LIONSIMBA Toolbox – Li-ION SIMulation BAttery Toolbox

Marcello Torchio[*], *Lalo Magni*[†], *Bhushan Gopaluni*[‡], *Richard D. Braatz*[§] *and Davide M. Raimondo*[¶]

October 2018

Version 1.024

[*] marcello.torchio01@ateneopv.it
[†] lalo.magni@unipv.it
[‡] bhushan.gopaluni@ubc.ca
[§] braatz@mit.edu
[¶] davide.raimondo@unipv.it

Contents

1 Introduction

1.1 Prerequisites

This manual is a brief user guide for the installation and usage of the LIONSIMBA Toolbox developed by Torchio et al. [1]. LIONSIMBA implements the well known theory-based pseudo two-dimensional (P2D) model developed and validated by the authors in [2], representing the electrochemical phenomena occurring inside a Li-ion cell. The toolbox has been implemented using the scientific scripting language Matlab and makes use of the Sundials suite [3] in order to solve the set of resulting highly nonlinear and tightly coupled Differential and Algebraic Equations (DAEs). The package is freely available at the following link:

`http://sisdin.unipv.it/labsisdin/lionsimba.php`

In the following, the software requirements are listed:

- Matlab R2014b or higher (Windows, Os-X or Linux versions)
- Sundials 2.6.2 (**versions of Sundials newer than 2.6.2 do not present a Matlab interface**).
- CasADi 3.1 (**Please make use of this version.**)

The aforementioned required tools can be obtained respectively at the following links:

- `http://www.mathworks.com/` - Mathworks Matlab home page
- `https://computation.llnl.gov/casc/sundials/main.html` - SUNDIALS suite home page.
- `https://github.com/casadi/casadi/wiki/InstallationInstructions` - CasADi installation instructions for Matlab

1.2 Download, Installation and Configuration of SUNDIALS

The SUNDIALS suite has recently deprecated the support to the interface for Matlab. **To correctly run the LIONSIMBA toolbox the SUNDIALS version 2.6.2 is needed.** This is the last version of SUNDIALS which has the support for the Matlab interface. In the following, a step-by-step guide is given for the correct installation of SUNDIALS

1. Download the SUNDIALS package 2.6.2 from the following URL:
   `http://computation.llnl.gov/projects/sundials/download/sundials-2.6.2.tar.gz`
   Once downloaded, unpack the tar.gz file into a directory.
2. Start Matlab and select **sundialsTB** as the current working folder.
3. Run the script **install_STB.m** located in the **sundialsTB** folder. This step will start a compilation procedure for the SUNDIALS interface for Matlab. Note that the user needs a Matlab compatible compiler to accomplish this step. Please refer to the Mathworks website in order to figure out the compatible compilers for your version of Matlab. When asked, answer *yes* for the compilation of the IDA interface.
4. Once the procedure issued by **install_STB.m** has finished, add the folder created by the installation script to the Matlab path.

5. SUNDIALS is correctly installed and configured to work with Matlab and LIONSIMBA.

Information about how to install and configure the SUNDIALS suite with Matlab can be obtained from the SUNDIALS User's guide.

In the following a list of common problems and corresponding solutions in the SUNDIALS installation is given.

1.2.1 SUNDIALS installation problems and solutions on Windows, Mac and Linux

On **Windows** platforms, a suitable compiler must be installed if not present, for instance:

- **MinGW-w64**, which is freely available at the following link: `https://it.mathworks.com/matlabcentral/fileexchange/\52848-matlab-sup +-compiler`.

  or selecting *Add On* in the Matlab home, then select **MATLAB Support for MinGW-w64 C/C++ Compiler** and finally **Download**. There can be problems with the download of third-party software in Matlab 2017a or earlier. In case of problems follow the **Installation instructions** in *Bug Report* .

- **Compiler SDK 7.1**, which is freely available at the following link: `https://developer.microsoft.com/it-it/windows/downloads/windows-10-sdk`

On **Mac** platforms, the compiler Xcode should be already installed. However, SUNDIALS installation may require the following changes in the script *kim.c*

- line 682:

  if (kimData == NULL) return; $\rightarrow$ if (kimData == NULL) return NULL;

- line 810:

  return; $\rightarrow$ return NULL;

1.3 Package structure

The files of LIONSIMBA are organized in different folders as follows:

```
LIONSIMBA Root Folder
├── User's Manual
├── battery_model_files
│   ├── external_functions
│   ├── interpolation_scripts
│   ├── numerical_tools
│   ├── P2D_equations
│   └── simulator_tools
└── example_scripts
```

The set of parameters used for simulation are reported inside the file *Parameters_init.m*. Within the P2D_equations folder, besides the numerical schemes used to implement LIONSIMBA, the user can find all the scripts that allow the customization of the electrochemical dynamics of the Li-ion cell.

- *electrolyteDiffusionCoefficients.m*: contains the analytical formula which provides the values of the diffusion coefficients inside the electrolyte phase.

- *electrolyteConductivity.m*: contains the analytical formula which provides the values of the electrolyte conductivity coefficients.
- *openCircuitPotential.m*: contains the analytical formula which provides the values of the Open Circuit Voltage (OCV) of the electrodes.
- *reactionRates.m*: contains the code able to provide the reaction rates coefficient for the ionic flux computation.
- *solidPhaseDiffusionCoefficients.m*: contains the code able to compute the solid phase diffusion coefficients.
- *getInputCurrent.m*: contains the code which analytically describes a variable profile of the current density. The user can implement its own generic non-linear function.

The toolbox comes with the analytical formulae for the different coefficients related to the particular Li-ion cell chemistry presented in [4].

1.4 Installation

In order to install LIONSIMBA, the root folder and its subfolders have to be added to the Matlab path. By setting the current working directory of Matlab as the LIONSIMBA Root folder, perform this action from the Matlab prompt:

```
% Add the current current working directory and its
    subfolders to the Matlab path
addpath(genpath(pwd))
% Save the Matlab path
savepath
```

1.5 Releases, bugs report and comments

All new releases of LIONSIMBA will be published on the project's web page:

`http://sisdin.unipv.it/labsisdin/lionsimba.php`

For bugs report or comments on this work, please send an e-mail to marcello.torchio01@ateneopv.it or davide.raimondo@unipv.it

2 Li-ion cell electrochemical model

In this section a brief description of numerical approach used to implement the P2D model is provided. Firstly an introduction to the set of non-linear Partial Differential and Algebraic Equations (PDAEs) is provided. Then the numerical implementation is addressed.

