation, BatPar can incorporate I-dependent and/or T-dependent parameterisation according to the user's demand (as configured in ㉕ to ㉚), and the codes above are to divert the entry of different dependences. If I-dependence is required by the user only, then **BatPar will jump to** `AmpDenpOnly` **and all the remaining codes within ECM_easy.m will be skipped**, and the parameters to be extracted will be SOC- and I-dependent. Similarly, if T-dependence is required only, `TempDenpOnly` will be the script to be executed instead of the remaining codes within ECM_easy.m, and the parameters to be extracted will be SOC- and T-dependent. `AmpAndTempDenp` is the script to handle both I-dependence and T-dependent parameterisation along with SOC-dependence. If neither I- or T-dependence is required by the user, the remaining codes will be executed, which means that the parameters to be extracted will be SOC-dependent only.

```
if flags.taufixed_Stage2.tf == 0
    disp(['Initialisation done - Paramterisation (with VARIABLE tau, without I or
    T dependence) in progress, including ',num2str(numel(S_j)),' STEPS.'])

    disp('************************************************************************
    ***************************')
```

```
else
    disp(['Initialisation done - Paramterisation (with constant tau, without I or
    T dependence) in progress, including 2 STAGES --> Each STAGE further includes
    ',num2str(numel(S_j)),' STEPS.'])

    disp('************************************************************************
    ***************************')
    disp('--> Proceeding to Stage 1 - solving parameters with VARIABLE tau.')
end
%% Iterate over LUT - Stage 1 - solving wtih variable tau
...
...
%% Recommend fixed tau constants
tau_1_reco = prctile (tau(1,:),50);
if NRC >= 2
tau_2_reco = prctile (tau(2,:),50);
else
tau_2_reco = [];
end
if NRC >= 3
tau_3_reco = prctile (tau(3,:),50);
else
tau_3_reco = [];
end
tau_recommended =[tau_1_reco, tau_2_reco,tau_3_reco]; % recommend fixed tau for
rerunning the codes
...
...
%% Stage 2 - solving fixed tau
    disp('--> Proceeding to Stage 2 - solving parameters with CONSTANT tau')
...
...
%%
Ci = tau ./ R; % calculate Ci
...
...
```

BatPar solves the RC pairs by TAUs and Rs first and then Cs are indirectly solved by `Ci = tau ./ R`. Rs are treated as variables with SOC, while TAUs can be either variables or constants – the user can specify this by ⑭. If TAUs are considered as variables, then there will be one single stage (Stage 1) to be executed. After Stage 1 execution, **fixed TAU constants can be found out by 'averaging' the variable TAUs at all SOCs** – `prctile` is used to perform averaging. If TAUs are required to be constants, the averaged TAUs will be considered as the constants and used in Stage 2, where the Rs will be solved again with the TAU constants and the R0 solved in Stage 1 substituted in.

```
%%% The show starts
for ii = 1:numel(S_j)
      disp(['     Step ',num2str(ii),' of ',num2str(numel(S_j)),', parameterising
      at  SOC ', num2str(S_j(ii)),])  % display the progress of parameterisation
...
```

```
...
    [OCV(ii),R0(ii),R(:,ii),tau(:,ii), Diff, Skip ] =
    easyECMfit(t_,I_,V_,T_,S_,NRC,S_j(ii),SOCTab,OCVTab,SOCSOCTab,R0R0Tab,flags,Cc
    ap,working_mode,R,tau);
```

In each of Stage 1 and Stage 2, you can find similar lines of codes as above. The for loop is used to solve parameters at all SOCs successively, and **`easyECMfit` is the actual function with an optimisation algorithm to solve the parameters** `[OCV(ii),R0(ii),R(:,ii),tau(:,ii)]`. Besides, `Diff` is the difference between experimental voltage and fitting model voltage, `Skip` is an indicator advising whether parameterisation at the current SOC should be skipped – due to insufficient data around the current SOC.

```
%% OCV and R0 check
%%% monotonicity detection for OCV
...
...
%%% outlier detection for R0
Outlier_ind = isoutlier(R0,'movmedian',3);  % Hampel filter --> necessary but not
sufficient
```

After solving the parameters, OCV will be checked by its monotonicity and R0 will be checked by its outliers. Currently the method for R0 outlier check is a simple Hampel filter using `'movmedian'` command – apparently this method can be replaced with better one.

```
% Overall horizon - Experiment VS Paramterised Model
...
...
if working_mode == 3 || working_mode == 4 % discharge
...
...
   plot(N(:,1), N(:,3), 'LineWidth',1);
...
...
   for kk = 1:numel(t)
    [Vhat,t_useful]=fullsolve(t{kk},I{kk},T{kk},S{kk},S_j,OCV,R0,R,tau,flags,NRC);
    if isnan(Vhat)
       DIFF_global{kk,1} = nan;
    else
       aaa= find (N(:,1) == t_useful(1)); bbb= find (N(:,1) == t_useful(end));
       if working_mode == 3 % anode
          if anode_voltage_sign < 0
           DIFF_global{kk,1} = Vhat - N(aaa:bbb,3);  % Global difference
...
...
```

The codes above are to plot the global comparison figure, as Fig. 9., and simultaneously calculate the difference (`DIFF_global`) between experimental voltage (`N(:,3)`) and fitting model voltage (`Vhat`). Noted that the plotting method and difference calculation method are specific to the configuration of parameterisation (discharge/charge + full cell/cathode/anode). `fullsolve` is the actual function to calculate the fitting model voltage (`Vhat`) over the timeline of the datasheet (`N(:,1)`), and `t_useful` is a filtered timeline based on `N(:,1)`, by only keeping the time periods that have useful data for comparison.

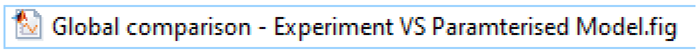

Global comparison - Experiment VS Paramterised Model.fig

Fig. 9. The global comparison plot (in result folder).

```
% generate a look-up table including every parameter, in the format of .xlsx
LookUpTable_AllInOne = zeros(length(S_j),12)*nan;
LookUpTable_AllInOne (:,1) = S_j';
LookUpTable_AllInOne (:,2) = OCV';
LookUpTable_AllInOne (:,3) = R0';
[good,~] = size(R); [luck,~] = size(tau);
LookUpTable_AllInOne (:,4:3+good) = R';
LookUpTable_AllInOne (:,7:6+luck) = Ci';
LookUpTable_AllInOne (:,10:9+luck) = tau';
LookUpTable_AllInOne = mat2cell(LookUpTable_AllInOne,ones(1,length(S_j)),[1 1 1 1
1 1 1 1 1 1 1 1]);
title = {'SOC','OCV','R0','R1','R2','R3','C1','C2','C3','tau1','tau2','tau3'};
LookUpTable_AllInOne = [title;LookUpTable_AllInOne];
writecell(LookUpTable_AllInOne,'LookUpTable_AllInOne.xlsx');
...
...
evalin('base', 'save(''INPUT.mat'')');
save OUTPUT
toc
```

The codes above are to save the parameterisaion results – parameter look-up table in `'LookUpTable_AllInOne.xlsx'`, GUI input variables in `'INPUT.mat'`, and non-input variables in `'OUTPUT.mat'`. **`'INPUT.mat'` and `'OUTPUT.mat'` are places to check when you observe suspicious parameterisation results and want to find out the reasons**. `toc` is the command to end the timer and the elapsed time is calculated by the combo of `tic-toc`.

3.3 Places of future improvement

In a meeting with Tom Holland, two places of improvement were identified:

1> SOC division is based on the user's choice of SOC window size, but this can tear apart a whole pulse (+relaxation period) into fragments lying in different SOC windows. In future improvement, the codes can take over the decision of SOC division from the user, and make pulse-specific division so that integrity of pulses can be assured – this is believed to offer parameters with more physical senses.

2> Prior solving of variable TAUs (Stage 1) is needed before solving fixed TAU constants (Stage 2). However, once Stage 2 is completed, the codes use parameters under fixed

TAUs to overwrite the parameters under variable TAUs – the parameters under variable TAUs (solved in Stage 1) can still be saved as an extra look-up table – they are still useful.

4. *loadexperiments_.m*

4.1 Functionality and structure

loadexperiments_.m, called by ECM_easy.m, is a script to read and pre-process the experimental datasheet and user's configurations of parameterisation. Therefore, understanding loadexperiments_.m in full is vital for linking the experimental datasheets with parametrisation program. More details are given in Section 4.2.

4.2 Core codes and hard-to-read codes

```
%% mode selection
...
...
    if strcmp(flags.anode_mode.tf,'yes')
        working_mode = 3;   % mode 3: anode - discharge
    else
        working_mode = 4;   % mode 4: full cell/cathode - discharge
    end
...
...
```

As mentioned in Section 3.2, the methods of parameterisation and post-processing are specific to configurations of discharge/charge + full cell/cathode/anode. `working_mode` is hereby defined to distinguish 4 kinds of configurations, as the matrix below. Noted that **cathode parameterisation uses nearly the same methods as full cell parameterisation**. Using `working_mode` as an indicator, BatPar decides on which codes/scripts/functions to execute.

$$\begin{bmatrix}\text{full cell/cathode}\\ \text{anode}\end{bmatrix} \times \begin{bmatrix}\text{discharge}\\ \text{charge}\end{bmatrix} = 4\; working\ modes$$

```
%% More switches
...
...
flags.taufixed_Stage2.tf = evalin('base','fixedtau_realy_tf');
...
...
```

`flags` is a struct with many binary attributes that indicate the true or false of configurations. E.g., the attribute `flags.taufixed_Stage2.tf` indicates whether fixed TAUs or variable TAUs should be worked out. The value of `flags.xxx.tf` can only be a string of either `'yes'` or `'no'` - until yesnoconvert.m is called, after which the value will be a number of either 1 or 0.

```
%% Initialisation - read and sort data
```

```
M = xlsread(evalin('base','datasheet_name'));
M = M (~isnan(M(:,1)),:);
N =
M(:,[str2num(evalin('base','time_column')),str2num(evalin('base','current_column')
),str2num(evalin('base','voltage_column')),str2num(evalin('base','record_column'))
,str2num(evalin('base','temp_column'))]); % keep 5 columns - time, current,
voltage, Rec#, temperature
[~,ind,~] = unique(N(:,1));
N = N(ind,:);                 % keep data at unique times
```

The codes above are to read the experimental datasheet and pass it to `M`. The **5 columns of time, current, voltage, record number, temperature** will be kept and further passed to `N` – in other words, no matter how many columns your datasheets have, the useful columns are the 5 columns only.

```
flags.datasheet_cleanup.tf = evalin('base','datasheet_cleanup_tf');
if strcmp(flags.datasheet_cleanup.tf,'yes')
    exp_start_line = str2num(evalin('base','row_start')); % the line number in N,
where the experiment formally strats
    exp_end_line = str2num(evalin('base','row_end')); % the line number in N,
where the experiment formally ends
    if isempty(exp_start_line)
        exp_start_line = 1;
    else
        exp_start_line = find (N(:,4) == exp_start_line);
    end
    if isempty(exp_end_line)
        [rowsN,columnsN]=size (N);
        exp_end_line = rowsN;
    else
        exp_end_line = find (N(:,4) == exp_end_line);
    end
else
    exp_start_line = 1;
    [rowsN,columnsN]=size (N);
    exp_end_line = rowsN;
end
```

Based on the user's inputs in ⑰ to ⑲, the above codes crop `N` by indexing the start and end line numbers to the `'record_column'` (`N(:,4)`). This functionality is especially useful when the original datasheet contains data of multiple experiments, but the user only wants to pick up one of the experiments to perform parameterisation. This is why the `'record_column'` is kept and passed to `N(:,4)`.

```
flags.dischargecurrentsubzero.tf = evalin('base','Current_belwozero_tf');
if strcmp(flags.dischargecurrentsubzero.tf,'yes')
    Current_sign = -1;
else
    Current_sign = 1;
```

```
end
t_ = N(exp_start_line:exp_end_line,1);
I_ = Current_sign * N(exp_start_line:exp_end_line,2);
if working_mode == 1 || working_mode == 3 % anode paramterisation
    anode_voltage_sign = mean(N(exp_start_line:exp_end_line,3));
    if anode_voltage_sign < 0 % input andoe voltage is mostly below zero
    V_ = N(exp_start_line:exp_end_line,3);
    else  % input andoe voltage is mostly above zero
    V_ = - N(exp_start_line:exp_end_line,3);
    end
else
    V_ = N(exp_start_line:exp_end_line,3);
end
T_ = N(exp_start_line:exp_end_line,5);
```

According to the user's input in ⑦, `Current_sign` is assigned. After that, the 4 columns of time, current, voltage and temperature will be extracted from `N` and passed to 4 new vectors of `t_`, `I_`, `V_`, and `T_`. The column of record number will not be kept anymore, because its mission of indexing line numbers has been accomplished and it is not useful anymore. One last thing to be noted is that **in anode parameterisation, anode voltage must be below zero, because the fitting model (in localsolve.m and fullsolve.m) takes anode voltage as below zero**. This is why a couple of lines of codes are put above to deal with the sign of `V_`.

```
%% SOC calculation and processing
Ccap = str2num(evalin('base','Nom_capacity'))*3600;
S_ = - cumtrapz(t_,I_/Ccap);  % delta SOC from time zero; calculated by coulomb
counting
bounds.upperbound = nan;  % the struct to save upper and lower bounds of SOC; to
be used in datasplitter.m
bounds.lowerbound = nan;
flags.MaxSOCis1.tf = evalin('base','MaxSOCis1'); % 100% SOC is the baseline
if strcmp(flags.MaxSOCis1.tf,'yes')
    S_real = S_ + (1-max(S_)); % Real SOC at each time; (1-max(S_))=initial SOC;
    S_= S_real;
    % S_(find(S_<0)) = 0; % Real SOC filters out minus values
    S_lowerbound = min(S_); % minimum SOC to be parameterised
    bounds.lowerbound = S_lowerbound;
else  % 0% SOC is the baseline
    S_real = S_ + (0-min(S_)); % Real SOC at each time; (0-min(S_))=initial SOC;
    S_= S_real;
    % S_(find(S_>1)) = 1; % Real SOC filters out values that outnumber 1
    S_upperbound = max(S_); % minimum SOC appeared in experiment
    bounds.upperbound = S_upperbound;
end
```

Assume that the experimental datasheets do not include a column of SOC (because the battery cyclers do not calculate/log SOC), SOC over the timeline of `t_` is calculated by the codes above by the method of coulomb counting. As mentioned in Section 3.2, **100% SOC is considered as the ceil (max SOC) in discharge experiments** – we cannot use 0% SOC as the floor because the battery can hardly be discharged to 0% SOC by high-rate current pulses. In contrast, **0% SOC is considered as the floor (min SOC) in charge experiments** – we cannot use 100% SOC as the ceil because the battery can hardly be charged to 100% SOC by high-rate current pulses.

```
%% optional - import SOC - OCV table
...
...
```

According to the user's inputs in ⑳ and ㉑, BatPar can import prescribed SOC-OCV table and thus skip OCV parameterisation. To successfully use this functionality, please put your SOC-OCV table in the folder 'SOC_OCV_table', as Fig. 1. Why this functionality is useful? We know that OCV can be directly measured from experiments – so why bother parameterising OCV, if OCV can be directly measured. Also, you may believe that the experimentally measured OCV should be more accurate than BatPar-parameterised OCV, so the measured OCV may be your choice, even though comparisons (done by the developer) indicated the gap between two OCVs was small (some 50 mV). In summary, please use this functionality if you are a perfectionist and picky about OCV accuracy.

```
%% optional - import SOC - R0 table
...
...
```

According to the user's inputs in ㉒ and ㉓, BatPar can import prescribed SOC-R0 table and thus skip R0 parameterisation. To successfully use this functionality, please put your SOC-OCV table in the folder 'SOC_R0_table', as Fig. 1. Why this functionality is useful? We know that theoretically, R0 changes with temperatures and C-rates, but not with SOC. However, the BatPar-parameterised R0 always change with SOC – so if you believe the changing R0 contradicts with theoretical physics, you can use the 'SOC_R0_table' to set up fixed R0 and then parameterise the other Rs and Cs. In summary, this functionality enables you to tailor R0 trace with SOC.

