# Single Particle Model - Introduction and Implementation in Python

Moin Ahmed

August 17, 2023
Version 1.1

**Abstract**

Lithium-ion batteries (LIB) are ubiquitous as energy storage systems in transportation and consumer electronics applications. Modeling and simulations of LIB operations for optimal battery cell design and battery pack management have become a significant industry and continue to garner academic research interest. This document attempts to explain a sample implementation of lithium-ion simulations in an object-oriented programming language, specifically Python. The link for the GitHub repository containing the relevant Python package is https://github.com/m0in92/SPPy. The author intends to keep updating the document to include mathematical models, numerical schemes, and sample implementation of more comprehensive electrochemical models, thermal models, and degradation models.

## Contents

## i List of Acronyms and Abbreviations

| | |
|---|---|
| BV | Butler-Volmer Equation |
| CC | Constant Current Cycler |
| LIB | Lithium-Ion Battery |
| OCV | Open-circuit potential |
| rk4 | Fourth-order version of Runge Kutta Method |
| P2D | Psuedo-2Dimensional Model |
| PDE | Partial Differential Equations |
| ROM | Reduced Order Model |
| SOC | State-of-Charge |
| UML | Unified Modelling Diagram |

## ii List of Symbols

| | |
|---|---|
| $c_{s,j}$ | Lithium concentration in the electrode $[mol/m^3]$ |
| $D_{s,j}$ | Diffusivity of electrode j $[m^2/s]$ |
| $J_j$ | Lithium-ion flux across electrode surface j $[mol/(m^2s)]$ |
| $i_{0,j}$ | Charge transfer co-efficient |
| $I$ | Current across the battery cell $[A]$ |
| $F$ | Faraday's constant [C/mol] |
| $r$ | Radial co-ordinate $[m]$ |
| $R_{cell}$ | Battery cell electrolyte resistance $[\Omega]$ |
| $S_j$ | Electrochemically active area of electrode j $[m^2]$ |
| $t$ | Temporal co-ordinate $[s]$ |
| $U_j$ | Open-circuit potential of the electrode j $[V]$ |
| $x_j$ | State-of-charge (SOC) of electrode j |
| $\alpha_a$ | Anodic charge transfer co-efficient |
| $\alpha_c$ | Cathodic charge transfer co-efficient |
| $\eta$ | Over-potential [V] |
| $\phi_{e,j}$ | Electrolyte potential in the j electrode region [V] |

| | |
|---|---|
| $\phi_{s,j}$ | Solid electrode potential in the j electrode region [V] |

## 1 Introduction

Lithium-ion batteries (LIBs) are used as energy storage systems for various transportation and consumer products applications. LIBs offer high cycling life (roughly 80 % after 1000-2000 cycles), energy density (250  270 $Whkg^{-1}$), minimum memory effects, and low discharge current. Within an application, the battery pack (Figure 1A) contains many battery cells or battery modules electrically connected in series or parallel, and the battery cell comes in various form factors: cylindrical, pouch, or prismatic (Figure 1B). Within a battery cell, the positive electrode, separator, and negative electrodes are layered within the current collector. The lithium ions flow within the battery cell while the electrons flow through the external circuit. Present-day commercial LIB can have a positive electrode that contains one of the various lithium metal oxides (e.g., lithium manganese oxide, lithium cobalt oxide, lithium iron phosphate, etc.), while the negative electrode has graphite as its electrochemically active material. Moreover, the electrolyte of commercial LIB contains lithium salt dissolved in one or a mixture of organic solvents[**Plett2015**].