2.1 Pseudo two-dimensional Model

As shown in Fig. 1, the Li-ion cell is composed of five main sections: the positive current collector (a), the cathode (p), the separator (s), the anode (n), and the negative current collector (v). The index $z \in \{a, p, s, n, v\}$ is used to indicate the different sections. The thickness of each battery section is defined by L_z and $L := \sum_z L_z$ represents the overall battery thickness. The electrodes and the porous separator of the Li-ion cell are placed in contact with an electrolyte solution, facilitating the flow of ions during charging and discharging processes. During a charging process,

the ions deintercalate from the positive electrode and, passing through the porous media of the separator, intercalate into the negative electrode. The inverse process occurs when discharging the cell. The diffusion of

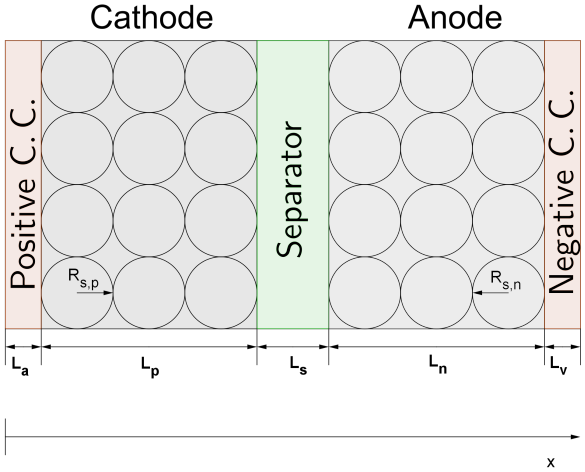

Figure 1: 1D diagram of a Li-ion cell.

ions within the electrodes are modeled by

$$\frac{\partial}{\partial t} c_s^{\text{avg}}(x,t) = -\frac{3}{R_s} j(x,t) \tag{1}$$

$$c_s^*(x,t) - c_s^{\text{avg}}(x,t) = -\frac{R_s}{5 D_{\text{eff}}^s} j(x,t), \tag{2}$$

where $t \in \mathbb{R}^+$ represents the time, $x \in \mathbb{R}$ is the one-dimensional spatial variable, $c_s^*(x,t)$ and $c_s^{\text{avg}}(x,t)$ are the surface and average concentration of solid particles respectively, the function $j(x,t)$ represents the ionic flux, and R_s and D_{eff}^s account for the particle radius and effective diffusion coefficients of the solid phases. The bulk State of Charge (SOC) of the anode is defined as

$$\text{SOC}(t) := \frac{1}{L_n \, c_s^{\text{max,n}}} \int_0^{L_n} c_s^{\text{avg}}(x,t)\, dx,$$

where $c_s^{\text{max,n}}$ represents the maximum concentration of Li-ions in the negative electrode. The flow of ions inside the electrolyte solution is modeled by a diffusion equation,

$$\epsilon \frac{\partial}{\partial t} c_e(x,t) = \frac{\partial}{\partial x}\left[D_{\text{eff}} \frac{\partial c_e(x,t)}{\partial x} \right] + a(1-t_+) j(x,t), \tag{3}$$

where $c_e(x,t)$ represents the electrolyte concentration of ions, t_+ defines the transference number, a is the particle surface area to volume ratio, D_{eff} accounts for the effective diffusion coefficients in the electrolyte, and ϵ represents the material porosity. According to Ohm's law, the conservation of charge in the electrodes can be defined as

$$\frac{\partial}{\partial x}\left[\sigma_{\text{eff}} \frac{\partial}{\partial x} \Phi_s(x,t) \right] = aF j(x,t), \tag{4}$$

where $\Phi_s(x,t)$ is the solid potential, σ_{eff} is the electrodes effective conductivity, and F is the Faraday's constant. The potential of the Li-ion cell is obtained as

$$V_{\text{out}}(t) := \Phi_s(0,t) - \Phi_s(L,t).$$

Similarly, a modified Ohm's law is used to represent the charge conservation within the electrolyte:

$$aFj(x,t) = -\frac{\partial}{\partial x}\left[\kappa_{\text{eff}}\frac{\partial}{\partial x}\Phi_e(x,t)\right] + \frac{\partial}{\partial x}\left[\frac{2\kappa_{\text{eff}}\mathrm{R}T(x,t)}{F}(1-t_+)\frac{\partial}{\partial x}\ln c_e(x,t)\right], \tag{5}$$

where $\Phi_e(x,t)$ is the electrolyte potential, $T(x,t)$ represents the temperature, R defines the universal gas constant, and κ_{eff} is the effective conductivity of the liquid phase. The temperature dynamics are modeled by an energy balance,

$$\rho C_p\frac{\partial}{\partial t}T(x,t) = \frac{\partial}{\partial x}\left[\lambda\frac{\partial}{\partial x}T(x,t)\right] + Q_{\text{ohm}}(x,t) + Q_{\text{rxn}}(x,t) + Q_{\text{rev}}(x,t), \tag{6}$$

where ρ is the material density, C_p is the specific heat, λ is the heat diffusion coefficient, and the terms $Q_{\text{ohm}}(x,t)$, $Q_{\text{rev}}(x,t)$, and $Q_{\text{rxn}}(x,t)$ account for ohmic, reversible, and reaction heat sources as shown in [5].

The above equations are coupled by means of the ionic flux, which is defined by the Butler-Volmer equation

$$j_{\text{int}}(x,t) = 2\frac{i_{0,\text{int}}}{F}\sinh\left[\frac{0.5F}{\mathrm{R}T(x,t)}\eta_{\text{int}}\right], \tag{7}$$

where the exchange current density is given by

$$i_{0,\text{int}} = Fk_{\text{eff}}\sqrt{c_e(x,t)(c_s^{\max}-c_s^*(x,t))c_s^*(x,t)}.$$

The overpotential at the anode side is defined as

$$\eta_{\text{int}} := \Phi_s(x,t) - \Phi_e(x,t) - U_n,$$

while at the cathode side as

$$\eta_{\text{int}} := \Phi_s(x,t) - \Phi_e(x,t) - U_p$$

where the terms U_p and U_n represent the cathode and anode OCV respectively, and k_{eff} represents the effective kinetic reaction rate. The intercalation flux $j_{\text{int}}(x,t)$ is zero inside the separator.