---

5. *dataextractor.m*

5.1 Functionality and structure

dataextractor.m, called by loadexperiments.m, is a function to recognise and extract the discharge and charge segments, respectively, given that one datasheet may contain data of both discharge and charge.

Even though most battery parameterisation tests are targeted either discharge or charge, but in some cases, the test protocols and the logged datasheets can be very complicated, with highly mixed, alternated discharge and charge segments. In these cases, manually sorting out the discharge or charge segments can be great trouble to the user - dataextractor.m is here to do the sort-out work instead of the user.

In dataextractor.m, SOC is used as the indicator of discharge or charge – increase of SOC comes with charge and decrease of SOC comes with discharge. Sounds easy? Actually there are more to carefully consider in code implementation, which are detailed in Section 5.2.

5.2 Core codes and hard-to-read codes

```
function [t, I, V, S, T, t_charge, I_charge, V_charge, S_charge, T_charge] =
dataextractor (t_, I_, V_, S_, T_)
...
...
```

This function accepts time, current, voltage, SOC, and T (`t_, I_, V_, S_, T_`) of the whole datasheet as inputs, and returns those of discharge segments (`t, I, V, S, T`) and of charge segments (`t_charge, I_charge, V_charge, S_charge, T_charge`). **Noted that these discharge and charged variables here are 'cells' - 'cell' is a data type in MATLAB and is different from 'matrix'**.

```
[~,maxlocs] = findpeaks(S_); % find values and locations of SOC local maximums
(rising edge)
[minvals,minlocs] = findpeaks(-S_);
minvals = -minvals;  % find values and locations of SOC local minimums (dropping
edge)
...
...
        for pulse_head = 1:length(S_)-1  % find pulse_head - where the discharge
        pulse started
          diffS = S_(pulse_head+1)-S_(pulse_head);
          if diffS ~= 0
              break
          end
         end
```

```
...
...
for mm = 1:length(maxlocs) % shift the locations of SOC local maximums from rising
edge to dropping edge
    for nn = maxlocs(mm):length(S_)-1
       diffS = S_(nn+1)-S_(maxlocs(mm));
       if diffS ~= 0
           maxlocs(mm) = nn;
           break
       end
    end
end
```

`findpeaks` command is used to find out the local maxima and minima of SOC – then these peaks can be used as breakpoints to divide discharge and charge segments. E.g., one maxima and its nearest following minima delimit a discharge segment in between. However, due to the **limitations of `findpeaks`**, two more adaptations are implemented - Fig. 10. helps understand the limitations and adaptations.

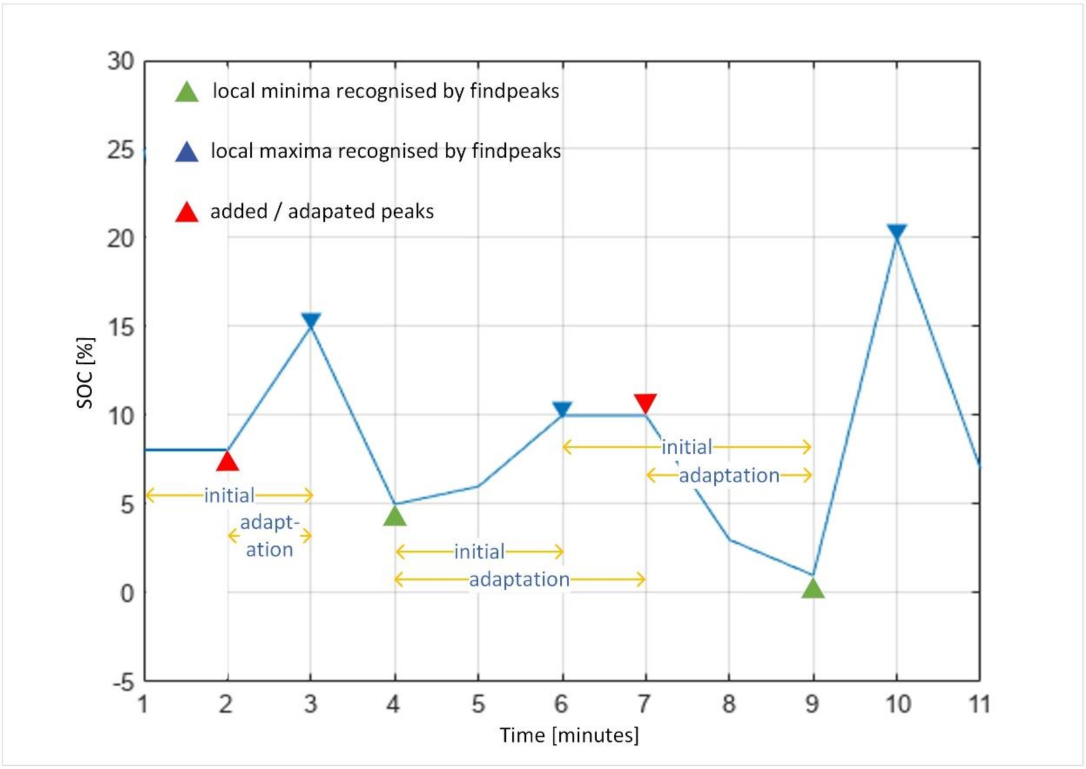


Fig. 10. Understanding the limitations of `findpeaks`, and why adaptations are implemented.

(1) The first limitation is that `findpeaks` cannot recognise the starting point of experiment, where the current started to change from zero. In Fig. 10, this starting point exists at Time 2, where the SOC started to change, as a result of current change. In dataextractor.m, this starting point is claimed as ‘`pulse_head`’, and there is a for-if-break loop to recognise it. Recognising this starting point is essential, because in some cases there can be a very long

period before the starting point. However, this period, delimited by the starting point, contains no useful data for parameterisation and thus should be removed.

(2) The second limitation is that `findpeaks` returns the leading edge of a 'plateau', and this is reflected at Time 6 in Fig. 10. However, it is the following edge (at Time 7 in Fig. 10) rather than the leading edge that should be recognised. This is because **a complete pulse is considered as the pulse itself plus the following relaxation period, rather than plus the relaxation period before the pulse**. In other words, the dynamic voltage and current behaviours during one relaxation are a result of the pulse before the relaxation. In dataextractor.m, after `findpeaks` returns a leading edge, the nearest following edge can be then recognised by a for-for-if-break loop.

As you may be aware that dataextractor.m is a long script with 280 lines of codes – why does this script need to be such long and what are the codes doing? Since the local maxima and local minima are expected to be recognised first and then used to delimit discharge/charge segments, there can be special cases – e.g., in experiments that contain one single discharge/charge segment, `findpeaks` will return no local maxima or local minima; in experiments that contain multiple discharge/charge segments, the number of local maxima can be the same as the number of local minima, or there can be one less or on more. dataextractor.m takes into account all the possible cases, as Fig. 11, and carefully tailors the extraction method for each case.

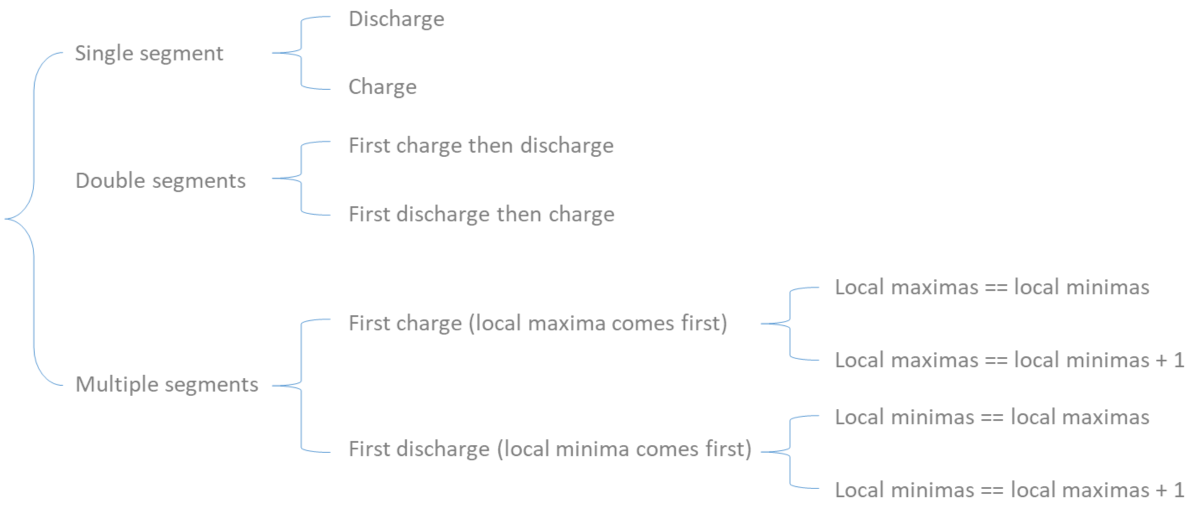

Single segment
Discharge
Charge
Double segments
First charge then discharge
First discharge then charge
Multiple segments
First charge (local maxima comes first)
Local maximas == local minimas
Local maximas == local minimas + 1
First discharge (local minima comes first)
Local minimas == local maximas
Local minimas == local maximas + 1

Fig. 11. All the possible cases of discharge-charge combination in one experiment, which are considered in extraction of discharge or charge segments.

----

6. *datasplitter.m & cleardown.m & yesnoconvert.m*

6.1 Functionality and structure

datasplitter.m, called by ECM_easy.m, is a function to sort out data segments lying in each SOC window. In other words, this script breaks down the originally time-dependent data segments and reforms SOC-dependent data segments, which will be used to solve SOC-dependent parameters.

cleandown.m, called by ECM_easy.m, is a function to recognise the length of each SOC-dependent data segment and accordingly discard data segments that are too short to be used to solve parameters.

yesnoconvert.m, called by ECM_easy.m, is a function to convert the values of flags.xxx.tf from `'yes'` or `'no'` to 1 or 0.

6.2 Core codes and hard-to-read codes

---

datasplitter.m

```
function [data,S_j] = datasplitter(t,I,V,T,S,S_j,bounds,flags)
OPT_factor = 1;
for ii = 1:numel(S_j)
        if strcmp(flags.MaxSOCis1.tf,'yes')  % 100% SOC reached
           if (ii-1==0)
               S_pesudobound = S_j(ii) - (S_j(ii+2) - S_j(ii+1))/OPT_factor;

               if bounds.lowerbound > S_pesudobound
                  lowerbound = bounds.lowerbound;
               else
                  lowerbound = S_pesudobound;
               end
           else
               lowerbound = S_j(ii) - (S_j(ii) - S_j(ii-1))/OPT_factor;
           end
           if (ii+1>numel(S_j))
               ...
               ...
           else
               upperbound = S_j(ii) + (S_j(ii+1)-S_j(ii))/OPT_factor;
           end
        else  % 0% SOC reached
           if (ii-1==0)
               ...
               ...
           else
               lowerbound = S_j(ii) - (S_j(ii) - S_j(ii-1))/OPT_factor;
           end
           if (ii+1>numel(S_j))
               if (ii+1>numel(S_j))
                  S_pesudobound = S_j(ii) + (S_j(ii-1) - S_j(ii-2))/OPT_factor;
```

```
                if bounds.upperbound < S_pesudobound
                   upperbound = bounds.upperbound;
                else
                   upperbound = S_pesudobound;
                end
            else
                upperbound = S_j(ii) + (S_j(ii+1)-S_j(ii))/OPT_factor;
            end
            else
                upperbound = S_j(ii) + (S_j(ii+1)-S_j(ii))/OPT_factor;
            end
         end
        % Find indices to find data in the given window
        ...
        ...
```

datasplitter accepts the discharge or charge data segments (`t,I,V,T,S,` sorted by dataextractor.m), original SOC division (`S_j`), actual SOC bounds (`bounds`, solved by ECM_easy.m) and `flags`, and returns the SOC-dependent data segments (`data`) along with an updated SOC division (`S_j`, updated by removing SOC windows of no data, as the experiment sometimes may not cover all the divided SOC windows).