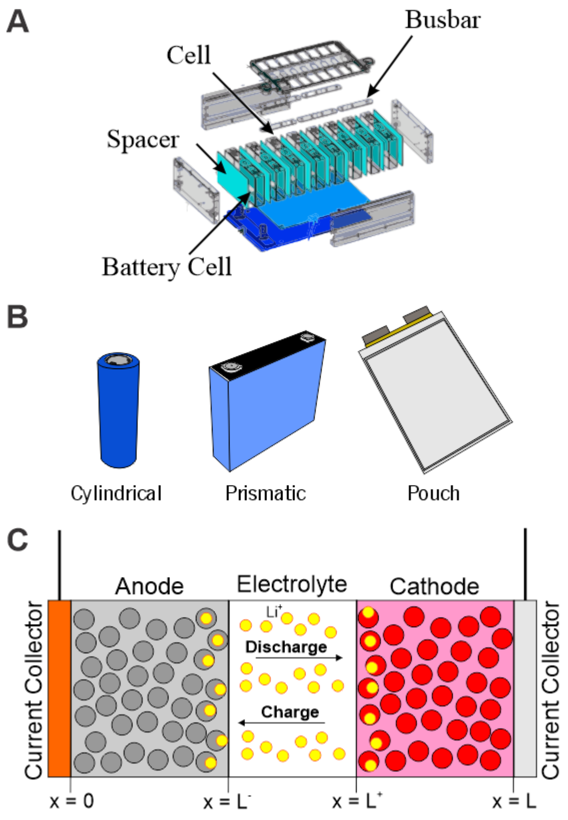

Figure 1: (A)Battery Pack, (B) Various LIB Form Factors, and (C) Flow of Lithium-ions within a Battery Cell

During charge, the lithium-ion de-intercalates from the intercalation sites within the positive electrode (oxidation), flows through the electrolyte, and intercalate into the intercalation sites with the electrochemically active negative electrode material. Meanwhile, the electrons flow from the positive terminal of the LIB to the negative terminal. The opposite flow of lithium-ions and electrodes occurs within the battery cell and the external circuit, respectively. An electrode's state-of-charge(SOC) represents the ratio of the current surface lithium concentration and its maximum lithium concentration capacity. During the battery operation, as the lithium inventory in the electrode keeps changing, the electrode's lithium stoichiometry and SOC keeps changing. Furthermore, the electrodes open circuit potential (OCP), which denotes its thermodynamic equilibrium potential, is a function of its SOC. Subsequently, the equilibrium potential of the LIB, open-circuit voltage (OCV), is a difference between the positive and negative electrodes OCP, i.e.,

$$OCV = OCP|_p - OCP|_n$$

As the flow of charges and electrons during battery operations face various sources of resistance, the LIB cell terminal voltage varies its OCV. The magnitude of this deviation of the cell terminal voltage from its equilibrium potential is referred to as its overpotential [**Newman2021**]. The overpotential during battery charge and discharge during a simulation instance is illustrated in the figures (Figures 2 and 3) below. During battery discharge, the energy lost from overpotential sources results in a lower cell terminal than its OCV (Figure 2). Similarly, energy spent overcoming these overpotential sources results in a higher cell terminal voltage than its OCV (Figure 3).