2.2 Numerical implementation

The overall set of nonlinear and tightly coupled Partial Differential Algebraic Equations (PDAEs) is used to represent all the electrochemical phenomena occurring in the Li-ion cell. A more detailed description of the model with the complete set of equations, numerical discretization and the set of parameters used in this work can be found in [1]. The resulting set of PDAEs together with the BCs is reformulated as a set of Differential Algebraic Equations (DAEs). The spatial domain is discretized according to the Finite Volume Method (FVM), whereas the time domain is left continuous, according to the Method of Lines (MOL) as discussed in [6]. Therefore the spatial domain is subdivided into $N_a + N_p + N_s + N_n + N_v$ control volumes. The resulting set of DAEs is then fed to the IDA solver from the SUNDIALS suite.

3 Configurable scripts and parameters

3.1 Configurable parameters

As mentioned in Section 1.3, the file *Parameters_init.m* contains parameters used to characterize both the Li-ion cell and the simulator settings. By running this script a Matlab structure is returned. **Note that the returned value must be put in a cell variable.** For instance:

```
param{1} = Parameters_init;
```

This structure will contain all the parameters and their values as defined in *Parameters_init.m*. If the cell array would contain more than one parameters structure, then the simulation will perform a battery pack simulation where series-connected cells will be considered.

```
param{1} = Parameters_init;
param{2} = Parameters_init;
...
param{n} = Parameters_init;
```

Note that the user can't remove or add new fields to the original parameters list. The description of every single parameter is reported in the Matlab source file.

3.2 Solid-phase diffusion models

LIONSIMBA allows the user to chose among three different models for the solid-phase diffusion:

- Fick's law diffusion equation (including the pseudo-second dimension r):

$$\frac{\partial c_s(r,t)}{\partial t} = \frac{1}{r^2}\frac{\partial}{\partial r}\left[r^2 \boldsymbol{D}_{\mathbf{eff}}^{\mathrm{s}} \frac{\partial c_s(r,t)}{\partial r}\right]$$

  with boundary conditions

$$\left.\frac{\partial c_s(r,t)}{\partial r}\right|_{r=0} = 0 \qquad \left.\frac{\partial c_s(r,t)}{\partial r}\right|_{r=R_p} = -\frac{j(x,t)}{\boldsymbol{D}_{\mathbf{eff}}^{\mathrm{s}}}$$

- two-parameters polynomial approximation [7]:

$$\frac{\partial c_s^{\mathrm{avg}}(x,t)}{\partial t} = -3\frac{j(x,t)}{R_p}$$
$$c_s^*(x,t) - c_s^{\mathrm{avg}}(x,t) = -\frac{R_p}{\boldsymbol{D}_{\mathbf{eff}}^{\mathrm{s}}}\frac{j(x,t)}{5}$$

- higher-order polynomial approximation [7]:

$$\frac{\partial c_s^{\mathrm{avg}}(x,t)}{\partial t} = -3\frac{j(x,t)}{R_p}$$
$$\frac{\partial q(x,t)}{\partial t} = -30\frac{\boldsymbol{D}_{\mathbf{eff}}^{\mathrm{s}}}{R_p^2}q(x,t) - \frac{45}{2}\frac{j(x,t)}{R_p^2}$$
$$c_s^*(x,t) - c_s^{\mathrm{avg}}(x,t) = -\frac{j(x,t)R_p}{35\boldsymbol{D}_{\mathbf{eff}}^{\mathrm{s}}} + 8R_p q(x,t)$$

The choice of the solid-phase diffusion model can be set by changing the value of the parameters *SolidPhaseDiffusion* inside *the Parameters_init.m*.

Listing 1: Solid-phase diffusion model selection

```
param{1} = Parameters_init;
% Two-terms model
param{1}.SolidPhaseDiffusion = 1;
% Higher-order model
param{1}.SolidPhaseDiffusion = 2;
% Fick's law of diffusion
param{1}.SolidPhaseDiffusion = 3;
```

3.3 Battery pack simulations

LIONSIMBA allows to simulate battery packs, in particular pack where series-connected cells are present. In order to do this just define a cell array containing several parameters structure, e.g.:

```
param{1} = Parameters_init;
param{2} = Parameters_init;
param{3} = Parameters_init;
```

Then each parameters structure can be modified, thus leading to independent scenarios:

```
% Change the cathode thickness of cell #1
param{1}.len_p = 5e-6;
% Reduce the initial anode SOC for cell #2
param{2}.cs_initn = 0.8*param{2}.cs_initn;
% Change the cathode porosity for cell #3
param{3}.eps_p = 0.55;
```

In order to start a simulation of series-connected cells it is sufficient to type:

```
out = startSimulation(0,3000,initialState,I,param);
```

In this way a battery pack will be simulated from 0 to 3000 seconds, with the initial states defined by initialState demanding (or providing) a current density equal to I, with each cell parametrized according to the elements of the param structure.

3.4 Configurable scripts

Besides parameters, some scripts can be modified in order to meet the simulation needs of different scenarios.

This imply that it is possible to customize the functions used for the computation of the diffusion coefficients in the electrolyte, the electrolyte conductivities and so on. This leads to test different chemistries and different types of Li-ion cells.

All the editable scripts provided in the package have an extra help text. All the general rules that have to be followed in order to correctly implement a custom defined function are reported.

For more information about the editable scripts, type:

```
help script_name
```

in the Matlab command line.

3.5 Run simulations

In order to start simulations, the user must call the *startSimulation* script from the Matlab command line. This script, after checking for the presence of the required tools, starts the simulation. The resulting output will be a structure; this output will contain all the solutions of the dependent variables and other parameters. Type *help startSimulation* from the command line in order to get more information about the output structure.

The *startSimulation* function needs four parameters:

```
results = startSimulation(t0,tf,initState,
    InputDensity,param)
```

where:

- t0 [s]: initial integration time (Mandatory)
- tf [s]: final integration time (Mandatory)
- initState : initial state structure (Can be empty [ ])
- InputDensity [A/m^2] or [W/m^2]: applied current/power density (Mandatory for constant current/power simulation scenarios)
- param : set of parameters (Can be empty [ ] or an cell array containing several parameters structure.)

The parameter *initialState* can be passed to the simulator in order to define a set of initial states from which the simulations will start. It could be useful for simulating pulse charge-discharge effects or for applying some control-oriented algorithms. If empty, then the simulator will start from Consistent Initial Conditions (CICs) according to the parameters defined in the *Parameters_init.m* script.