`OPT_factor` is a division factor determining the actual length of each SOC window. **This factor was once handed over to the user to determine, but in the current version of BatPar its value is fixed as 1 – based on massive observations of using different factor values ranging from 0.5 to 2**. Fig. 12. gives more explanation of the effect of `OPT_factor`. Let's say you put 0.05 as the SOC window size, so parameters will be solved at SOC = […, 0.7, 0.75, 0.8, 0.85, 0.9, …]. Now assume that the parameterisation process is proceeding to SOC = 0.8. If `OPT_factor =1`, then the data segments between SOC = 0.75 and SOC = 0.85 will be collected and used for solving parameters at SOC = 0.8. If `OPT_factor =0.5`, then the data segments between 0.7 and 0.9 will be used. Namely, the larger `OPT_factor`, the less data segments used. Why is `OPT_factor` important, or in other words, why does the actual length of SOC window matter? **This refers to the dilemma between underfitting and overfitting.** If you have a very narrow SOC window containing very few data segments, the solved parameters can be easily 'abducted' by the 'bad' data - as we know experimental data is not always perfect and sometimes noise measurement and wrong logging are unavoidable. In contrast, if you have a very broad SOC window containing too many data segments, the parameters will be very much averaged over the broad window and therefore lose much fidelity to the targeted SOC (e.g., SOC = 0.8). The optimal length of SOC window can be investigated and improved in the future.

In the codes above, there are lots of if-else things – they are used to determine the actual length of SOC window by considering special cases. For example, for discharge parameterisation at SOC = 1, the corresponding SOC window will be between SOC = 0.95

and SOC = 1. Another special case in discharge parameterisation is about the actual minimum SOC – this is why `bounds` is passed into datasplitter.m for evaluation.

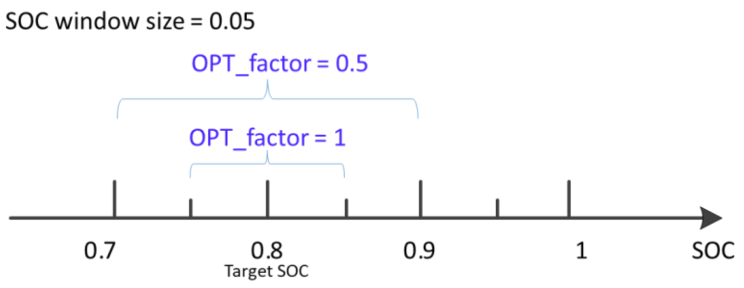


Fig. 12. Explanation of the effect of `OPT_factor` on actual length of SOC window.

----

datasplitter.m

```
for ijk = 1:numel(t) % for each targeted SOC
           ind1 = (lowerbound <= S{ijk}) + (S{ijk} <= upperbound) == 2;
          ...
          ...
```

After getting `lowerbound` and `upperbound` for each SOC window, they can be used to crop the data of SOC (`S`) and as such obtaining the index of the other data (`ind1` of `t,I,V,T`).

```
% data check --> filter out empty data clusters
count = 0;
datainsufficient = zeros(1,numel(S_j));
for zhu = 1:numel(S_j)
    if isempty (data.t{zhu})
        datainsufficient (zhu) = 1;
        count = count + 1;
    end
end
if count ~= 0
    datainsufficient = logical(datainsufficient);
    S_j              = S_j(~datainsufficient);
    data.t           = data.t(~datainsufficient);
    data.I           = data.I(~datainsufficient);
    data.V           = data.V(~datainsufficient);
    data.T           = data.T(~datainsufficient);
    data.S           = data.S(~datainsufficient);
    data.label       = data.label(~datainsufficient);
end
```

The codes above are used to recognise and remove empty SOC windows - no data segments can be allocated to the SOC windows because the experimental data of charge/discharge does not cover the SOC windows.

----

cleandown.m

```
function data = cleandown(data,NRC)
...
...
ind = cellfun('length',data.t{iii})>= (NRC * 3 + 2) * 2;

    % Why (NRC * 3 + 2) * 2? Because:
    % Firstly, there are (NRC * 3 + 2) unknowns in the ECM, as each RC pair has
    three unknowns - Ri, tau_i and V0_i, and the ECM has two more unknowns - OCV
    and R0
    % Secondly, * 2 is a safety coefficient to ensure a unique solution. i.e., the
    solver (fmincon) is fed with one more fold data than necessary
...
...
```

After executing datasplitter.m, we will have at least one data segment allocated for each targeted SOC. Here comes a problem - we don't want data segment that is too short, because it may not set up enough equations to solve the parameters (unknowns). For example, **there are (n*3 + 2) unknowns in a nth order RC circuit, so we need at least (n*3 + 2) equations to solve them, which means the data segment should contain at least (n*3 + 2) elements**. Beyond that, more elements can better enhance robustness of solved parameters, so a multiplication factor of 2 is further applied. In summary, the number of elements within a data segment should be no less than (n * 3 + 2) * 2.

6.3 Places of future improvement

The developer identified two places of improvement:

1> As aforementioned, in datasplitter.m the `OPT_factor` can be optimised rather than using an empirical fixed value of 1.

2> yesnoconvert.m can be indeed removed from BatPar, as it does nothing but converting data types, and the conversion is not essential. It also causes confusion to incorporation of new functionalities because the developer needs to be very careful with the data type of `flags.xxx.tf` (`'yes'`/ `'no'` or 0/1, mentioned in Section 4.2). Also because of this, removing yesnoconvert.m needs massive changes in many scripts and functions accordingly.

7. *easyECMfit.m*

7.1 Functionality and structure

easyECMfit.m, called by ECM_easy.m, AmpDenpOnly.m, TempDenpOnly.m, and AmpAndTempDenp.m, is the actual script to solve parameters. This script is the most difficult-to-understand one among all the scripts and functions, as it embeds an optimisation algorithm into battery circuit representations. Understanding easyECMfit.m in full is vital for incorporating new functionalities. More details are given in Section 7.2.

7.2 Core codes and hard-to-read codes

```
function [OCV,R0,R,tau,DIFF_,skip_] =
easyECMfit(t,I,V,T,S,NRC,S_j,SOCTab,OCVTab,SOCSOCTab,R0R0Tab,flags,Ccap,working_mo
de,R,tau)
```

This function accepts lots of inputs to fit different parameterisation configurations. `t,I,V,T,S` are the data segments previously sorted as specific to one certain SOC (`S_j`). `NRC` is the number of RC pairs. `SOCTab,OCVTab,SOCSOCTab,R0R0Tab` come from loadexperiments_.m and are the prescribed SOC-OCV table and SOC-R0 table, if prescribed indeed; otherwise, these tables will be empty (NaN). `Ccap` is battery capacity in Ampere*Second. **`R` and `tau` exist on both the input side and output side of easyECMfit.m**. On the input side, `R` and `tau` are the Rs and TAUs solved in all the previous calculations (at all the previous SOCs) – easyECMfit.m is put under a for loop of SOC, so `R` and `tau` solved at all the previous SOCs are used as inputs to assist the calculation at the current SOC (`S_j`). On the output side, `R` and `tau` are the Rs and TAUs to be solved at the current SOC (`S_j`). `DIFF_` is a vector logging the difference between experiment and parameterised model in terms of terminal voltage at around the current SOC (`S_j`). `skip_` is an indicator of data sufficiency specifically regarding R0 solving – in some cases the experimental data segments fed to easyECMfit.m are poor in quality and/or in quantity, in which cases no valid R0 can be worked out. In these cases the value of `skip_` will be 1 – as a result, parameterisation at the current SOC cannot work out and will be skipped.

```
N = numel(t);                      % number of data segments in this window
if N == 0 % no data
    OCV = nan;
    R0 = nan;
    R = nan(NRC,1);
    tau = nan(NRC,1);
    DIFF_ = nan;
    skip_ = 1;
    return
end
```

As abovementioned, if the experiment data segments are poor in quantity (`N == 0`), `skip_` will be 1 and all the parameters will be NaN. After those, execution of easyECMfit.m will be terminated (`return`).

```
%% Calculate R0 from jumps by dV/dI
if  flags.taufix.tf  == 1  % Stage 2 enforcing; R0 and tau were already solved in 
Stage 1
    R0 = flags.R0solved.val;
else                              % Stage 1 enforcing;
    if (flags.SOCR0table.tf == 1)    % the SOC-R0 table exists
        if S_j >= min(SOCSOCTab) && S_j <= max(SOCSOCTab)
        R0  =  interp1(SOCSOCTab,R0R0Tab,S_j);
        end
    else
        R0_ini = cell(N,1); % initialise R0
        for kk = 1:N  % for each segment%
            I_R0 = I{kk};
            V_R0 = V{kk};
            dI = diff(I_R0);
            dV = diff(V_R0);
            if working_mode == 3 || working_mode == 4  % discharge
               if flags.pulsehead4R0.tf == 1
                    index = find (dI >  Ccap/3600 * 0.15);
               else
                    index = find (dI < - Ccap/3600 * 0.15);
               end
            else              % charge
               if flags.pulsehead4R0.tf == 1
                    index = find (dI < - Ccap/3600 * 0.15);
               else
                    index = find (dI > Ccap/3600 * 0.15);
               end
            end
            R0_ini{kk} = abs(dV(index)./dI(index));
        end
            R0_ini = cell2mat(R0_ini);  % cell to matrix
            R0_ini = R0_ini(R0_ini ~= 0);    % filter out zero
            R0_ini = R0_ini(~isnan(R0_ini)); % filter out NaN
            R0 = mean(rmoutliers(R0_ini)); % filter out outliers and then average
    end
end
```

The codes above are for R0 calculation. **It should be highlighted that in BatPar, R0 calculation is independent of calculations of all the other parameters – R0 is first worked out and then used as a known condition to help solve the other parameters.**

The above codes are explained in an inside-out manner. R0 is solved by `dV/dI` – **it is assumed that when battery current jumps from zero to nonzero (<u>pulse head</u>), or from**

nonzero to zero (<u>pulse end</u>), the dV and dI be can significant enough to get rid of the influence of noises from the battery cycler. To be more specific, **BatPar uses either pulse head or pulse end ⑮, rather than the mix of both, to calculate R0. Based on massive observations of using pulse head only, pulse end only, or both pulse head and end, it is found that towards getting smoother R0 with SOC, the priority should be (1) pulse head > (2) pulse end > (3) both pulse head and end**. **Namely, where possible, you should always use pulse head to perform R0 calculation**. The reason is: from a physical point of view, pulse head usually comes after a long relaxation, so we can believe that the dV **and** dI are purely caused by R0. In contrast, pulse end usually comes as the end of a pulse, by which battery physics has been significantly influenced by charge transfer and diffusion, so the dV **and** dI are not only caused by R0 but unavoidably influence by the Rs.

If it is discharge parameterisation, pulse heads are recognised by `index = find (dI > Ccap/3600 * 0.15)` – this assumes that the `dI` at pulse head should be larger than 0.15C; in other words, any `dI` that is smaller than 0.15C is considered as noise and as such is NOT used to calculate R0. **However, there can be cases in which data at pulse head is of bad quality (e.g., due to battery cycler incompetence, battery current failed to change from zero to the target current within one time step), then you should use pulse end instead to calculate R0, as long as data at pulse end is of good quality (battery current successfully dropped to zero directly without any transition).** In discharge parameterisation, pulse ends are recognised by `index = find (dI < - Ccap/3600 * 0.15).` If neither pulse head nor pulse end is of good quality, please remember the brand of the battery cycler used and put it in your blacklist.

There are `N` data segments, and each data segment can enable a calculated R0. All the calculated R0s from all the data segments will be averaged (`mean`) and then considered as the final R0 at the present SOC. Before averaging, three kinds of pro-processing are performed: (1) `R0_ini = R0_ini(R0_ini ~= 0); % filter out zero` This targets the cases in which `dV = 0`; (2) `R0_ini = R0_ini(~isnan(R0_ini)); % filter out NaN` This targets the cases in which `dI = 0`; (3) `rmoutliers(R0_ini); % filter out outliers` In case there are significant outliers.

Furthermore, there are two cases in which R0 calculation will be skipped. Case 1 is when a prescribed R0 table is imported, as configured in the GUI by ㉒ ㉓ and reflected in the codes by (`flags.SOCR0table.tf == 1`). Case 2 is in Stage 2 – solving fixed tau, where R0 calculation will be skipped because R0 has already been solved in Stage 1 - solving variable tau, and saved in the variable `flags.R0solved.val`. As abovementioned, R0 is independent from all the other parameters, so R0 don't need to be calculated again.

```
if isnan(R0)
      OCV = nan;
      R0 = nan;
      R = nan(NRC,1);
      tau = nan(NRC,1);
      DIFF_ = nan;
      skip_ = 1;
      return
end

if flags.R0only.tf == 1  % Parameterisation of R0 ONLY
      OCV = nan;
      R = nan(NRC,1);
      tau = nan(NRC,1);
      DIFF_ = nan;
      return
end
```

If the solved R0 is NaN (`isnan(R0)`) or BatPar is parameterising R0 only (`flags.R0only.tf == 1`), all of the Rs, TAUs and differences will not be solved anymore but be assigned with NaN.

```
%% Formulate an optimisation problem to fit in fmincon - to solove OCV, Ri and
Tau_i

% OCV1    OCV2      Ri          tau_i            V0_i
% 1        2      3~2+NRC    3+NRC~2+2*NRC     3+2*NRC~2+2*NRC+N*NRC

% fmincon: https://uk.mathworks.com/help/optim/ug/fmincon.html
% [x,fval] = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)

% param0 is 'x0'- intial values of unkonwns;
param0 = zeros(2+(2+N)*NRC,1);  % 2{OCV} + 2{Ri & tau_i} *NRC + N{V0_i of each
segment} *NRC
param0(3+NRC:2+2*NRC) = 0;    % tau_i=5
param0(3:2+NRC) = 0;      % R_i=0.001
...
...
```

This section of codes configures the arguments of 'fmincon' algorithm that solves the unknowns of OCV, Rs and TAUs by minimising the cost function formulated (the RSME between fitting model voltage and experimental cell voltage). Before going further into the codes, it is strongly recommended to read the documentation of 'fmincon' at `https://uk.mathworks.com/help/optim/ug/fmincon.html`. More explanations of the arguments are given below.