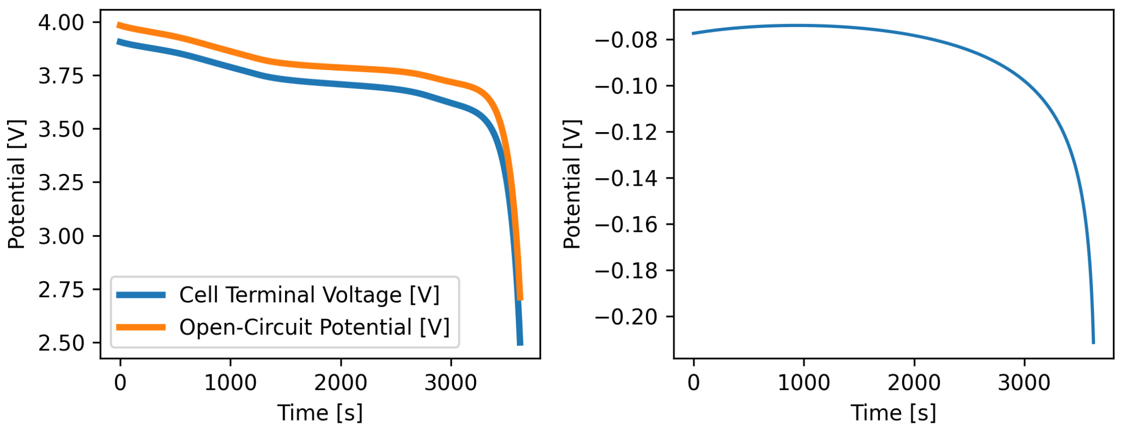

Figure 2: Battery cell terminal voltage and OCV during discharge (left) and their difference (right)

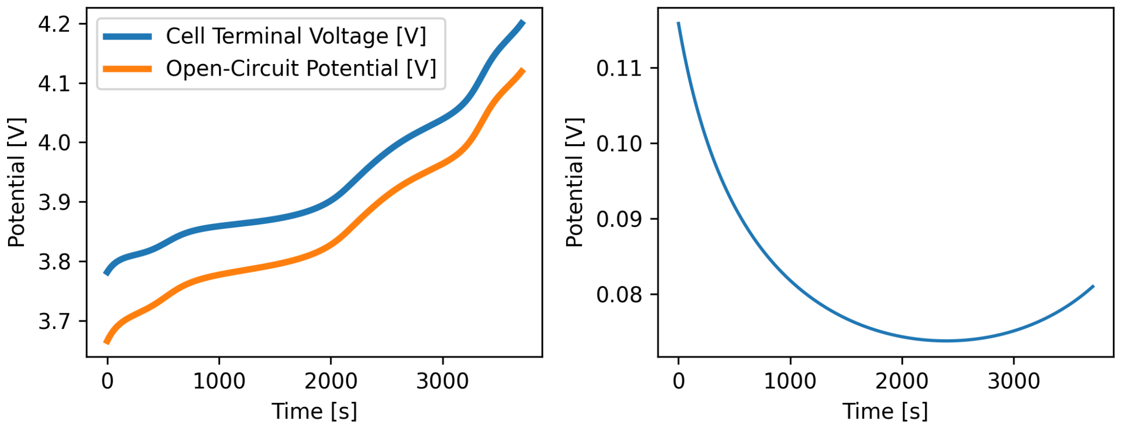

Figure 3: Battery cell terminal voltage and OCV during charge (left) and their difference (right)

Fuller, Doyle, and Neumann reported an electrochemical model for LIBs, which is still widely used today. The model uses the mass and charge conservation partial differential equations (PDEs) in both the solid (positive and negative electrodes) and solution phase (electrolyte in the positive electrode, separator, and negative electrode regions). These PDEs are coupled by the Butler-Volmer equation (BV), which models the flux of the lithium-ion across the electrode-electrolyte region. The PDEs are usually solved in 1D across the thickness of the battery cell, while the electrode particles are modeled in the spherical coordinates. For this reason, the model is also referred to as the Pseudo-2D (P2D) model [**Doyle1993**, **Fuller˙1994**]. The resulting equations can be numerically (e.g., finite differences, finite volume [**Torchio˙2016**], and finite element approaches.) solved at discrete spatial and temporal points. However due to the relative difficultly of implementation and the required computation resource required, a simpler single particle model (SPM) has also been used for battery cell simulations. SPM ignores the electrolyte dynamics during the battery operations. The result is a single PDE and an algebraic equation [**Lee˙2012**]. Electrochemical models, including SPM and P2D models, model the cell terminal voltage taking into account. Thermal and battery cell degradation models can be incorporated with these models [**Guo2011**, **O˙Kane˙2022**].
The current version of the document introduces the SPM model and its implementation in object-oriented programming in the subsequent sections.