The applied current/power density *InputDensity* is used respectively by the simulator only for simulations in galvanostatic/costant input power conditions. Note that according to this implementation, **positive current densities make the battery charging, while negative current densities make the battery discharging**.

In the *Parameters_init.m* script it is possible to define five operational modes (*param1.OperationMode=...*):

- Galvanostatic input current
- Costant input power
- Potentiostatic
- Variable input current
- Variable input power

If the first operational mode is chosen, the value of the parameter *InputDensity* will be used as a fixed current density applied to the battery for the whole simulation from t_0 to t_f. The second operational mode set the value of the input power *InputDensity* and change the values of the current and the voltage of the simulation. If the potentiostatic mode is selected, the simulator will apply to the battery a variable cur-

rent profile which will keep the voltage at the end of the cell fixed to the value set in the *Parameters_init.m* script. Like the variable current profile, the parameter *InputDensity* is neglected while operating in potentiostatic mode. In case of variable input profile, the simulator will not make use of the value passed in the parameter *InputDensity*, but it will get the value of the applied current density through the script *getInputCurrent.m*. Finally, the Variable input power will make use of the power profile described in the file *getInputPowerDensity.m*.

3.6 Rootfinding feature for discontinuities handling in the applied current

From version 1.024 of LIONSIMBA, the software has been upgraded in order to handle situations in which the applied current density presents discontinuities. The application of current profiles that exhibit significant discontinuities between two adjacent time steps, could provoke the integrator to crash because the integration tolerances are not met. For this reason, when the flag

```
AppliedCurrent
```

of the parameters structure is set to **2**, the simulator will be able to identify whenever a discontinuity will take place in the applied current profile and will handle it accordingly. The car cycling example has been extended with a second version which shows how to make use of this new feature.

3.7 Feedback-based custom current profiles

From LIONSIMBA1.024 it is possible to define the custom profile of the current density as a function of the battery internal states. Indeed, when running simulations with the parameters flag **AppliedCurrent** set to **2**, the function used to determine the value of the current density accepts as input also the data related to the values of the internal states at time instant t. This extra information can be used to provide a value of I_{app} which is a function of the battery states. The example script **Proportional_voltage_control.m** has been added to show this new feature. More information can be found in the script.

3.8 Analytical Jacobian support

From version 1.022 of LIONSIMBA, an important feature has been added. In particular, thanks to the support of the CasADitoolbox, the integration process of the P2D model has been significantly speeded up by means of the evaluation of the analytical form of the Jacobian matrix of the entire PDE model. CasADiprovides a Matlab interface for working with symbolical variables and very efficient routines for the evaluation of analytical derivatives [8]. The parameter which enables the evaluation of the Jacobian matrix is the **UseJacobian** flag present in the **Parameters_init.m** file. To enable the evaluation of the Jacobian, type

```
param{1} = Parameters_init;
param{1}.UseJacobian = 1;
```

With this flag enabled, when starting a simulation with the function **startSimulation**, LIONSIMBAwill firstly evaluate the analytical Jacobian of the P2D model and automatically will make use of it to provide support to the integration process. Moreover, in order to gain the maximum performance, at the end of each simulation cycle the Jacobian

function is returned among the simulation results. In this way, the user can set the **JacobianFunction** field of the parameters structure to avoid the recalculation of the analytical Jacobian matrix.

Suppose to run a simulation, and to assign the results into the variable **result**. By defining

```
param{1}.JacobianFunction = result.JacobianFun;
```

the Jacobian matrix will be stored and ready for future usage. In fact, if another simulation is needed (under the assumption that no changes have been made to the model), a new simulation can be run with the parameters structure containing already the analytical Jacobian matrix.

Remark: The same Jacobian matrix can be reused across several simulations **if and only if** the model structure and/or parameterization have not been changed between one run and the next one. As soon as the model structure (or parameterization) changes, a new Jacobian matrix has to be evaluated otherwise the previous one would point to a wrong model structure (or parameterization). In case of modifications of the model it is sufficient to leave the **JacobianFunction** field empty, and LIONSIMBA will automatically reevaluate the Jacobian matrix. The **CC_CV_charge.m** example script makes use of this approach to carry out the entire simulation.

Note: to ensure full compatibility, the CasADi matlab package is required [8]. This software is freely available at `https://github.com/casadi/casadi/wiki/InstallationInstructions`

3.9 Output data

At the end of each simulation, a data structure is returned. This structure will contain all the results related to the particular scenario defined. Each field of the structure, moreover, is a cell array. For instance:

```
% Define two parameters structure to simulate a
    battery pack
param{1} = Parameters_init;
param{2} = Parameters_init;
% Simulate 1000 s at -30 A/m2
out = startSimulation(0,1000,[],-30,param);
% Plot cell #1 voltage
plot(out.Voltage{1})
% Hold the figure and plot the cell #2 voltage
plot(out.Voltage{2},'--')
```

By indexing the output structure fields, the user will access to the variables of the different cells. When a single cell is simulated, only one element of the cell array will be present.

In order to have a full list of output fields, please type

```
help startSimulation
```

from the Matlab command line.

4 Test Simulations

Simulation results were obtained using Matlab R2014b on a Windows 7@3.2GHz PC with 8 GB of RAM for the experimental battery parame-

ters in [4] with a cutoff voltage of 2.5 V and environmental temperature of 298.15 K. For the proposed chemistry, the 1C value is ≈ 30 A/m^2. The effectiveness and ease of use of the proposed framework is shown.