All the unknowns are put together in one array - '`param`', so '`param0`' is the initial values (i.e., initial guesses) of the array. This array has (`2+2*NRC+N*NRC`) elements, where `NRC` is the number of RC pairs, and `N` is the number of data segments. The first two elements are `OCV1` and `OCV2`, as OCV is broken down into two unknowns by `OCV = OCV1*S_j + OCV2`. In this way, OCV is to be solved as a linear function of SOC, with OCV1 being the slope and OCV2 being the y-intercept. This practice of using two unknowns to linearly comprise OCV, which was implemented by alumni Ruben Tomlin, can significantly increase the accuracy of OCV parametrisation, compared to using only one unknown. **The `3~2+NRC` elements in the array represent the Rs, as there are `NRC` Rs to be solved. Similarly, there are `NRC` TAUs to be solved, and they are the `3+NRC~2+2*NRC` elements. The remaining `3+2*NRC~2+2*NRC+N*NRC` elements represents the V0s** – what are they? If you go back to the governing equations in Section 0.2, you will see the equations are ordinary differential equations (ODEs) – one ODE requires one given initial value to enable a unique solution, and V0 is the initial value specific to one RC pair and one data segment. In total, there are N*NRC V0s required as initial values to solve N*NRC ODEs.

```
% upperbound - ub
ub = Inf*ones(2+(2+N)*NRC,1);  % initialisation of upperbound; Inf means no
upperbound
ub(3:2+NRC) = 0.3; % for most cells, its Ri <= 0.5
ub(3+NRC) = 1000; % tau_1 <= 100 %10
if NRC ==2
ub(3+NRC) = 50; %tau_1 <= 100 %50
ub(4+NRC) = 1500; % tau_2  <= 1500
end
if NRC ==3
ub(3+NRC) = 10; %tau_1 <= 10 %10
ub(4+NRC) = 100; % tau_2  <= 100
ub(5+NRC) = 1000; % tau_3  <= 1000
end

% if we relimit taui and Ri
if (flags.Ritaui_relimit.tf == 1)
    tau1_limit = str2num(evalin('base','tau1_limit'));
    tau2_limit = str2num(evalin('base','tau2_limit'));
    tau3_limit = str2num(evalin('base','tau3_limit'));
    Ri_limit = str2num(evalin('base','Ri_limit'));

    if isempty(tau1_limit) == 0
        ub(3+NRC) = tau1_limit;
    end
    if isempty(tau2_limit) == 0
        ub(4+NRC) = tau2_limit;
    end
    if isempty(tau3_limit) == 0
        ub(5+NRC) = tau3_limit;
    end
    if isempty(Ri_limit) == 0
        ub(3:2+NRC) = Ri_limit;
    end
end
```

`ub` is the vector defining upper bounds for every unknown and it is initialised as '`Inf`', which means the unknowns are not limited at first. Then `ub(3:2+NRC) = 0.3` sets up the upper bounds for all the Rs (excluding R0), and it is assumed that any R should be no larger than 0.3 Ohm **– this is quite an empirical value that comes from rough observations of various cells parameterised. In other words, the upper bounds for Rs here can be overfitted for some cells.** However, there must be specific upper bounds (as low as possible), because we cannot simply use 'Inf' as the upper bounds, in which case the algorithm will use ~Inf as the solution set and the solving process would be very time-consuming.

Similar to the upper bounds for Rs, **the upper bounds for TAUs are also empirical values**. Dependent on the number of RC pairs, the upper bounds for TAU1, TAU2, and TAU3 vary.

Given that the upper bounds for Rs and TAUs may not be 'one-size-fits-all', the user can customise the bounds using the GUI - user can operate ㉛ - ㉟ to 'relimit' the bounds.

```
% if we are fixing tau from stage 1
 if (flags.taufix.tf == 1)
 lb(3+NRC:2+2*NRC) = flags.taufix.val; % fix the lowerbound and upperbound the 
same at flags.taufix.val
 ub(3+NRC:2+2*NRC) = flags.taufix.val;
 end

% the SOC-OCV table exists
if (flags.SOCOCVtable.tf == 1)
    if S_j >= min(SOCTab) && S_j <= max(SOCTab)
    OCVbound  =  interp1(SOCTab,OCVTab,S_j);
    ub(1) = 0;
    lb(1) = 0;
    ub(2) = OCVbound;
    lb(2) = OCVbound;
    end
end
```

If easyECMfit.m is called by Stage 2 (solving fixed TAU constants), then `flags.taufix.tf == 1` is satisfied. Then `lb` and `ub` will be made the same as `flags.taufix.val`, which is previously worked out by Stage 1. In this way, TAUs will be fixed constants across the while SOC range.

If the user uses ⑳ - ㉑ to import a SOC_OCV table, then `flags.taufix.tf == 1` is satisfied.

In this case the `lb` and `ub` for OCV1 will be 0, and the `lb` and `ub` for OCV2 will be the same as `OCVbound`, which is acquired by linearly interpolating the SOC-OCV table.

```
% (22 Apr 2022) fix the problem of inaccurate OCV param at 100% SOC discharge and 
at 0% SOC charge
if flags.SOCOCVtable.tf == 0
    if ((working_mode == 3)||(working_mode == 4)) && (S_j == 1)  % dishcharge and 
S_j == 1
    ub(1) = 0;
    lb(1) = 0;
    ub(2) = max(cellfun(@max, V));
    lb(2) = max(cellfun(@max, V));
    % (22 Aug 2022) fix the problem of inaccurate R1 R2 param at 100% SOC
discharge
        for hhh = 1: NRC
            ub(3+hhh-1) = prctile (R(hhh,:),75);  % ub(3+hhh-1) = prctile
(R(hhh,end-2:end),75);
            ub(3+NRC+hhh-1) = prctile (tau(hhh,:),75);  % ub(3+NRC+hhh-1) =
prctile (tau(hhh,end-2:end),75);
        end
    end

    if ((working_mode == 1)||(working_mode == 2)) && (S_j == 0)  % charge and S_j
== 0
    ub(1) = 0;
    lb(1) = 0;
    ub(2) = min(cellfun(@min, V));
    lb(2) = min(cellfun(@min, V));
    end
end
```

When it is discharge parametrisation and the parametrisation is at 100% SOC, the `lb` and `ub` for OCV2 will be set the same at `max(cellfun(@max, V))`, which is the maximum battery voltage ever in all the data segments. In other words, at 100% SOC, battery OCV is considered the same as the maximum battery voltage ever appeared in the experiment. This practice is to fix a problem previously observed – the parameterised OCV at 100% SOC was lower than that at 95% SOC, which was not acceptable.

Similarly, when it is charge parametrisation and the parametrisation is at 0% SOC, the `lb` and `ub` for OCV2 will be set the same at `min(cellfun(@min, V))`, which is the minimum battery voltage ever in all the data segments. In other words, at 0% SOC, battery OCV is considered the same as the minimum battery voltage ever appeared in the experiment. This practice is to fix a problem previously observed – the parameterised OCV at 0% SOC was higher than that at 5% SOC, which was not acceptable.

When it is discharge parametrisation and the parametrisation is at 100% SOC, it was also observed in some cases that the parametrised Rs and TAUs can jump ridiculously high, which does not make sense or have any ‘physics’. To fix this problem, the `ub` for Rs is set at `prctile (R(hhh,:),75)`, which is the upper quartile of the parameterised Rs over 0~95% SOC (let us say the SOC window size is 5%), and the `ub` for TAUs is set at `prctile (tau(hhh,:),75)`, which is the upper quartile of the parameterised TAUs over 0~95% SOC. This practice effectively drags the parametrised Rs (at 100% SOC) down to reasonable values.

```
% Aeq * x = beq --> OCV-[V0_1 + V0_2+ V0_3] = V(1)+I(1)*R0   Use N constraints as
initial conditions, important to convergence!!!
```

```
Aeq = zeros(N,2+(2+N)*NRC); % N by (2+(2+N)*NRC) matrix --> N
segments(constraints), (2+(2+N)*NRC) rows in x
beq = zeros(N,1);           % N by 1 vector
for kk = 1:N   % for each segment(cosntraint)
    Aeq(kk,1) = S{kk}(1); % OCV1 * SOC
    Aeq(kk,2) = 1; % OCV2 * 1
    Aeq(kk,3+(1+kk)*NRC:2+(2+kk)*NRC) = -1;  % [V0_1 + V0_2+ V0_3] * -1
    beq(kk) = V{kk}(1) + I{kk}(1)*R0;  % V(1)+I(1)*R0
end
```

The above lines of codes define the linear equality constraints (`Aeq * x = beq`). What are the constraints? If you look back at *Fig. 4* and the governing equations below it, you will know that each constraint applies Kirchhoff's voltage law to the starting point of one data segment (N segments/constraints in total). Why the starting point? This is because the starting point can engage V0s in the constraints.

```
%  A * x <= b -->
%  tau_1 - tau_2 <= 0, tau_2 - tau_3 <= 0, (NRC-1) constraints
%  -Vmax < V0_1 + V0_2+ V0_3 < Vmax,  N*2 constraints
%  Vmin < OCV1*S_j+OCV2 < Vmax,  0 or 2 constraints
if (flags.SOCOCVtable.tf == 1)
        A = zeros(NRC-1+2*N,2+(2+N)*NRC); % NRC-1 by 2+(2+N)*NRC matrix --> NRC-1
constraints, (2+(2+N)*NRC) rows in x
        b = zeros(NRC-1+2*N,1);           % NRC-1 by 1 vector
    else
        A = zeros(NRC+2*N+1,2+(2+N)*NRC); % NRC+1 by 2+(2+N)*NRC matrix --> NRC+1
constraints, (2+(2+N)*NRC) rows in x
        b = zeros(NRC+2*N+1,1);           % NRC+1 by 1 vector
    end

    % constraints for tau
    for kk = 1:NRC-1
        A(kk,2+NRC+kk) = 1;
        A(kk,3+NRC+kk) = -1;      % if kk=1 --> tau_1 - tau_2 <= 0   if kk=2 -->
tau_2 - tau_3 <= 0
    end

    % constraints for V0_i
    for jj = 1:N
        for tao = 1:NRC
        A(NRC-1+jj,2+2*NRC+tao+(jj-1)*NRC) = 1;
        A(NRC-1+N+jj,2+2*NRC+tao+(jj-1)*NRC) = -1;
        end
        if flags.cathode_mode.tf == 0 && flags.anode_mode.tf == 0  % full cell
        b(NRC-1+jj) = str2num(evalin('base','voltage_up_limit')); % V0_1 + V0_2+
V0_3 <= 4.2
        b(NRC-1+N+jj) = str2num(evalin('base','voltage_up_limit')); % -[V0_1 +
V0_2+ V0_3] <= 4.2
        else
            if flags.cathode_mode.tf == 1 % cathode
            b(NRC-1+jj) = str2num(evalin('base','voltage_up_limit')) * 1.2; % a
safety factor of 1.2 is placed for cathode
            b(NRC-1+N+jj) = str2num(evalin('base','voltage_up_limit')) * 1.2; % a
safety factor of 1.2 is placed for cathode
```

```
            else  % anode
            b(NRC-1+jj) = str2num(evalin('base','voltage_up_limit')) * 0.2; % a
safety factor of 0.2 is placed for cathode
            b(NRC-1+N+jj) = str2num(evalin('base','voltage_up_limit')) * 0.2; % a
safety factor of 0.2 is placed for cathode
            end
        end
    end
```

The above lines of codes define the linear inequality constraints (`A * x <= b`). The are 3 kinds of constraints – `(NRC-1)` constraints for TAU, `N*2` constraints for V0s and `0 or 2` constraints for OCV – if the user imported a prescribed SOC-OCV table, OCV would not be parameterised so there is `0` constraint for OCV, otherwise `2` constraints for OCV.

The constraints for TAU define that `tau_1 <= tau_2` and `tau_2 <= tau_3`.

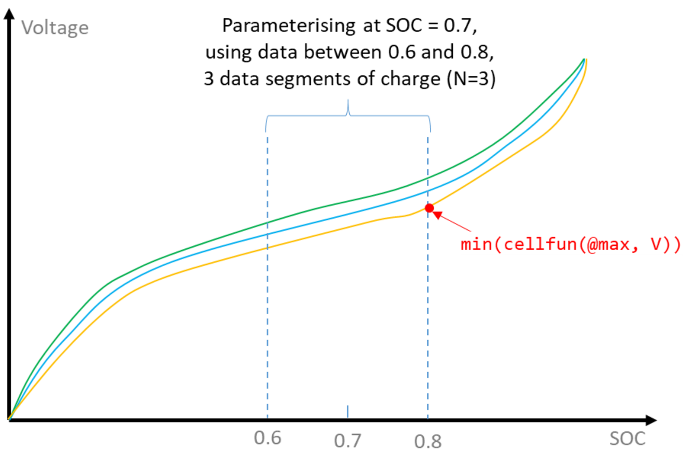
For full-cell parametrisation, the constraints for V0s define that the sum of V0s (`V0_1 + V0_2+ V0_3`) should be no larger than `Vmax` and should be no smaller than -`Vmax`. `Vmax` is the maximum full-cell voltage defined by the user in ⑨. For cathode parametrisation, a multiplication factor 1.2 is applied to `Vmax`. For anode parametrisation, a multiplication factor 0.2 is applied to `Vmax`. **Please be noted that again, these multiplication factors were decided based on rough observations so are empirical, which may cause overfitting.**

```
% constraints for OCV
    if (flags.SOCOCVtable.tf == 0)
    A(NRC+2*N,1) = S_j;
    A(NRC+2*N,2) = 1;
        if flags.cathode_mode.tf == 0 && flags.anode_mode.tf == 0 % full cell
        b(NRC+2*N) = str2num(evalin('base','voltage_up_limit'));  % OCV1*S_j +
OCV2 <= 4.2
        else
            if flags.cathode_mode.tf == 1 % cathode
            b(NRC+2*N) = str2num(evalin('base','voltage_up_limit')) * 1.2; % a
safety factor of 1.2 is placed for cathode
            else % anode
            b(NRC+2*N) = 0; % Voc < 0
            end
        end
        if working_mode == 2  % full cell/cathode charge
            b(NRC+2*N) = min(cellfun(@max, V));    % OCV1*S_j + OCV2 <=
min(max(V))


        end
          -------------------------------------------------------

    A(NRC+2*N+1,1) = -S_j;

    A(NRC+2*N+1,2) = -1;
        if working_mode == 4  % full cell/cathode discharge
```

```
            b(NRC+2*N+1) = - max(cellfun(@min, V));  % -OCV1*S_j - OCV2 <= -
max(min(V))         (OCV1*S_j + OCV2 >= max(min(V))
        end
        if working_mode == 2  % full cell/cathode chagre
            b(NRC+2*N+1) = - str2num(evalin('base','voltage_low_limit'));  % -
OCV1*S_j - OCV2 <= - 2.7  (OCV1*S_j + OCV2 >= 2.7)
        end
        if working_mode == 1  % anode - charge
            b(NRC+2*N+1) = str2num(evalin('base','voltage_up_limit')) * 0.2 ; % -
OCV1*S_j - OCV2 <= 0.2* Vmax  (OCV1*S_j + OCV2 >= -0.2*Vmax)
        end
        if working_mode == 3  % anode - discharge
            b(NRC+2*N+1) = - max(cellfun(@min, V)); % -OCV1*S_j - OCV2 <= -
max(min(V))               (OCV1*S_j + OCV2 >= max(min(V))

        end
    end
```

The codes above define constraints for OCV (`OCV1*S_j + OCV2`), and can be divided into two parts, as separated by the dotted line.