## 2 Model Equations

### 2.1 Electrode Flux

Here, the positive and negative current from the battery is denoted for the battery charge and discharge, respectively. Furthermore, the lithium-ion fluxes in the positive and negative electrode are given below, respectively

$$J_p = \frac{I}{FS_p} \tag{2.1.1}$$

$$J_n = -\frac{I}{FS_n} \tag{2.1.2}$$

It is noteworthy that the positive flux denotes the flux exiting the electrode particle and vice-versa. The direction of the flux in the negative electrode is opposite the sign of the current and, hence, there is a negative sign in Eq 2.1.2.

## 2.2 Electrode Lithium-Ion Concentration

Assumptions

1. No spatial variation in electrode diffusivity
2. The change in lithium concentration is driven by the concentration gradient within the electrode.

The concentration in the electrodes is represented by the following partial differential equation. It uses the Fick's first law of diffusion to model the flux driven by concentration gradient.

$$\frac{\partial c_{s,j}}{\partial t} = \frac{D_{s,j}}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial c_{s,j}}{\partial r}\right) \tag{2.2.1}$$

with the boundary conditions

$$\left.\frac{\partial c_{s,j}}{\partial r}\right|_{r=0} = 0 \tag{2.2.2}$$

$$\left.\frac{\partial c_{s,j}}{\partial r}\right|_{r=R} = -\frac{J_j}{D_{s,j}} \tag{2.2.3}$$

### 2.2.1 Reduced Order Model

In some applications, such as when performing multiple simulations in tandem with limited computational resources, there might be a need to simplify PDE above to a simpler ODE expression. In this context, Simpler means an expression that is easier and quicker to solve. The volume averaging technique is one of the ways to represent the above PDE (Eq 2.2.1) to an ODE. In this technique, the average of a quantity is considered within the spatial geometry; hence, an ODE is obtained that expresses the time evolution of the average of that quantity.Applying the volume averaging technique results in the following set of ODEs and an algebraic equation[**Krishnan2018**][**Torchio˙2016**]

$$\frac{\partial c_{s,j}^{avg}}{\partial t} = -3\frac{J_j}{R_j} \tag{2.2.4}$$

$$\frac{d\bar{c}_s kr}{dt} + \frac{30D_{s,j}}{r^2}\bar{c}_s kr = -\frac{45}{2r_k^2}J_j \tag{2.2.5}$$

$$c_{j,surf} = c_{s,j}^{avg} - \frac{r_k}{35D_{s,j}} + \frac{8r_k}{35}\bar{c}_s kr \tag{2.2.6}$$

## 2.3 Cell Terminal Potential

Assumptions

1. The potential drop across the electrolyte in the cell is represented by ohmic drop, i.e.,

$$\phi_{e,p} - \phi_{e,n} = IR_{cell}$$

2. The anodic and cathodic charge transfer coefficient are equal, i.e.,

$$\alpha_c = \alpha_a = \alpha = 0.5$$

The Butler-Volmer equation (BV) models the lithium-ion flux across the electrode-electrolyte interface and is represented by

$$J_j = i_{0,j}\left\{\exp\left(\frac{\alpha F}{RT}\eta_j\right) - \exp\left(\frac{(1-\alpha)F}{RT}\eta_j\right)\right\} \tag{2.3.1}$$

Where, the over-potential $\eta_j$ is given by

$$\eta_j = \phi_{s,j} - \phi_{e,j} - U_j \tag{2.3.2}$$

After applying the values of $\alpha$ from Assumption 2 above, Eqn 2.3.1 becomes

$$J_j = i_{0,j}\left\{\exp\left(\frac{F}{2RT}\eta_j\right) - \exp\left(\frac{F}{2RT}\eta_j\right)\right\} \tag{2.3.3}$$

Applying hyperbolic identity $\left(\sinh x = \dfrac{\exp x - \exp -x}{2}\right)$, Eq 2.3.3 becomes

$$J_j = 2i_{0,j}\sinh\left(\frac{F}{RT}\eta_j\right)$$

Solving for the over-potential gives the following

$$\eta_j = \frac{2RT}{F}\sinh^{-1}\left(\frac{J_j}{2i_{i_{0,j}}}\right)$$

Inserting the expression for the over-potential in the equation above gives

$$\phi_{s,j} - \phi_{e,j} - U_j = \frac{2RT}{F} \sinh^{-1}\left(\frac{J_j}{2i_{0,j}}\right)$$