In the first scenario (Fig. 2), -1C discharge simulations are compared for a very wide range of heat exchange coefficient h, with high h being the most challenging for retaining numerical stability in dynamic simulations. As expected, decreasing the value of the parameter h leads to a faster increase of the cell temperature. Moreover, due to the coupling of all the governing equations, it is possible to note the influence of different temperatures on the cell voltage. In the second scenario (Fig. 3), for a fixed value of $h = 1$ W/(m^2 K), different discharge cycles are compared at -0.5C, -1C, and -2C. According to the different applied currents, the temperature rises in different ways; it is interesting to note the high slope of the temperature during a -2C discharge, mainly due to the electrolyte concentration c_e being driven to zero in the positive electrode by the high discharge rate. In the third scenario, the framework is used to simulate a hybrid charge-discharge cycle, emulating the throttle of a HEV. During breaking, the battery gets charged. In Fig. 4 it is possible to analyze the battery response under a hybrid charge-discharge cycle. The solid potential behavior is primarily due to the different applied C rates, with discontinuous changes producing voltage drops. Different slopes of the voltage curve are related to the different C rates applied. Temperature rise is recorded in the first 50 seconds of simulations, which are followed by a slight decrease of the temperature mainly due to the exchange of heat with the surrounding environment ($h = 1$ W/(m^2 K)) and due to the lower current density applied. At around 250 s, temperature starts to increase due to the -1C rate applied during moderate speed; high slope of increase at around 410 s is due to the higher value of the discharge current which during an overtake reaches the value of -2C. Returning to moderate speed makes the temperature slope more gentle. During the last 10 seconds, temperature decreases due to the significant change in applied current and due to dissipation of heat with surrounding ambient.

In Fig. 5, the application of an ABMS is addressed. In this particular simulation, a model predictive control algorithm [9] is adopted to drive the SOC of the battery to a given value, while accounting for input and output constraints. The initial SOC was around 20% and its reference value was set to 85%. According to LIONSIMBA, the estimation of the SOC can be easily carried out by defining a custom function. In this particular scenario, the SOC has been computed as

$$\mathrm{SOC}(t) = \frac{1}{l_n \, c_{s,n}^{\max}} \int_0^{l_n} c_s^{\mathrm{avg}}(x,t)\, dx$$

The temperature maximum bound was set to $313.5K$, with the voltage set to 4.2 V. The BMS applies a current density which is almost fixed at 1C value for the entire simulation, while starting to drop as the SOC approaches its final value. The behavior of the SOC is almost linear during the first 2500 s, while starting to change according to the current drop, in order to approach smoothly the final stage of charge. In Fig. 6 it is possible to see that, according to the different charging stages, the electrolyte concentration diffuses in different ways. Starting from $c_e^{\mathrm{init}} = 1000$ mol/m^3, the input current induces a drop of concentration within the battery sections due to the diffusion of ions from the cathode to the anode. Approaching the final stage of charging, the concentration starts to converge back to the initial value of 1000 mol/m^3 and, around

5500 s, reaches the steady value. This behavior emphasizes the property of the FVM to conserve properties within numerical roundoff.

In Fig. 7, simulations have been run disabling the thermal dynamics leading to an isothermal environment. This particular configuration can be exploited in order to assess the influence of different constant temperatures at which the battery can operate.

In Fig. 8, it is simulated a discharge-charge cycle with the use of constant input power density. In particular, for the temperature dynamic the lumped thermal model has been used. For the first 1500 s the power is set to different negative values, that increase the temperature especially when the power density is set a higher value. Even if the power remain negative from 1000 s and 1500 s, the temperature starts to decrease due to the high different of power set. The current plot follows the power cycle and the slope of this parameter is horizontal only in the resting phase(from 1500 s and 2000 s and from 2500 s and 3000 s), while when the power has higher values the current has an higher slope.

In Fig. 9, it is simulated a sinusoidal charge-discharge cycle of variable input power density. The SOC decreases on every period of constant value that is due to the phase shift put initially in the function. The temperature has little jumps that are due to the continuously change in the sign of the power.

All the results of the proposed simulations can be reproduced by running the example scripts available with LIONSIMBA.

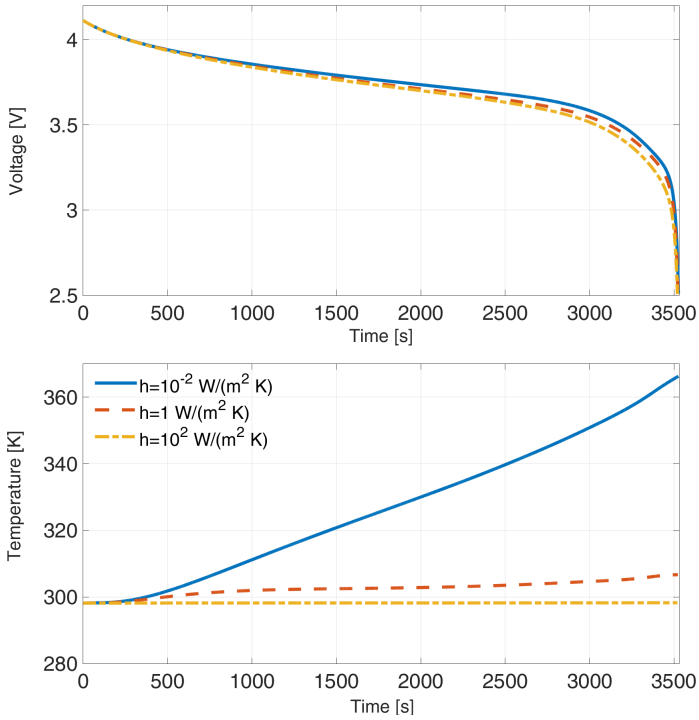

Figure 2: -1C discharge cycle run under different heat exchange parameters: blue line $h = 0.01\text{W}/(\text{m}^2\ \text{K})$, dashed orange line $h = 1\ \text{W}/(\text{m}^2\ \text{K})$ and dot-dashed yellow line $h = 100\ \text{W}/(\text{m}^2\ \text{K})$.

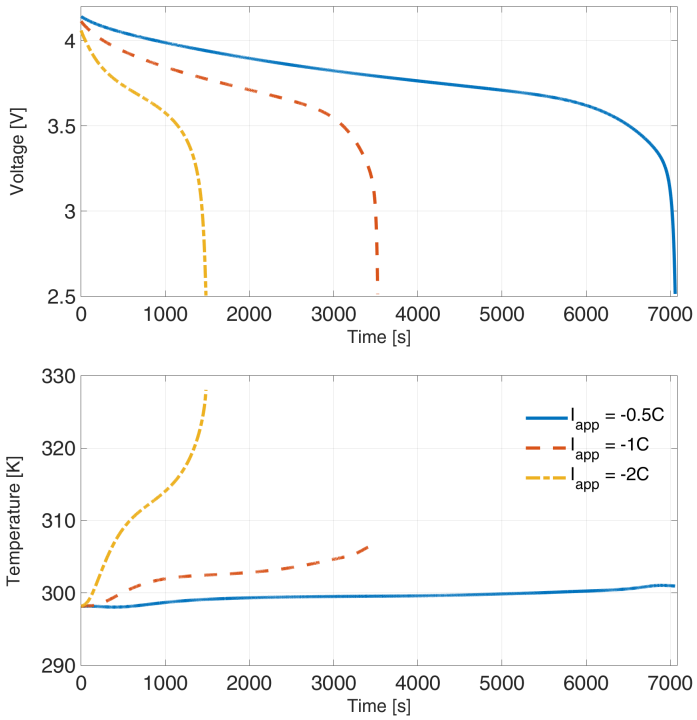

Figure 3: Full discharge cycle run under different C rates: -2C (dot-dashed yellow), -1C (dashed orange line), and -0.5C (blue line).