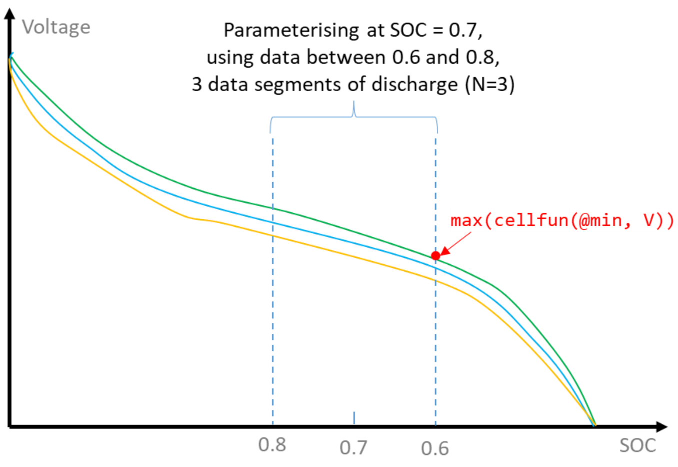
The first part constrains the upper limits for OCV, and can be further divided into three cases – full cell, cathode and anode. For full-cell parametrisation, the upper limit is `Vmax`. For cathode parametrisation, the upper limit is `1.2 * Vmax` - again, the multiplication factor 1.2 was decided based on rough observations so is empirical, which may cause overfitting. For anode parameterisation, the upper limit is `0`, as anode voltage is considered below zero. Apart from the above three cases, **there is one special case -** **`full cell/cathode charge`**. In this case, `cellfun(@max, V)` gets the maximum voltage of EACH data segment, then `min(cellfun(@max, V))` gets the minimum of maximum voltage of ALL data segments and is used as the upper limit. **Still sounds complicated? *Fig. 13* gives graphical explanations on it**. Then it comes up with another question – why bother setting up this constraint for the special case? **This is because in `full cell/cathode charge` parameterisation, the constraint can effectively & essentially lead to a monotonous OCV curve over the full SOC range. In other words, the constraint is used to guarantee a monotonous OCV, which was implemented in a very recent version of BatPar. Quite straightforward - if the developer was forced to name the most ingenious piece of codes over BatPar, then this constraint would be it.**


Voltage
Parameterising at SOC = 0.7,
using data between 0.6 and 0.8,
3 data segments of charge (N=3)
min(cellfun(@max, V))
0.6
0.7
0.8
SOC

$$OCV_{SOC=0.7} < OCV_{SOC=0.8} < \min(\text{cellfun}(@\max, \text{V}))$$

Fig. 13. Explanation of the constraint (`OCV <= min(cellfun(@max, V))`) in full cell/cathode charge parameterisation.

The second part constrains the lower limits for OCV, and can be further divided into four cases – full cell/cathode discharge, full cell/cathode charge, anode charge, and anode discharge. For full cell/cathode discharge parameterisation, `cellfun(@min, V)` gets the minimum voltage of EACH data segment, then `max(cellfun(@min, V))` gets the maximum of minimum voltage of ALL data segments and is used as the lower limit. **Still sounds complicated? *Fig. 14* gives graphical explanations on it**. **This constraint can effectively & essentially lead to a monotonous OCV curve over the full SOC range. In other words, the constraint is used to guarantee a monotonous OCV, which was implemented in a very recent version of BatPar.**


Voltage
Parameterising at SOC = 0.7,
using data between 0.6 and 0.8,
3 data segments of discharge (N=3)
max(cellfun(@min, V))
0.8
0.7
0.6
SOC

$$OCV_{SOC=0.7} > OCV_{SOC=0.6} > \max(\text{cellfun}(@\min, \text{V}))$$

Fig. 14. Explanation of the constraint (`OCV >= max(cellfun(@min, V))`) in full cell/cathode discharge parameterisation.

For full cell/cathode charge parameterisation, the lower limit is `Vmin`, which is the minimum full-cell voltage defined by the user in ⑩. For anode charge parametrisation, the lower limit is `(-0.2) * Vmax` - again, the multiplication factor -0.2 was decided based on rough observations so is empirical, which may cause overfitting. Also, be noted that **anode OCV is considered below zero**. For anode discharge parametrisation, the lower limit is `max(cellfun(@min, V))` – explanations for this will not be provided – it is homework for the reader. (To be fair, if you can understand *Fig. 13* and *Fig. 14*, then you should be able to understand this as well)

```
%% Sovle formulated optmisation problem by fmincon

% options =
optimoptions(@fmincon,'Display','off','MaxFunctionEvaluations',NRC*1500,'MaxIterat
ions',NRC*500,'OptimalityTolerance',1e-20,'StepTolerance', 1e-20,
'FunctionTolerance', 1e-20);
options =
optimoptions(@fmincon,'Display','off','MaxFunctionEvaluations',NRC*1500,'MaxIterat
ions',NRC*500,'OptimalityTolerance',1e-10,'StepTolerance', 1e-10,
'FunctionTolerance', 1e-10);

% default values: MaxFunctionEvaluations:3000; MaxIterations:1000;
OptimalityTolerance:1e-6; StepTolerance:1e-10; FunctionTolerance:1e-6
% In general, you can improve accuracy of params by increasing -ations or
decreasing tolerances, but it is not always worth doing - tiny improvements VS
significant computational loads, and sometimes overfitting!!!

x_solved = fmincon(@(param)
F_easyECMfit(t,I,V,T,S,NRC,R0,flags,param),param0,A,b,Aeq,beq,lb,ub,[],options); %
x_solved - column vector

% split up params to output
OCV1 = x_solved(1); OCV2 = x_solved(2); OCV = OCV1*S_j + OCV2; % optimised value
of the OCV
R = x_solved(3:2+NRC); tau = x_solved(3+NRC:2+2*NRC);          % optimised value
of Ri and tau_i
V0_i = x_solved(3+2*NRC:end); % optimised value of V0_i
initdata = transpose(reshape(V0_i,[NRC N])); % reshape V0_i
```

This section of codes calls the algorithm fmincon to solve the parameters. ‘`options = Optimoptions…`’ is the command to configure the algorithm. ‘`x_solved = fmincon…`’ is the command to execute the algorithm. `F_easyECMfit` is the cost function used in the algorithm.

```
%% difference of voltage between experiment and parameterised model

DIFF_ = [];  % Intialise a vector connector
for kk = 1:N
        Diff_Exp_Mod =
localsolve(t{kk},I{kk},T{kk},S{kk},OCV1,OCV2,R0,R,tau,initdata(kk,:),flags)-
V{kk}; % difference between experimental and model voltage
        Diff_Exp_Mod = Diff_Exp_Mod(~isnan(Diff_Exp_Mod)); % filter out NaN
        DIFF_ = cat(1,DIFF_,Diff_Exp_Mod); % Connect two column vectors; '1' menas
along column direction
end
```

The codes above calculate the difference of voltage between experiment and parameterised model. `DIFF_` is the vector saving the difference of `N` data segments, and it will be further used to calculate local RMSE (the RMSE you can see in the command window as parameterisation proceeds to the next SOC window). `localsolve` is the function calculating the voltage of the parameterised model, and `V{kk}` is the experiment voltage.

```
%% Plots of voltage of experiment and parameterised model

if (flags.allplots.tf == 1)
    figure;
    xlabel('Relative time (s)');
    ylabel('Voltage (V)');
    title (['Experiment VS Paramterised Model - Voltage at around SOC = ',
num2str(S_j),]);
    hold on;
    for kk = 1:N
        plot(t{kk}-t{kk}(1),V{kk}); legend (['Experiment data ',num2str(kk),]);
        plot(t{kk}-
t{kk}(1),localsolve(t{kk},I{kk},T{kk},S{kk},OCV1,OCV2,R0,R,tau,initdata(kk,:),flag
s)); legend (['Model data ',num2str(kk),]);
    end

    for iii = 1:N
    leg_string{2*iii-1} = ['Experiment data ',num2str(iii)];
    leg_string{2*iii} = ['Model data ',num2str(iii)];
    end
    legend(leg_string); % legends for experiment and model data

    drawnow;
end
```

This section of codes generates plots comparing the experiment voltage and parameterised model voltage, specific to the current SOC window. Currently this section is disabled because if not, there would be too many plots popping up during parameterisation, which imposes a great burden to the memory and GPU. Besides, at end of parameterisation, there will be another plot showing similar but more summative information (`Global comparison - Experiment VS Paramterised Model.fig`). If in any case, you want to enable the plots, please go to loadexperiments_.m and set `flags.allplots.tf = 'yes'` – the developer takes no responsibility if your video card burns.

7.3 Places of future improvement

Apparently, easyECMfit.m is the file with the most places of improvement.

1> About R0 calculation. '*any `dI` that is smaller than 0.15C is considered as noise and as such is NOT used to calculate R0*' (*← search this sentence to locate context*). Due to this, if your experiment was conducted with very small current that is smaller than 0.15C, BatPar would not give any parameterisation results but errors of 'insufficient data'. **So if the current in your experiment is unavoidably very small, do not forget to go to the codes and replace 0.15C with a proper value.**

2> About from variable TAU to fixed TAU. To get fixed TAU, BatPar must solve the variable TAU beforehand, as the developer found no direct solution to work out fixed TAU. However, it is worth investigating the approach to get fixed TAU directly without working out variable TAU, as **this could reduce the solving time of BatPar by half**.

3> About initial guesses of parameters (`param0`). It is pretty sure that the initial guesses have significant impacts on final parameterisation results, which was explained in the slides. **For a highly nonlinear system such as the battery under AMPP test protocol, different sets of initial guesses can lead to very different sets of parameters, even though any set of parameters can well account for the battery circuit**. There are two directions for future improvement – one is to set up 'smart' initial guesses rather than using arbitrary values. In this case, the challenge is how to define and figure out 'smart' guesses – perhaps via a deep pre-processing on the test data? The other direction is to use multiple sets of initial guesses to run the parameterisation and choose best of all – the price to pay is more solving time.

4> About upper bounds (`ub`) for Rs and Cs. Currently the upper bounds are empirical values which may cause overfitting. Even though the 'relimit' functionality is provided for the user, it would be better if 'smart' upper bounds can be found out without user's efforts on it – again, the challenge is how to define and figure out 'smart' upper bounds.

5> About the multiplication factors (0.2 or 1.2) in constraints for V0s and OCVs of anode/cathode parameterisation. Currently the multiplication factors are empirical values which may cause overfitting. However, it should be relatively easy to improve the multiplication factors – in the public place, there should be referenced voltage range for the cathode/anode, if we know the materials for the cathode/anode?

6> About more constraints to be formulated in the algorithm. As you can see, there have been many equality and inequality constraints formulated in the codes, but more constraints can further improve the accuracy of parameters and reduce solving time. It is worth thinking about extra constraints and implement them in the codes.

7> (Proposed by Tom Holland) About options of fmincon algorithm (`options`). Currently, the `'OptimalityTolerance', 'StepTolerance', 'StepTolerance'` of fmincon each is set as `1e-10` – the tolerances may be a bit too small? It is worth trying larger tolerances and see the outcomes.

8. *localsolve.m* & *fullsolve.m*

8.1 Functionality and structure

localsolve.m, called by easyECMfit.m and F_easyECMfit.m, is a function of the parameterised battery model. Namely, this function can be regarded as a battery model using the parameters solved.

fullsolve.m, called by ECM_easy.m, AmpDenpOnly.m, TempDenpOnly.m, and AmpAndTempDenp.m, is also a function of the parameterised battery model. Different from localsolve.m that only works for a specific SOC window, fullsolve.m represents a global battery model that accounts for the whole SOC range appeared in the experiment.

8.2 Core codes and hard-to-read codes

---

localsolve.m

```
function Vhat = localsolve(t,I,T,S,OCV1,OCV2,R0,R,tau,initdata,flags)
```

This function accepts two kinds of inputs – (1) time, current, temperature, SOC (`t,I,T,S`) from the experiment, (2) parameterised OCV, Rs, TAUs, V0s (`OCV1,OCV2,R0,R,tau,initdata`) for the present SOC window. This function returns battery voltage (`Vhat`) solved by the parameterised model.

---

localsolve.m

```
maxdt = 0.05;
tnew = sort(vertcat((t(1):maxdt:t(end))',t(end))); % subdivide timeline; vertcat -
splice two vectors; sort - ascending order
tnew = unique(tnew);                                % unique tnew in case t(end)
can be divisible by maxdt, in which case there would be two 't(end)' in tnew

I      = interp1(t,I,tnew);
T      = interp1(t,T,tnew);        % interpolate new data on tnew
S      = interp1(t,S,tnew);

if (flags.R0temp.tf == 1)  %
    R0 = R0(T);            %
end                        %
…
```

The codes above maps the experimental data (`I, T, S`) from old timeline (`t`) to a new, fixed-step timeline (`tnew`). The reason for doing this is that experiments sometimes use variable time steps, in which case the data is not evenly logged with respect to time. However, **for integration purpose (integration is necessary to solve ODE), the time steps should**

better be evenly divided and also be small enough. `maxdt` is the new time interval, which is set as 0.05 second.

`flags.R0temp.tf` came from an earlier version and was used for T-dependent R0 parameterisation. However, in the latest version T-dependent parameterisation was realised in another way so this variable has **no effect** on anything of parametrisation. Definitely, this can be removed from the codes.