The above expression at the negative electrode end ($L = 0$) and positive electrode end ($L = L^{tot}$)are

$$\phi_{s,p} - \phi_{e,p} - U_p = \frac{2RT}{F} \sinh^{-1}\left(\frac{J_p}{2i_{0,p}}\right) \tag{2.3.4}$$

$$\phi_{s,n} - \phi_{e,n} - U_n = \frac{2RT}{F} \sinh^{-1}\left(\frac{J_n}{2i_{0,n}}\right) \tag{2.3.5}$$

Subtracting Eq 2.3.4 from Eq 2.3.5 gives

$$V_{cell} = U_p - U_n - \sinh^{-1}\left(\frac{m_p}{2}\right) + \sinh^{-1}\left(\frac{m_n}{2}\right) + IR_{cell} \tag{2.3.6}$$

Where

$$m_p = \frac{I}{Fi_{0,p}} \tag{2.3.7}$$

$$m_n = -\frac{I}{Fi_{0,n}} \tag{2.3.8}$$

## 3 Numerical Schemes for the Solid Electrode Diffusion

This section focuses on the numerical solution schemes for solving the partial differential equation (PDE) pertaining to the lithium concentration within the electrodes (Eq 2.2.1). An analytical solution for the diffusion equation does not exist once the dynamic nature of the diffusivity and the rate constant is considered[**Guo2011**]. This PDE with the associated boundary conditions is illustrated again for convenience.

$$\frac{\partial c_{s,j}}{\partial t} = \frac{D_{s,j}}{r^2}\frac{\partial}{\partial r}\left(r^2 \frac{\partial c_{s,j}}{\partial r}\right)$$

$$\left.\frac{\partial c_{s,j}}{\partial r}\right|_{r=0} = 0$$

$$\left.\frac{\partial c_{s,j}}{\partial r}\right|_{r=R} = -\frac{J_j}{D_{s,j}}$$

Furthermore, the above PDE can be scaled with respect to concentration and radial co-ordinates, as shown below.

$$\frac{\partial x_{s,j}}{\partial t} = \frac{D_{s,j}}{R\bar{r}^2}\frac{\partial}{\partial \bar{r}}\left(\bar{r}^2 \frac{\partial x_{s,j}}{\partial \bar{r}}\right) \tag{3.0.1}$$

$$\left.\frac{\partial x_{s,j}}{\partial r}\right|_{r=0} = 0 \tag{3.0.2}$$

$$\left.\frac{\partial x_{s,j}}{\partial r}\right|_{r=R} = -\frac{J_j R}{D_{s,j} c_{s,j,max}} = \delta_j \tag{3.0.3}$$

Where $x_{s,j}$ represents the state-of-charge (SOC) of the electrode.

## 3.1 Crank-Nicolson Method

Crank-Nicolson scheme is implicit and has a second-order accuracy in time. Here, backward time difference is applied to the time derivative of the diffusion PDE, while the average of the centered-finite difference is applied to the spatial derivative. Before proceeding, it is convenient to expand the derivative in the PDE as shown below.

$$\frac{\partial c_{s,j}}{\partial t} = \frac{D_{s,j}}{r^2}\frac{\partial}{\partial r}\left(r^2 \frac{\partial c_{s,j}}{\partial r}\right) = D_{s,j}\frac{\partial^2 c_{s,j}}{\partial r^2} + 2\frac{D_{s,j}}{r}\frac{\partial c_{s,j}}{\partial r}$$