5 Acknowledgement

Grateful acknowledgement is made to Prof. Gregory Offer, Ian Campbell and Krishnakumar Gopalakrishnan (Imperial College, London) for their contribution in the development of LIONSIMBA 2.0. Grateful acknowledgement is also made to Alessio Stefanini and Andrea Pozzi for their support in the user's guide maintenance and in the beta testing of LIONSIMBA 2.0.

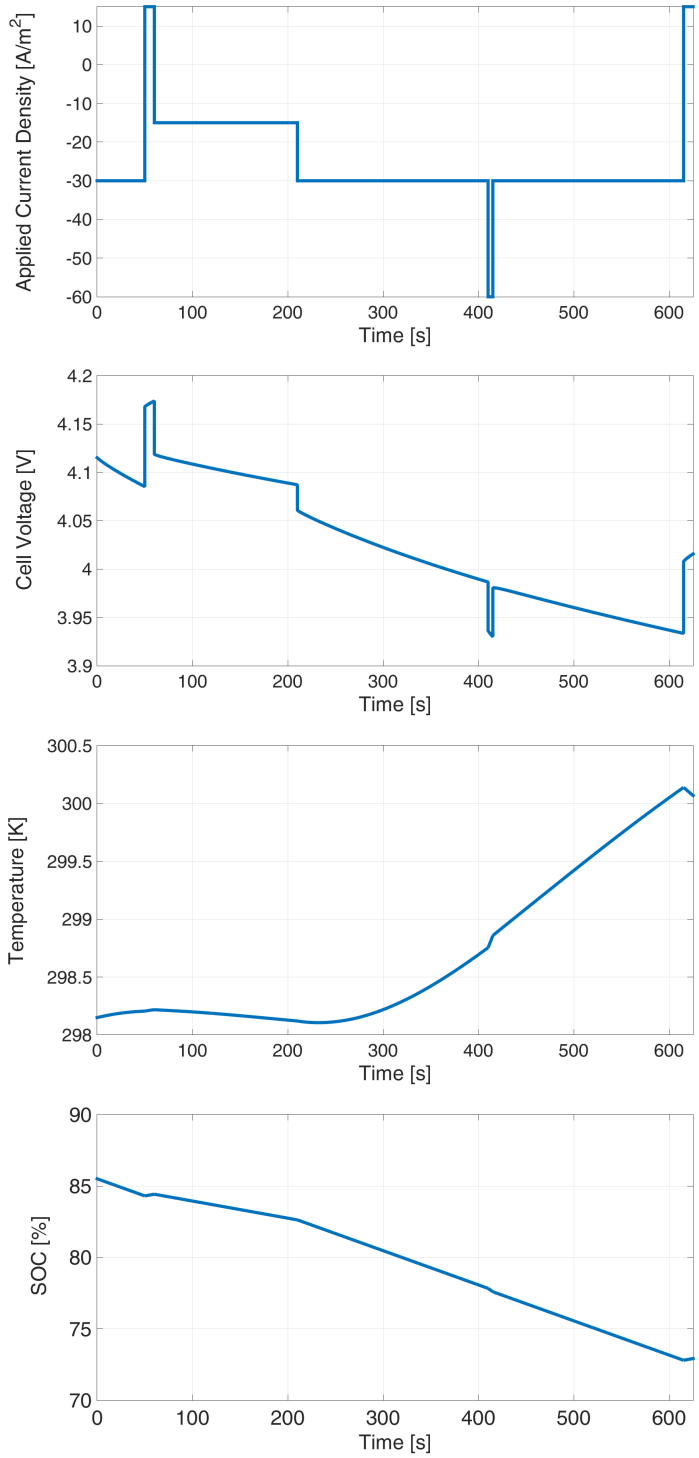

Figure 4: Hybrid charging-discharging cycle.