----

localsolve.m

```
len = length(tnew);
dt = diff(tnew);
Vi = zeros(len,length(initdata));    % Voltage of each RC pair; initdata - N by
NRC vector, length(initdata) = NRC
                                     % Vi - len by NRC matrix
Vi(1,:) = initdata + 1E-10*sort(rand(size(initdata)),'descend'); % random noise to
split tau_1 and tau_2, R1 and R2. Otherwise, the algorithm would probably give
tau_1=tau_2 and R1=R2
% be noted, initdata is auto-transposed by this assignment above

% Integrate up; See Ruben's slides --> 3RC ECM
A = transpose(1./tau);   % tau - NRC by 1 vector
B = transpose(R./tau);   % column vector transposed to row vector

% Calculate Vi on tnew
for k = 2:len
    Vi(k,:) = ( 1 - dt(k-1)*A ) .* Vi(k-1,:) ...
               + (dt(k-1)/2)*B*(I(k-1) + I(k)); % (I(k-1) + I(k))/2 - midpoint
end
Vhat = OCV1*S+OCV2- R0.*I - sum(Vi,2);      % OCV & R0 - single number;
                                            % S & I - column vector (the vector
holding data at around S_j);
                                            % sum(Vi,2) - sum up the voltage of
each RC pair; '2' means sum over the 2 dimsension
                                            % Vhat - column vector
Vhat = interp1(tnew,Vhat,t);   % interpolate back onto t
end
```

The above codes calculate `Vhat` based on the governing equations in Section 0.2. Please go through the comments alongside the codes to understand the codes. Just one line needing extra explanation - `Vi(1,:) = initdata + 1E-10*sort(rand(size(initdata)),'descend')`. This was implemented by Ruben, and **the key point is to make sure that the fmincon algorithm will not treat all the RC pairs as identical and consequently solves TAU1=TAU2 and R1=R2**, as localsolve.m is called by F_easyECMfit.m and F_easyECMfit.m is further called by fmincon (i.e., localsolve.m is indirectly called by fmincon).

----

fullsolve.m

```
function [Vhat, t_useful] = fullsolve(t,I,T,S,S_j,OCV,R0,R,tau,flags,NRC)
```

Compared to localsolve.m, fullsolve.m NO longer accepts V0s as input, but adds two more inputs as SOC division (`S_j`) and number of RC pairs (`NRC`). It returns one more input as the useful timeline of experiment (`t_useful,` the timeline with valid data for parameterisation).

----

fullsolve.m

```
OCV = OCV (~isnan(OCV));  % available tables
R0 = R0 (~isnan(R0));
R = reshape(R(~isnan(R)),NRC,[]);
tau = reshape(tau(~isnan(tau)),NRC,[]);


S_j = S_j(~isnan(OCV));
index = ((S >= min(S_j)) & (S <= max(S_j)));  % index of avaible SOC

t = t(index);   %  available timeline
I = I(index);
T = T(index);
S = S(index);

if length(t)<2 % no available timeline
    Vhat = nan;
    t_useful = nan;
else
...
...
```

As aforementioned, fullsolve.m generally accepts two kinds of inputs – experimental data (`t,I,T,S`) and parameters (`OCV,R0,R,tau`), using which to feed the battery model so as to return battery voltage across full SOC range. However, **these inputs need to be cleaned up before feeding the battery model, because the solved parameters can be 'NaN', if T dependence and/or C-rate dependence is configured**. The above codes firstly clean up `OCV,R0,R,tau` by filtering out NaN, and then use the `OCV` as the index to clean up SOC division (`S_j`) and finally clean up the experimental data (`t,I,T,S`). The cleaned experimental data should at least contain two data points, otherwise (`if length(t)<2`) the battery model cannot work.

----

fullsolve.m

```
if (flags.R0temp.tf == 1)                       %
    R0 = interpn(flags.R0temp.val,S_j,R0,T,S);   %
else
    R0 = interp1(S_j,R0,S);
end
OCV = interp1(S_j,OCV,S);
R = nlininterpvec(S_j,R,S);
```

```
tau = nlininterpvec(S_j,tau,S);
```

`flags.R0temp.tf` came from an earlier version and was used for T-dependent R0 parameterisation. However, in the latest version T-dependent parameterisation was realised in another way so this variable has **no effect** on anything of parametrisation, as `(flags.R0temp.tf == 1)` is never satisfied. Definitely, this can be removed from the codes.

The key point of the above codes is to interpolate parameters from fixed SOC division to full SOC range. As you can see, interpolations of R0 and OCV use the MATLAB in-built command (`interp1`), since OCV and R0 are vectors. In contrast, Rs and TAUs are matrices so `interp1` cannot be directly applied. In this case, Ruben developed the function `nlininterpvec` to interpolate Rs and TAUs.

----

9. *nlininterpvec.m & F_easyECMfit.m & OCVparameterisationONLY.m & RandTauRelimit.m*

9.1 Functionality and structure

nlininterpvec.m, called by fullsolve.m, is a function to interpolate Rs and TAUs from fixed SOC division to full SOC range. As this function has been introduced in Section 8.2 and has only three lines of codes, it will not be elaborated in the following.

F_easyECMfit.m, called by eastECMfit.m, is the cost function used in fmincon algorithm. It formulates the RMSE between experiment voltage and parametrised model voltage.

OCVparameterisationONLY.m, called by ECM_Easy.m, is a script to preform OCV parameterisation while skipping the other parameters.

RandTauRelimit.m, called by ECM_easy.m, AmpDenpOnly.m, TempDenpOnly.m, and AmpAndTempDenp.m, is a script to advise the user whether the parameterised Rs and TAUs are overfitted, so the user can take further action to re-parameterise by increasing the upper limits for Rs and/or TAUs.

9.2 Core codes and hard-to-read codes

F_easyECMfit.m

```
function F = F_easyECMfit(t,I,V,T,S,NRC,R0,flags,param)
    N = numel(t); % number of data segment
    OCV1 = param(1);OCV2 = param(2);
    R = param(3:2+NRC); tau = param(3+NRC:2+2*NRC);
    initdata = transpose(reshape(param(3+2*NRC:end),[NRC N])); %
param(3+2*NRC:end), NRC*N by 1 vector --> reshape to NRC by N matrix --> transpose
to N by NRC matrix
    F = 0;
    L2 = @(time,z) sqrt(trapz(time,z.^2));  % anonymous function; trapz -
cumulative trapezoidal numerical integration
    for kk = 1:N
        F = F + L2(t{kk},V{kk} -
localsolve(t{kk},I{kk},T{kk},S{kk},OCV1,OCV2,R0,R,tau,initdata(kk,:),flags)); %
trapz [(V_experiment - V_model)^2] dt
    end
end
```

F_easyECMfit.m accepts experimental data (`t,I,V,T,S`) and battery parameters (`R0, param`) as inputs, and returns the RMSE between experimental voltage (`V`) and parametrised model voltage (solved by `localsolve`). **Be noted that `R0` is already solved outside of the function, so it is a known value. In contrast, OCV (composed of `OCV1` and `OCV2`), Rs, TAUs and V0s (`initdata`) are unknown** – they were formulated in the variable `initdata` and passed into the function. This function is very well commented by the developer so please read the comments to understand `F` and `L2`.

OCVparameterisationONLY.m

In many ways, 'OCVparameterisationONLY.m' is a scale-reduced 'ECM_easy.m', since OCVparameterisationONLY uses the same code structure as ECM_easy but parameterises OCV only. Dependent on the user's choice in ㉔ , OCVparameterisationONLY has two working modes – true OCV or pseudo OCV, which are further explained in the following.

OCVparameterisationONLY.m

```
% true OCV extraction
    j = 1;
    for i = 1:length(I_)-1
         if (I_(i) == 0) && (I_(i+1) ~= 0)
            SOC_OCV (j,1) = S_ (i);
            SOC_OCV (j,2) = V_ (i);
            j = j + 1;
         end
    end
    if I_(end) == 0
        SOC_OCV (j,1) = S_ (end);
        SOC_OCV (j,2) = V_ (end);
    end
    SOC_OCV = sortrows(SOC_OCV,1);
```

The method to extract true OCV is very simple – just identify the end of relaxation by `if (I_(i) == 0) && (I_(i+1) ~= 0)`, and then record the corresponded battery voltage (`V_(i)`) as the OCV at the present SOC (`S_(i)`). There is one special case – if battery current at the end of experimental data (`I_(end)`) is zero, then the end-of-experiment battery voltage will be recorded as one more point of OCV. `sortrows` is used to sort out the extracted OCV in an ascending order in case there are multiple true OCV tests in one experimental datasheet.

OCVparameterisationONLY.m

```
% ascending sortation
    SOC_old = cell2mat(S_);
    OCV_old = cell2mat(V_);
    [~,ind_,~] = unique(SOC_old); % (~:neglected  ind: index of unique rows)
    SOC_OCV_old(:,1) = SOC_old(ind_);
    SOC_OCV_old(:,2) = OCV_old(ind_);
    SOC_OCV_old = sortrows(SOC_OCV_old,1);
    ...
    ...
% OCV interpolation
```

```
    SOC_new = min(SOC_OCV_old(:,1)) : 1e-4 : max(SOC_OCV_old(:,1));  % new SOC
trace
    SOC_new = SOC_new';
    OCV_new = interp1(SOC_OCV_old(:,1), SOC_OCV_old(:,2), SOC_new);  % generate
new OCV trace;
```

The method to extract pseudo OCV is very simple – just use the raw battery voltage in the experiment as the OCV. In terms of code implementation, `unique(SOC_old)` performs data cleaning by removing repetitive SOC points in case there are multiple pseudo OCV tests in one experimental datasheet.

The remaining codes perform linear interpolation to generate a more divided SOC-OCV table with the SOC interval being `1e-4` – Perhaps this is a bit too divided. Also, the interpolation is not necessarily needed, if your pseudo OCV test is well logged.

RandTauRelimit.m

```
if NRC == 1
   if isempty(tau1_limit) == 1
        if tau_1_reco >= 1000 * 0.95
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
            disp('     Please increase the upper limit for tau1 and rerun the
codes')
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
        end
   else
        if tau_1_reco >= tau1_limit * 0.95
           disp('     ======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
            disp('     Please increase the upper limit for tau1 and rerun the
codes')
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
        end
   end

   RR1 = prctile (R(1,:),50);    % median of R1

   if isempty(Ri_limit) == 1
        if RR1 >= 0.3 * 0.95
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
            disp('     Please increase the upper limit for Ri and rerun the
codes')
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
        end
   else
        if RR1 >= Ri_limit * 0.95
           disp('     ======CAUTION======CAUTION======CAUTION======CAUTION======CAUTION======')
            disp('     Please increase the upper limit for Ri and rerun the
codes')
```

```
            disp('
======CAUTION======CAUTION======CAUTION======CAUTION======')
        end
   end
end
...
...
```

As aforementioned, there is a possibility that the Rs and TAUs are overfitted because the upper bonds for Rs and TAUs are too low. Even though the user is given the change to customise the upper bonds through ㉛ - ㉟ , the upper bonds may set too low in the first place.

RandTauRelimit.m is the script to inform the user whether it is the case. Based on the number of RC pairs (1, 2, or 3), the RandTauRelimit can be divided into three parts, and the codes above are for the case of 1 RC pair (`if NRC == 1`). `if isempty(tau1_limit) == 1` means that the user does not customise an upper limit for TAU1, so the default value 1,000 will be used as the upper limit, otherwise, `tau1_limit` will be used. **The criterion to judge whether the upper limit is low is to compare the median of parameterised TAU1 over full SOC range (`tau_1_reco`) to the upper limit * 95%.** If the former is no smaller than the later, then it is assumed that the upper limit is too low, and message will be displayed in MATLAB command window to inform the user. Similar approach applies to R1 **– if the median of parameterised R1 over full SOC range (`RR1`) is not smaller than the upper limit of R1 * 95%, then it is reckoned that the upper limit of R1 is too low.**

----

10. *TempDenpOnly.m & data_temperatureDependent.m & PostProcess4Temp.m*

10.1 Functionality and structure

TempDenpOnly.m, diverted from ECM_easy.m, is a script to perform both SOC- and T-dependent parameterisation. This script, in many ways, is very similar to ECM_easy.m. The difference is that ECM_esay.m can only solve SOC-dependent parameterisation, while TempDenpOnly.m further considers T-dependence.

data_temperatureDependent.m, called by TempDenpOnly.m, is a function to reform data segments from 1-D SOC-dependent to 2-D SOC- and T-dependent. In other words, this function applies temperature windows to the SOC-dependent data segments to produce new data segments that lie in different SOC and temperature windows.

PostProcess4Temp.m, called by TempDenpOnly.m, is a script for post-processing the SOC- and T-dependent parameters. It generates the look-up tables and plots of parameters at different SOCs and Ts. There is nothing much to explain about the post-processing codes, as the codes are all about variable restructuring and visualisation - the best way to understand the codes is to look at the generated look-up tables and plots and try to link them back with the codes.

10.2 Core codes and hard-to-read codes

----

data_temperatureDependent.m

```
function [dataTTT, SOC_steps]= data_temperatureDependent (data, Temps_depended,
Temps_sensitivity, S_j, NRC)
```

`data_temperatureDependent` is the first line and the first function being called in TempDenpOnly.m. **This function accepts SOC-dependent data segments (`data`) in the format of 1 by N cell, temperatures (`Temps_depended`) at which parameters will be solved respectively, as defined by the user in ㉙, sensitivity of temperature (`Temps_sensitivity`) which delimits error range of temperatures, as defined by the user in ㉚,** SOC division (`S_j`), and number of RC pairs (`NRC`).

This function returns the both SOC- and T-dependent data segments (`dataTTT`) in the format of a 2D cell, with T being the index of rows and SOC being the index of columns, number of non-empty SOC divisions at each temperature (`SOC_steps`), i.e., number of useful columns in each row of `dataTTT`.