The spatial and temporal points are denoted by $i$ and $k$, respectively, where $i = \{1, 2 \cdots, N\}$ and $k = \{1, 2, \cdots, K\}$. After applying the discretization as discussed above, the following is obtained

$$\frac{c_i^{k+1} - c_i^k}{\Delta t} = \frac{1}{2}\left(D_{s,j}\frac{c_{i+1}^{k+1} - 2c_i^{k+1} + c_{i-1}^{k+1}}{\Delta r^2} + \frac{D_{s,j}}{r_i}\frac{c_{i+1}^{k+1} - c_{i-1}^{k+1}}{\Delta r} + D_{s,j}\frac{c_{i+1}^{k} - 2c_i^{k} + c_{i-1}^{k}}{\Delta r^2} + \frac{D_{s,j}}{r_i}\frac{c_{i+1}^{k} - c_{i-1}^{k}}{\Delta r}\right) \tag{3.1.1}$$

The following constants are introduced

$$A = \frac{D_{s,j}\Delta t}{\Delta r^2} \tag{3.1.2}$$

$$B = \frac{D_{s,j}\Delta t}{\Delta r} \tag{3.1.3}$$

The following is obtained after substituting the above constants into Eq 3.1.1 and rearranging

$$[1 + A]\, c_i^{k+1} - \left[\frac{A}{2} + \frac{B}{r_i}\right] c_{i+1}^{k+1} - \left[\frac{A}{2} - \frac{B}{r_i}\right] c_{i-1}^{k+1} = [1 + A]\, c_i^{k} + \left[\frac{A}{2} + \frac{B}{r_i}\right] c_{i+1}^{k} + \left[\frac{A}{2} - \frac{B}{r_i}\right] c_{i-1}^{k} \tag{3.1.4}$$

The discretization has to also account for the symmetric (at $r = 0$) and flux (at $r = R_j$) boundary conditions. Also, the singularity occurs for the diffusion equation at $r = 0$, which for the symmetric boundary condition can be remedied by applying L'Hpital's rule [**Thibault˙1987**]

$$\lim_{r\to 0}\frac{D_{s,j}}{r^2}\frac{\partial r}{\partial}\left(r^2\frac{\partial c_{s,j}}{\partial r}\right)=\lim_{r\to 0}D_{s,j}\frac{\partial^2 c_{s,j}}{\partial r^2}+2\frac{D_{s,j}}{r}\frac{\partial c_{s,j}}{\partial r}=D_{s,j}\frac{\partial^2 c_{s,j}}{\partial r^2}+2D_{s,j}\frac{\partial^2 c_{s,j}}{\partial r^2}=3D_{s,j}\frac{\partial^2 c_{s,j}}{\partial r^2}$$

After applying the discretization to the boundary conditions, the following is obtained

$$(1+3A)\,c_1^{k+1}-3Ac_2^{k+1}=(1-3A)\,c_1^k+3Ac_2^k \tag{3.1.5}$$

$$(1+A)\,c_N^{k+1}-Ac_{N-1}^{k+1}=(1-A)\,c_N^k-\left(A+\frac{B}{r}\right)\left(\frac{2\Delta rJ_j}{D_{s,j}}\right)+Ac_{N-1}^k \tag{3.1.6}$$

Eqs 3.1.4 , 3.1.5, and 3.1.6 can be represented in a matrix form as

$$\begin{bmatrix}1+3A & -3A & \dots & & 0\\ & \ddots & & & \\ & -\frac{A}{2}+\frac{B}{r_i} & 1+A & \frac{A}{2}+\frac{B}{r_i} & \\ & & & \ddots & \\ 0 & & \dots & -A & (1+A)\end{bmatrix}\begin{bmatrix}c_1^{k+1}\\ \vdots\\ c_N^{k+1}\end{bmatrix}=\begin{bmatrix}1+3A & -3A & \dots & & 0\\ & \ddots & & & \\ & \frac{A}{2}-\frac{B}{r_i} & 1+A & \frac{A}{2}+\frac{B}{r_i} & \\ & & & \ddots & \\ 0 & & \dots & -A & 1+A\end{bmatrix}\begin{bmatrix}c_1^{k}\\ \vdots\\ c_N^{k}\end{bmatrix}-\begin{bmatrix}0\\ \vdots\\ \left(A+\frac{B}{R}\right)\left(\frac{2\Delta rJ_j}{D_{s,j}}\right)\end{bmatrix} \tag{3.1.7}$$