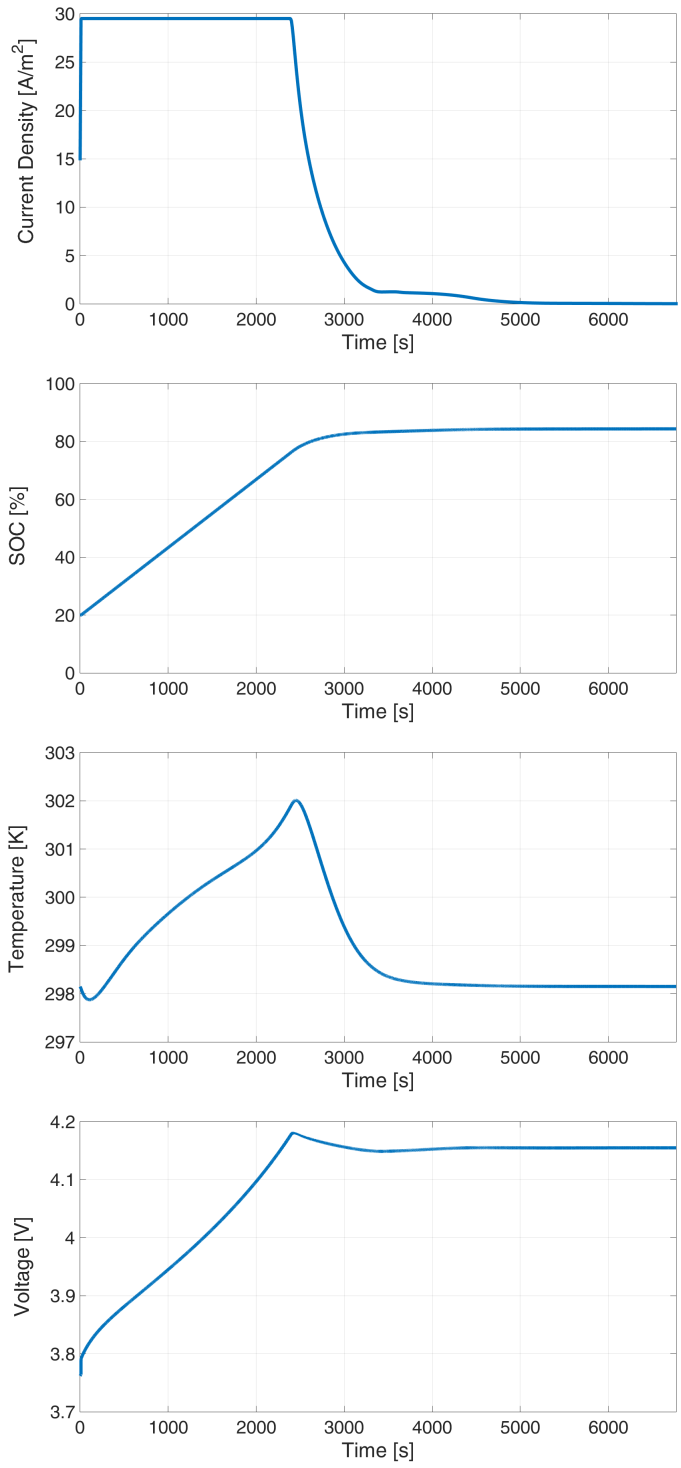

Figure 5: *ABMS control*: an MPC algorithm [9] is used to drive the charge of the battery from 20% to 85% while considering voltage, temperature, and current constraints.

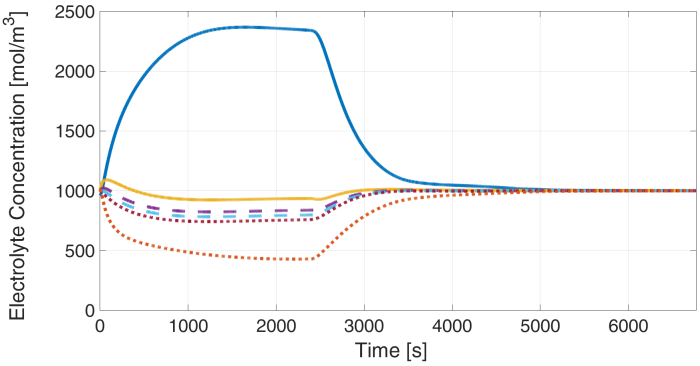

Figure 6: *ABMS control – Electrolyte concentration:* The behavior of the first and last volume of each section of the battery is depicted, where the countinous lines belong to the cathode, the dashed lines to the separator and the dotted lines to the anode.

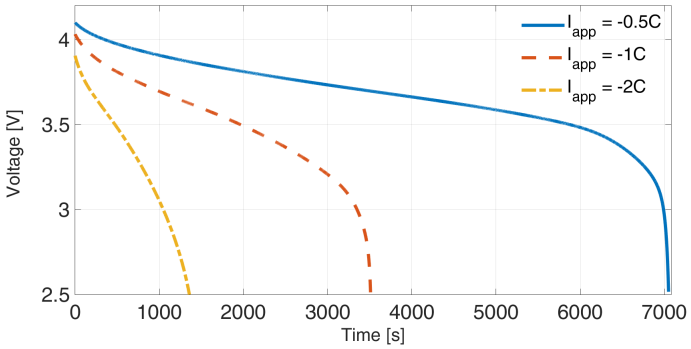

Figure 7: Full discharge cycle in an isothermal environment: blue line -0.5C, dashed orange line -1C, and dot-dashed yellow line -2C.