---

data_temperatureDependent.m

```
% data reorganisation
for i = 1:length(Temps_depended)     % every Temp
    for j = 1: length(data.T)        % every SOC
        count = 0;
        for k = 1: length(data.T{1,j}) % every segment

            indexTTT =  ((Temps_depended(i) - Temps_sensitivity) <=
data.T{1,j}{k}) + (data.T{1,j}{k} <= (Temps_depended(i) + Temps_sensitivity)) ==
2;

            if any(indexTTT)
                    count = count + 1;
                    dataTTT.I {i,j}{count,1} = data.I{1,j}{k}(indexTTT);
                    dataTTT.t {i,j}{count,1} = data.t{1,j}{k}(indexTTT);
                    dataTTT.T {i,j}{count,1} = data.T{1,j}{k}(indexTTT);
                    dataTTT.V {i,j}{count,1} = data.V{1,j}{k}(indexTTT);
                    dataTTT.S {i,j}{count,1} = data.S{1,j}{k}(indexTTT);
            end
        end
    end
end
```

The codes above add T-dependence to the SOC-dependent data segments. At a certain temperature (`Temps_depended(i)`), every data segment with every SOC (`data.T{1,j}{k}`) will be filtered by '`((Temps_depended(i) - Temps_sensitivity) <= data.T{1,j}{k}) + (data.T{1,j}{k} <= (Temps_depended(i) + Temps_sensitivity)) == 2`', which gives a temperature index `indexTTT`. If `indexTTT` is not empty (`if any(indexTTT)`), it means that the present data segment contains data specific/useful to the target temperature (`Temps_depended(i)`). In this case, the filtered data (`data.I{1,j}{k}(indexTTT)`) will be saved in a new cell (`dataTTT.I {i,j}{count,1}`).

---

data_temperatureDependent.m

```
% data check --> filter out empty data cells
indexDataT = ~cellfun(@isempty,dataTTT.t);

for zhu = 1:length(Temps_depended)

    dataTTT_t = dataTTT.t(zhu,:);
    dataTTT_temp.t {zhu,1} = {dataTTT_t(indexDataT(zhu,:))};
    dataTTT_I = dataTTT.I(zhu,:);
    dataTTT_temp.I {zhu,1} = {dataTTT_I(indexDataT(zhu,:))};
    dataTTT_T = dataTTT.T(zhu,:);
    dataTTT_temp.T {zhu,1} = {dataTTT_T(indexDataT(zhu,:))};
    dataTTT_V = dataTTT.V(zhu,:);
    dataTTT_temp.V {zhu,1} = {dataTTT_V(indexDataT(zhu,:))};
    dataTTT_S = dataTTT.S(zhu,:);
    dataTTT_temp.S {zhu,1} = {dataTTT_S(indexDataT(zhu,:))};
    dataTTT_temp.S_j_I {zhu,1} = {S_j(indexDataT(zhu,:))};
end
```

There can be cases that the experiment covers all temperatures and SOCs, but does not cover the combination of a specific temperature and SOC. In this case, the correspond element in `dataTTT` will be empty. The codes above are to remove the empty elements in `dataTTT` so that the empty elements will not be returned to the function, because empty elements can cause errors in TempDenpOnly.m.

----

data_temperatureDependent.m

```
% clean down
for tao = 1:length(Temps_depended)
    for i = 1: length(dataTTT_temp.t{tao,1}{1,1})
        ind = cellfun('length', dataTTT_temp.t{tao,1}{1,1}{1,i}) >= (NRC * 3 + 2)
* 2;
        dataTTT_temp_temp.t{tao,1}{1,1}{1,i} =
dataTTT_temp.t{tao,1}{1,1}{1,i}(ind);
        dataTTT_temp_temp.I{tao,1}{1,1}{1,i} =
dataTTT_temp.I{tao,1}{1,1}{1,i}(ind);
        dataTTT_temp_temp.T{tao,1}{1,1}{1,i} =
dataTTT_temp.T{tao,1}{1,1}{1,i}(ind);
        dataTTT_temp_temp.V{tao,1}{1,1}{1,i} =
dataTTT_temp.V{tao,1}{1,1}{1,i}(ind);
        dataTTT_temp_temp.S{tao,1}{1,1}{1,i} =
dataTTT_temp.S{tao,1}{1,1}{1,i}(ind);
    end
end
dataTTT_temp_temp.S_j_I = dataTTT_temp.S_j_I;
dataTTT = dataTTT_temp_temp;
```

The codes above are very similar to cleandown.m, with the purpose to recognise the length of each data segment in `dataTTT_temp` and accordingly discard data segments that are too short to be used to solve parameters. The difference between cleandown.m and the above codes is that cleandown.m is used to clean SOC-dependent data segments, while the above codes are to clean both SOC- and T-dependent data segments.

TempDenpOnly.m

As aforementioned, TempDenpOnly.m is very similar to ECM_easy.m, so the following will focus on the two key differences of TempDenpOnly.m from ECM_easy.m.

```
for iiii = 1:length(Temps_depended)

disp(['-> Parameterisation taking place at TEMPERATURE ',num2str(iiii),' of
',num2str(length(Temps_depended)),' -- ', num2str(Temps_depended(iiii)),'degC.'])
...
```

...

Let us say you specified 4 Ts and 20 SOCs. The codes will firstly access data segments for the first T and perform parameterisation at all the 20 SOCs. After that, the codes will proceed to the second T and perform parameterisation at all the 20 SOCs…This is how BatPar performs SOC- and T-dependent paramenterisation.

TempDenpOnly.m

```
if working_mode == 3 || working_mode == 4
    f111=figure;
    hold on
    for kk = 1:numel(T) % discharge segment
        indexTTT = ((Temps_depended(iiii) - Temps_sensitivity) <= T{kk}) + (T{kk}
<= (Temps_depended(iiii) + Temps_sensitivity)) == 2;
        plot(S{kk}(indexTTT),V{kk}(indexTTT))
        xlabel('SOC');
        ylabel('Voltage (V)');
        title (['Data extraction (discharge) - Voltage VS SOC @
',num2str(Temps_depended(iiii)),'degC']);
    end
    ...
    ...
else
    f222=figure;
    hold on
    for kk = 1:numel(T_charge) % discharge segment
        indexTTT = ((Temps_depended(iiii) - Temps_sensitivity) <= T_charge{kk}) +
(T_charge{kk} <= (Temps_depended(iiii) + Temps_sensitivity)) == 2;
        plot(S_charge{kk}(indexTTT),V_charge{kk}(indexTTT))
        xlabel('SOC');
        ylabel('Voltage (V)');
        title (['Data extraction (charge) - Voltage VS SOC @ ',
num2str(Temps_depended(iiii)),'degC']);
    end
    ...
    ...
```

In both SOC- and T-dependent parameterisation, there is one new plot about T-dependent data extraction. If it is discharge parameterisation, figure `f111` will appear, otherwise figure `f222` will.

----

11. *AmpDenpOnly.m* & *data_currentDependent.m* & *currentIndexing.m* & *PostProcess4Crate.m*

11.1 Functionality and structure

AmpDenpOnly.m, diverted from ECM_easy.m, is a script to perform both SOC- and I-dependent parameterisation. This script, in many ways, is very similar to ECM_easy.m. The difference is that ECM_esay.m can only solve SOC-dependent parameterisation, while AmpDenpOnly.m further considers I-dependence. AmpDenpOnly.m is almost the same as TempDenpOnly.m, except that the temperature-related variables are replaced with current-related ones, so this script will not be further explained in the following (To be fair, if you can understand TempDenpOnly.m, then you should be able to understand AmpDenpOnly.m as well).

data_ currentDependent.m, called by AmpDenpOnly.m, is a function to reform data segments from 1-D SOC-dependent to 2-D SOC- and I-dependent. In other words, this function applies current windows to the SOC-dependent data segments to produce new data segments that lie in different SOC and current windows.

currentIndexing.m, called by data_currentDependent.m, AmpDenpOnly.m, data_IandTdependent.m, and AmpAndTempDenp.m, is a function to 'crop' ONE certain piece of data segment according to the current window required. Please be noted that current cropping is much more complicated than temperature cropping, because one complete current pulse is considered as the whole of pulse head, pulse plateau, pulse end and relaxation period, and an ideal current cropping is to include the complete current pulse that than only the pulse plateau.

PostProcess4Crate.m, called by AmpDenpOnly.m, is a script for post-processing the SOC- and I-dependent parameters. It generates the look-up tables and plots of parameters at different SOCs and Is. There is nothing much to explain about the post-processing codes, as the codes are all about variable restructuring and visualisation - the best way to understand the codes is to look at the generated look-up tables and plots and try to link them back with the codes.

11.2 Core codes and hard-to-read codes

---

currentIndexing.m

```
function indexIII = currentIndexing (IData, Current, Sensitivity)
```

This function accepts ONE data segment of battery current (`IData`), ONE target current (`Current`), and sensitivity of current (`Sensitivity`). It returns a current index (`indexIII`) which indicates the locations of data points (current pulses) specific/useful to parameterisation at the target current.

----

currentIndexing.m

```
indexIII =  ((Current - Sensitivity) <= IData) + (IData <= (Current + 
Sensitivity)) == 2; % pre-indexing
                indexIII_copy = indexIII;
                for ccc = 1: length(indexIII) - 1
                    if (IData(ccc) == 0) && (indexIII(ccc+1) == 1)
                        indexIII_copy(ccc) = 1;                            % 
indexing: add the leading edge (pulse head) of the current pulse
                    end
                    if  (indexIII(ccc) == 1) && (IData(ccc+1) == 0)        % Make 
sure that current drops to ZERO; Only in this case, can the pulse end be 
considered
                        for p_index = ccc + 1 : length(indexIII) - 1
                            if IData(p_index) == 0 && IData(p_index +1) ~= 0
                                indexIII_copy(ccc + 1 : p_index) = 1;  % 
indexing: add the following edge of the current pulse, as well as the relaxation 
after the pulse
                                break
                            end
                        end
                    end
                end
        indexIII = indexIII_copy;
```

To understand the above codes as of how `indexIII` is produced, Fig. 15 gives graphical explanations. **As aforementioned, a complete current pulse is considered as the whole of pulse head, pulse plateau, pulse end and relaxation period, and is acquired by 'assembling' all the parts of a current pulse step by step**. Firstly, `indexIII = ((Current - Sensitivity) <= IData) + (IData <= (Current + Sensitivity)) == 2` finds the plateau part of the pulse. Then `if (IData(ccc) == 0) && (indexIII(ccc+1) == 1)` finds the head of the pulse. Then `if (indexIII(ccc) == 1) && (IData(ccc+1) == 0)` finds the end of the pulse. Finally, `if IData(p_index) == 0 && IData(p_index +1) ~= 0` finds the relaxation period following the pulse. currentIndexing.m aims to extract complete current pulses with all the parts together, but if it is impossible to do so due to the data itself, currentIndexing.m will extract incomplete current pulses with as many parts as possible.

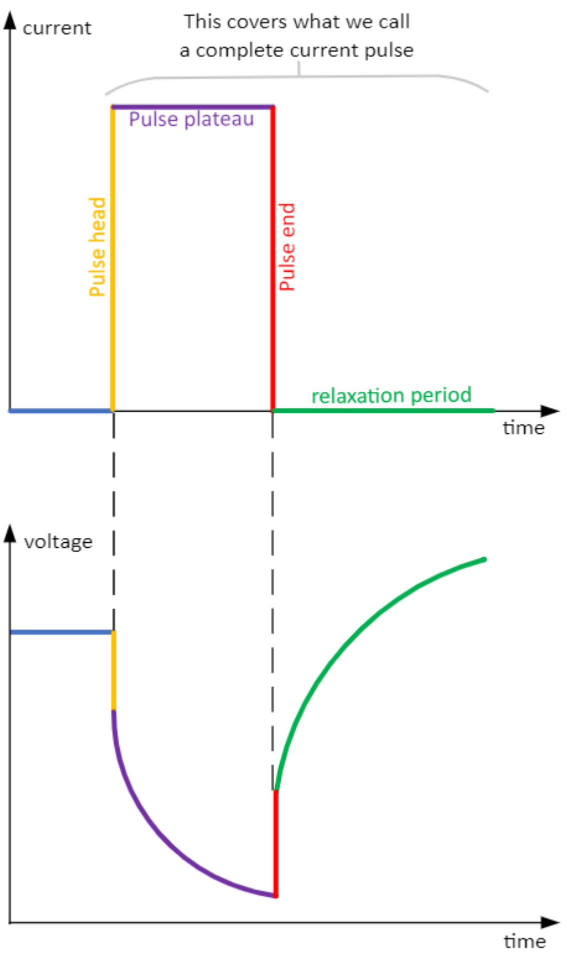


Fig. 15. Explanation of ‘current pulse assembling’, which is the functionality of currentIndexing.m

----

data_ currentDependent.m

```
function [dataIII, SOC_steps]= data_currentDependent (data, Currents_depended,
Currents_sensitivity, S_j , NRC)
```

data_currentDependent.m is the first line and the first function being called in AmpDenpOnly.m. **This function accepts SOC-dependent data segments (`data`) in the format of 1 by N cell, currents (`Currents_depended`) at which parameters will be solved respectively, as defined by the user in ㉖, sensitivity of current (`Currents_sensitivity`) which delimits error range of currents, as defined by the user in ㉗**, SOC division (`S_j`), and number of RC pairs (`NRC`).

This function returns the both SOC- and I-dependent data segments (`dataIII`) in the format of a 2D cell, with I being the index of rows and SOC being the index of columns, number of non-empty SOC divisions at each current (`SOC_steps`), i.e., number of useful columns in each row of `dataIII`.