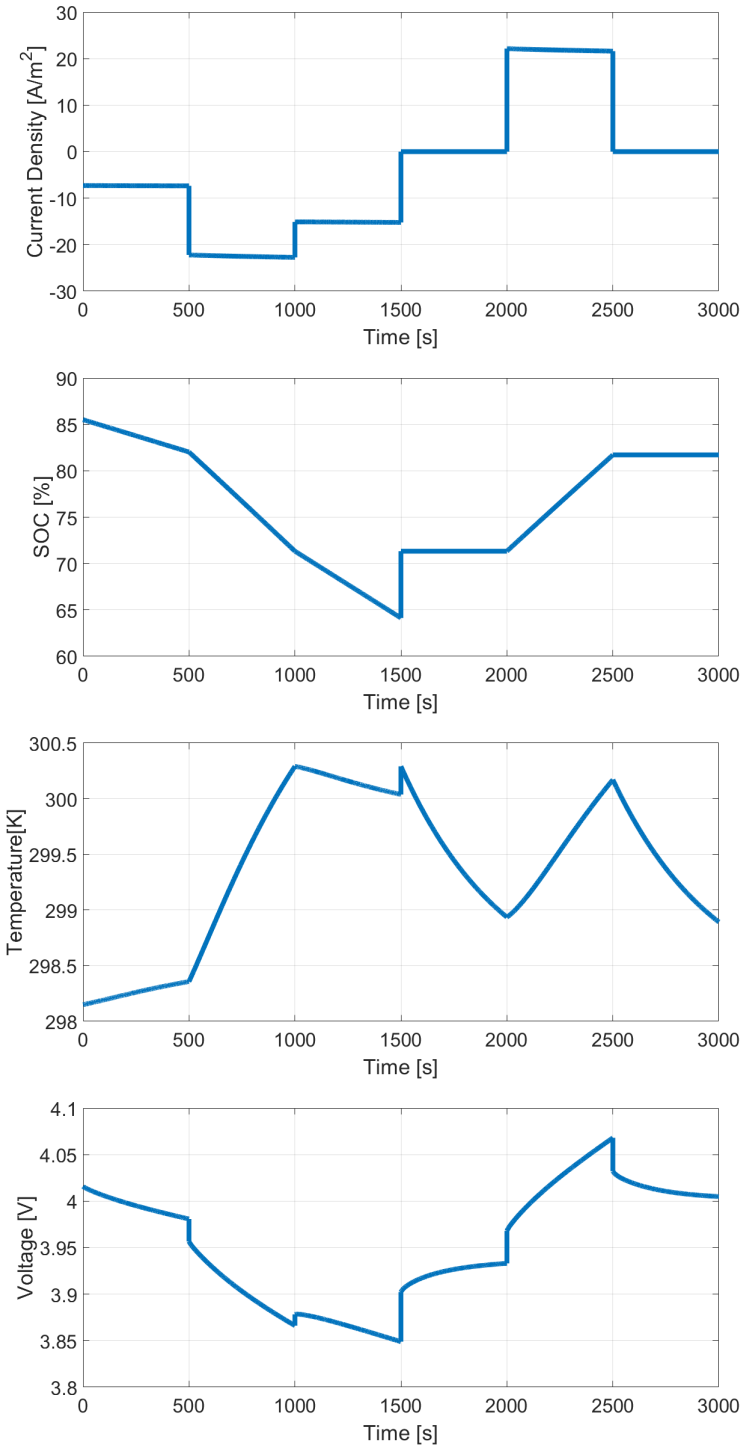

Figure 8: Charging-discharging cycle done with a constant input power density

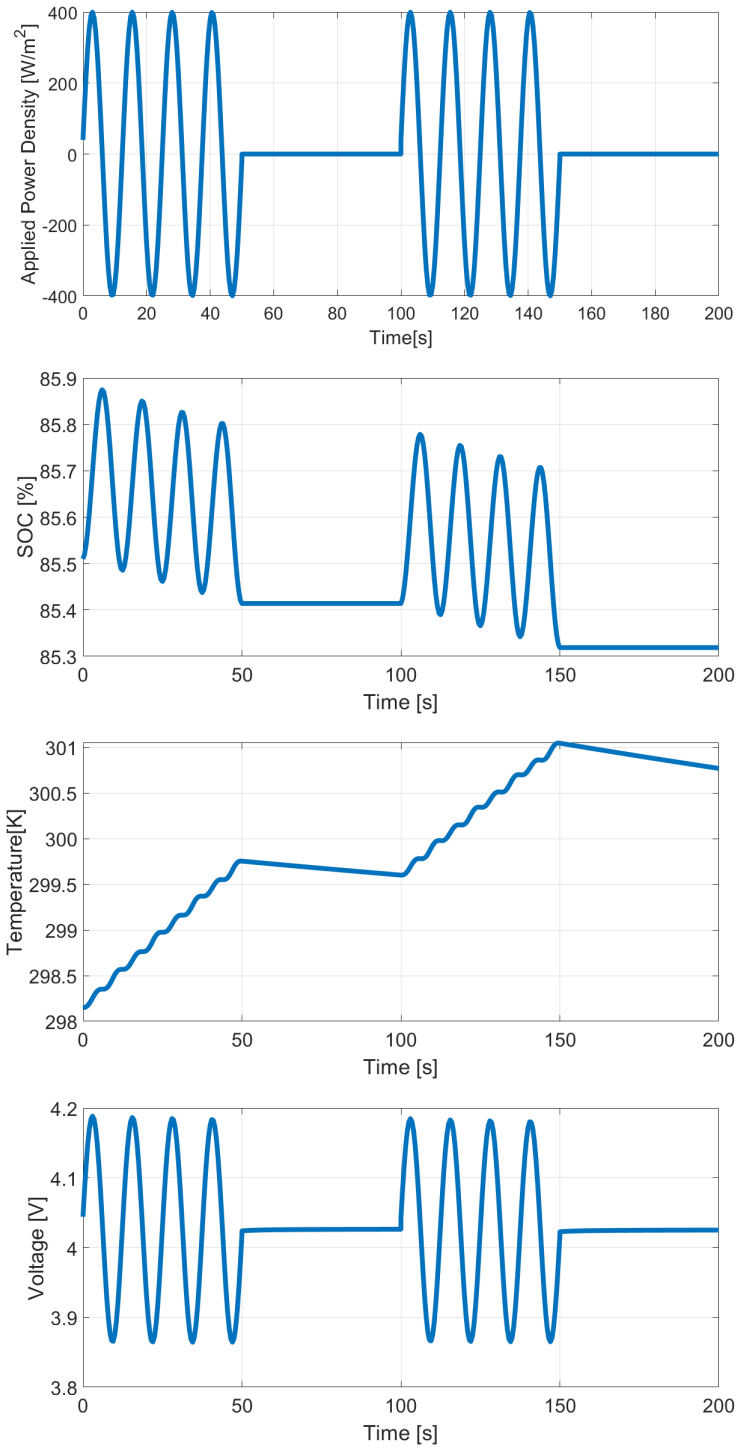

Figure 9: Sinusoidal charging-discharging cycle done with a variable input power density

References

[1] M. Torchio, L. Magni, R. B. Gopaluni, R. D. Braatz, and D. M. Raimondo, "LIONSIMBA: A matlab framework based on a finite volume model suitable for li-ion battery design, simulation, and control," *Journal of The Electrochemical Society*, vol. 163, no. 7, pp. A1192–A1205, 2016. [Online]. Available: http://jes.ecsdl.org/content/163/7/A1192.abstract

[2] M. Doyle, T. F. Fuller, and J. Newman, "Modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell," *Journal of the Electrochemical Society*, vol. 140, no. 6, pp. 1526–1533, 1993.

[3] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 363–396, Sep. 2005.

[4] P. W. C. Northrop, V. Ramadesigan, S. De, and V. R. Subramanian, "Coordinate transformation, orthogonal collocation, model reformulation and simulation of electrochemical-thermal behavior of lithium-ion battery stacks," *Journal of The Electrochemical Society*, vol. 158, no. 12, pp. A1461–A1477, 2011.

[5] K. Kumaresan, G. Sikha, and R. E. White, "Thermal model for a Li-ion cell," *Journal of the Electrochemical Society*, vol. 155, no. 2, pp. A164–A171, 2008.

[6] W. E. Schiesser, *The Numerical Method of Lines*. Academic Press, 1991.

[7] V. Ramadesigan, V. Boovaragavan, J. C. Pirkle, and V. R. Subramanian, "Efficient reformulation of solid-phase diffusion in physics-based lithium-ion battery models," *Journal of The Electrochemical Society*, vol. 157, no. 7, pp. A854–A860, 2010.

[8] J. Andersson, "A General-Purpose Software Framework for Dynamic Optimization," PhD thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, October 2013.

[9] M. Torchio, N. A. Wolff, D. M. Raimondo, L. Magni, U. Krewer, R. B. Gopaluni, J. A. Paulson, and R. D. Braatz, "Real-time model predictive control for the optimal charging of a lithium-ion battery," in *Proceedings of thet American Control Conference*, 2015, pp. 4536–4541.