----

data_ currentDependent.m

The second half of codes in data_ currentDependent.m is almost the same as those in data_temperatureDependent.m. Therefore, the following will not explain the second half but focus on the first half of codes, as below.

```
indexIII = currentIndexing (data.I{1,j}{k}, Currents_depended(i), 
Currents_sensitivity);

            if any(indexIII)
                [~, minlocs] = findpeaks(-indexIII);
                if ~isempty(minlocs)   % more than one current pusles

                    count = count + 1;  % The first current pusle
                    indexIII_temp = indexIII;
                    indexIII_temp(minlocs(1) : end) = false;
                    dataIII.I {i,j}{count,1} = data.I{1,j}{k}(indexIII_temp);
                    dataIII.t {i,j}{count,1} = data.t{1,j}{k}(indexIII_temp);
                    dataIII.T {i,j}{count,1} = data.T{1,j}{k}(indexIII_temp);
                    dataIII.V {i,j}{count,1} = data.V{1,j}{k}(indexIII_temp);
                    dataIII.S {i,j}{count,1} = data.S{1,j}{k}(indexIII_temp);

                    count = count + 1;  % The last current pulse
                    indexIII_temp = indexIII;
                    indexIII_temp(1 : minlocs(end)) = false;
                    dataIII.I {i,j}{count,1} = data.I{1,j}{k}(indexIII_temp);
                    dataIII.t {i,j}{count,1} = data.t{1,j}{k}(indexIII_temp);
                    dataIII.T {i,j}{count,1} = data.T{1,j}{k}(indexIII_temp);
                    dataIII.V {i,j}{count,1} = data.V{1,j}{k}(indexIII_temp);
                    dataIII.S {i,j}{count,1} = data.S{1,j}{k}(indexIII_temp);

                    if  length(minlocs) > 1   % The other current pulses except 
the first and the last ones
                        for hhh = 2: length(minlocs)
                            count = count + 1;
                            indexIII_temp = indexIII;
                            indexIII_temp(1 : minlocs(hhh - 1)) = false;
                            indexIII_temp(minlocs(hhh) : end) = false;
                            dataIII.I {i,j}{count,1} = 
data.I{1,j}{k}(indexIII_temp);
                            dataIII.t {i,j}{count,1} = 
data.t{1,j}{k}(indexIII_temp);
                            dataIII.T {i,j}{count,1} = 
data.T{1,j}{k}(indexIII_temp);
                            dataIII.V {i,j}{count,1} = 
data.V{1,j}{k}(indexIII_temp);
                            dataIII.S {i,j}{count,1} = 
data.S{1,j}{k}(indexIII_temp);
```

```
                end
            end

        else                        % only one current pulse
            count = count + 1;
            dataIII.I {i,j}{count,1} = data.I{1,j}{k}(indexIII);
            dataIII.t {i,j}{count,1} = data.t{1,j}{k}(indexIII);
            dataIII.T {i,j}{count,1} = data.T{1,j}{k}(indexIII);
            dataIII.V {i,j}{count,1} = data.V{1,j}{k}(indexIII);
            dataIII.S {i,j}{count,1} = data.S{1,j}{k}(indexIII);
        end
    end
```

As abovementioned, data_currentDependent.m gives a current index (`indexIII`) which indicates the locations of data points (current pulses) specific/useful to parameterisation at the target current. If `indexIII` is not empty `(if any(indexIII))`, it means that the present data segment contains at least one current pulse specific/useful to the target current. For consistent solving and post-processing purpose, multiple current pulses would better be separated into different data segments rather than mixed in one data segment. **To this end, the codes afterwards actually accomplish one mission – separate different current pulses and save each in `dataIII`**. `[~, minlocs] = findpeaks(-indexIII)` is used to detect the number of current pulses. `if ~isempty(minlocs)`, it means that there are more than one current pulses; otherwise, there is only one current pulse, so using `indexIII` as the index, the single current pulse can be directly saved in `dataIII`. `if ~isempty(minlocs)`, the codes above first save the first and last current pulses in `dataIII`, and then use `if length(minlocs) > 1` to detect if there are more current pulses except the first and last ones. If so, the loop `for hhh = 2:length(minlocs)` will extract and save the remaining current pulses in `dataIII`.

----

12. *AmpAndTempDenp.m* & *data_IandTdependent.m* & *PostProcess4CrateandTemps.m*

12.1 Functionality and structure

AmpAndTempDenp.m, diverted from ECM_easy.m, is a script to perform all of SOC-, T- and I-dependent parameterisation. This script is an aggregation of ECM_easy.m (SOC-), TempDenpOnly.m (T-), and AmpDenpOnly.m (I-).

data_ IandTdependent.m, called by AmpAndTempDenp.m, is a function to reform data segments from 1-D SOC-dependent to 3-D SOC-, T-, and I-dependent. In other words, this function applies temperature windows and current windows to the SOC-dependent data segments to produce new data segments that lie in different SOC, temperature, and current windows. This function is an aggregation of data_temperatureDependent.m (SOC- and T-) , and data_ currentDependent.m (SOC- and I-).

PostProcess4CrateandTemps.m, called by AmpAndTempDenp.m, is a script for post-processing the SOC-, T- and I-dependent parameters. It generates the look-up tables and plots of parameters at different SOCs, Ts and Is. This script is an aggregation of PostProcess4Temp.m (SOC- and T-), and PostProcess4Crate.m (SOC- and I-). There is nothing much to explain about the post-processing codes, as the codes are all about variable restructuring and visualisation - the best way to understand the codes is to look at the generated look-up tables and plots and try to link them back with the codes.

12.2 Core codes and hard-to-read codes

----

data_IandTdependent.m

```
function [dataIT, SOC_steps]= data_IandTdependent (data, Currents_depended,
Currents_sensitivity, Temps_depended, Temps_sensitivity, S_j , NRC)

% data reorganisation - specific to I
...
...
% data reorganisation - specific to T
...
...
dataIT = dataIT_temp
```

data_IandTdependent.m is the first line and the first function being called in AmpAndTempDenp.m. This function accepts SOC-dependent data segments (`data`) in the format of 1 by N cell, currents (`Currents_depended`) at which parameters will be solved respectively, as defined by the user in ㉖, sensitivity of current (`Currents_ sensitivity`) which delimits error range of currents, as defined by the user in ㉗ , temperatures (`Temps_depended`) at which parameters will be solved respectively, as defined by the user in ㉙ , sensitivity of temperature (`Temps_sensitivity`) which delimits error range of temperatures, as defined by the user in ㉚ , SOC division (`S_j`), and number of RC pairs (`NRC`).

This function returns the SOC-, T- and I-dependent data segments (`dataIT`) in the format of a 3D cell, with the first 2D being I-indexed rows and T-indexed columns, and the 3rd D being the SOC-dependent data segments. This function also returns the number of non-empty SOC divisions at each I&T combination (`SOC_steps`), i.e., number of useful columns in each row and column of `dataIT`.

The codes under `% data reorganisation - specific to I` firstly add I-dependence to the SOC-dependent data, which are almost the same as the codes in data_ currentDependent.m. Based on this, the codes under `% data reorganisation - specific to T` further add T-dependence. This part of codes is similar to that in data_temperatureDependent.m, however, the key difference is about data type - the result from this part of codes is a 3D cell rather than a 2D cell. Thus, the final output of this function (`dataIT`) is a 3D cell with SOC-, T- and I-dependence.

AmpAndTempDenp.m

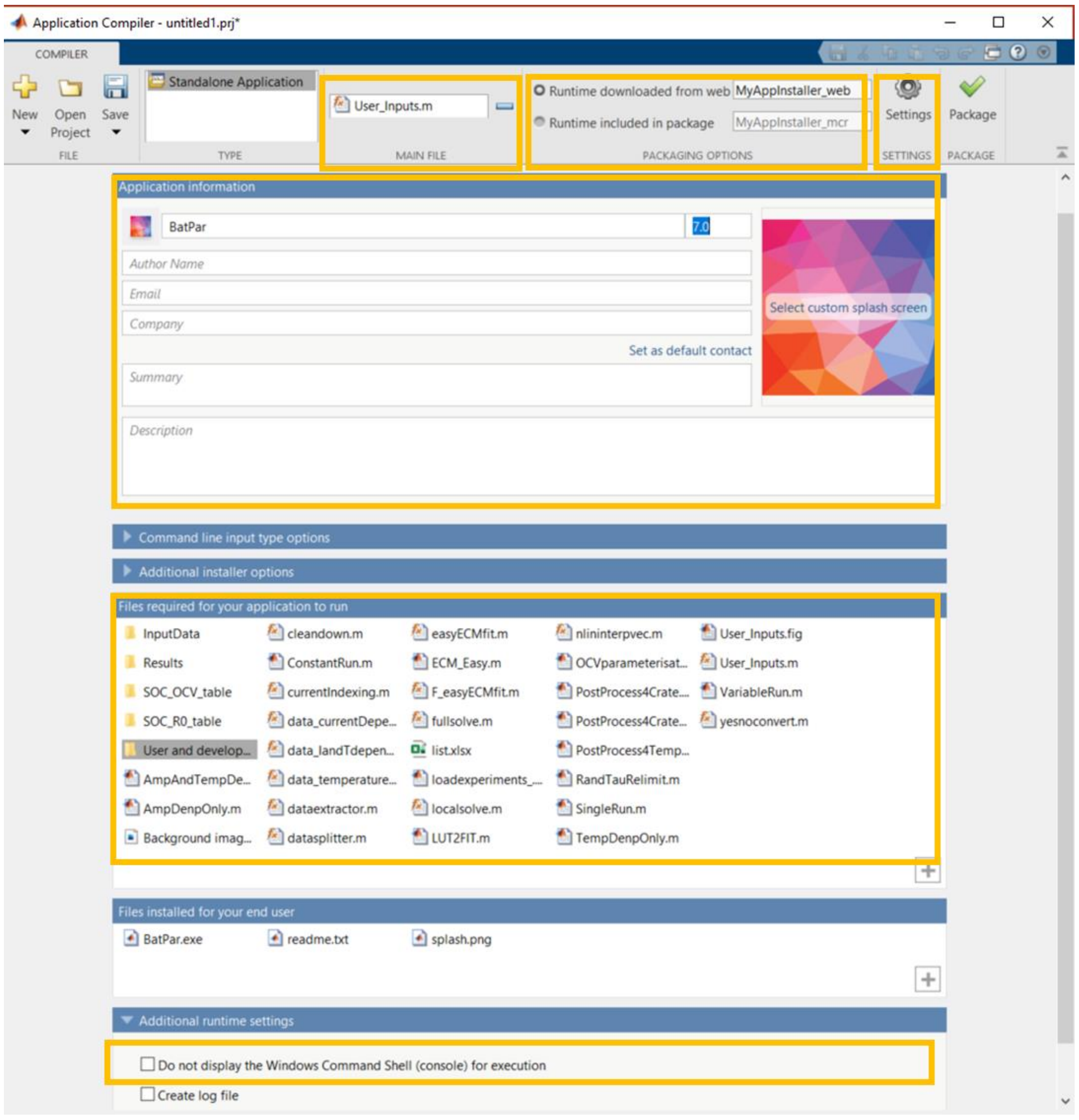
```
disp(['Initialisation done - Paramterisation (with CONSTANT tau and I dependence
and T dependence) in progress, including
',num2str(length(Currents_depended)*length(Temps_depended)), ' PHASES (',
num2str(length(Currents_depended)),' CURRENTS by
',num2str(length(Temps_depended)),' TEMPERATURES) --> Each PHASE further includes
2 STAGES --> Each STAGE further includes ~',num2str(max(max(SOC_steps))),'
STEPS.'])
...
...
for iiii = 1:length(Currents_depended)
    for jjjj = 1:length(Temps_depended)
    ...
    ...
```

The codes above display the progress of I- and T-dependent parameterisation in the MATLAB command window, and as such introduce the concepts of ‘`PHASE`’, ‘`STAGE`’, and ‘`STEP`. ‘`PHASE`’ represents the number of I&T combinations. E.g., if you have 3 currents and 5 temperatures, then there will be 15 phases. Each phase will be parameterised on by one, which is looped by `for iiii = 1:length(Currents_depended)` and `for jjjj = 1:length(Temps_depended)`. ‘`STAGE`’ underlies ‘`PHASE`’, and there will be one stage if variable TAU is configured for parameterisation, or two stages if constant TAU is configured. ‘`STEP` underlies ‘`STAGE’`, and represents the number of SOC divisions.

----

13. Generating BatPar installer

You may be aware that BatPar was packaged into an executable program, which can be installed through an installer. The major benefit of converting BatPar from MATLAB codes to a packaged .exe program is about protection of source codes. Once packaged, the user will not be able to see the source codes but can still use the full functionalities. To generate such an installer, guides are provided below.


New
Open Project
Save
FILE
TYPE
MAIN FILE
PACKAGING OPTIONS
Settings
SETTINGS
Package
PACKAGE
Select custom splash screen
Set as default contact
Command line input type options
Additional installer options
Additional runtime settings

Application Compiler - untitled1.prj*

COMPILER

Standalone Application

User_Inputs.m

Runtime downloaded from web: MyAppInstaller_web

Runtime included in package: MyAppInstaller_mcr

Application information

BatPar 7.0

Author Name

Email

Company

Summary

Description

Files required for your application to run

InputData	cleandown.m	easyECMfit.m	nlininterpvec.m	User_Inputs.fig
Results	ConstantRun.m	ECM_Easy.m	OCVparameterisat...	User_Inputs.m
SOC_OCV_table	currentIndexing.m	F_easyECMfit.m	PostProcess4Crate...	VariableRun.m
SOC_R0_table	data_currentDepe...	fullsolve.m	PostProcess4Crate...	yesnoconvert.m
User and develop...	data_landTdepen...	list.xlsx	PostProcess4Temp...	
AmpAndTempDe...	data_temperature...	loadexperiments_...	RandTauRelimit.m	
AmpDenpOnly.m	dataextractor.m	localsolve.m	SingleRun.m	
Background imag...	datasplitter.m	LUT2FIT.m	TempDenpOnly.m	

Files installed for your end user

BatPar.exe readme.txt splash.png

☐ Do not display the Windows Command Shell (console) for execution

☐ Create log file

Fig. 16. Use application complier to generate user-oriented installer of BatPar.

1. Use 'deploytool' command to open the application complier, as Fig. 16.

2. Select User_inputs.m as the main file.

3. It is recommended to tick on ‘Runtime downloaded from Web’. In this way the size of generated installer can be much smaller. Otherwise, you will see a large installer of several GB.

4. Click ‘settings’ to configure the output folders

5. Fill in the blanks in ‘application information’

6. In ‘Files required for your application to run’, put every file of BatPar except the main file (User_inputs.m).

7. Tick OFF ‘Do not display the Windows Command Shell (Console) for execution’ – because the console will display useful real-time parameterisation progress, like the MATLAB command window.

8. Click ‘Package’ – then you will find the installer in the output folder(s).

With this BatPar installer, users do not even need to install MATLAB – but it is not clear if it is legal to use the installer in the absence of MATLAB licences – it is the user’s responsibility.

===============This is the end of BatPar developer manual===============