This tridiagonal matrix can be solved using the Thomas algorithm (refer to the Appendix B). Thomas algorithm needs the matrix to be diagonally dominant [**Thomas˙TDMA**], i.e.,

$$|b_i|>|a_i|+|c_i|$$

Where $a_i$, $b_i$, and $c_i$ are the lower, main, and upper diagonal elements, on a row $i$, respectively. It can easily be shown that the above condition is met in Eq 3.1.7.

## 3.2 Eigen Expansion Method

Guo *et al.* utilized eigen expansion method to numerically solve for the lithium concentration inside of the electrode particle [**Guo2011**]. The solution technique and the relevant equations of this method are presented in this section. Using the scaled PDE (Eq 3.0.1), the solution of the PDE can be represented as a summation of the temporal varying, spatial and time-varying, and an additional term. This additional term takes into account the non-homogeneous boundary conditions (Eqs 3.0.2 and 3.0.3).

$$x_j=\frac{\delta}{2}\overline{r}_j^2+C(t)+w(\overline{r}_j,t)$$

Where the last term was expanded using the Fourier series.

$$w(\overline{r}_j, t) = \sum_{n=0}^{\infty} a_k(t) z_k(\overline{r}_j)$$

Furthermore, Guo *et al.* solved for the functions in the above equations. When the above series is expanded to finite $N$ terms, the resulting equation for the electrode's surface SOC can be expressed as

$$x_{j,surf} = x_{ini,j} + \frac{1}{5}\delta_j + 3\int_0^t \frac{D_{s,j}(\tau)}{R_j^2}\delta dt + \sum_{n=1}^{N}\left(u_{j,k} - \frac{2\delta_j}{\lambda_k^2}\right) \tag{3.2.1}$$

Where the eigen function, $u_{j,k}$, can be solved through the ordinary differential equation (ODE)

$$\frac{du_{k,j}(t)}{dt} = -\frac{\lambda_k^2 D_{s,j}(t)}{R_j^2} u_{j,k} + 2\frac{D_{s,j}(t)}{R_j^2}\delta_j$$

and the eigen value, $\lambda_k$, is the root of the following equation.

$$\sin\lambda_k - \lambda_k \cos\lambda_k = 0$$

## 4 Implementation in Python

### 4.1 General Code Organization

The sample Python package for the single particle model (SPM), delineated in this section, is written to ensure it is user-friendly, modular, and flexible. User-friendly enables the user to write complex lithium-ion battery (LIB) complex simulation scripts with the fewest lines of code, and the script structure is logically intuitive. Modular code will allow for the expandability of the more electrochemical, thermal, and degradation models (as planned to be discussed in future versions of this document). Unit tests are performed to test the functionalities of various functions and classes. However, the unit tests are not discussed in this document, and the scripts for unit tests can be viewed under the test directories. A flexible package structure will enable the user to quickly understand and change the various battery cell and simulation parameters during usage.

Generally, various battery parameters must be stored, accessed, and modified during a simulation. Moreover, model equations, along with the numerical solution schemes, need to be stored and utilized. Furthermore, the battery simulation will be run under some battery cycling conditions. For example, the battery cell simulation will undergo a discharge, charge, or a combination of both. Also, the storage of simulated parameters, during and after the simulation, and visualization of simulation results, after the simulation, will be needed. Class objects are declared for the battery cell components, SPM model equations, numerical methods, cycling conditions, simulation storage, and simulation visualization. A conceptual perspective of the class objects in unified modeling

language is depicted below (Figure 4).
A more detailed commentary on the code for the various class objects are discussed in the subsequent sections. Note that the comments on classes about cyclers and visualization are not included in this version of the documentation.

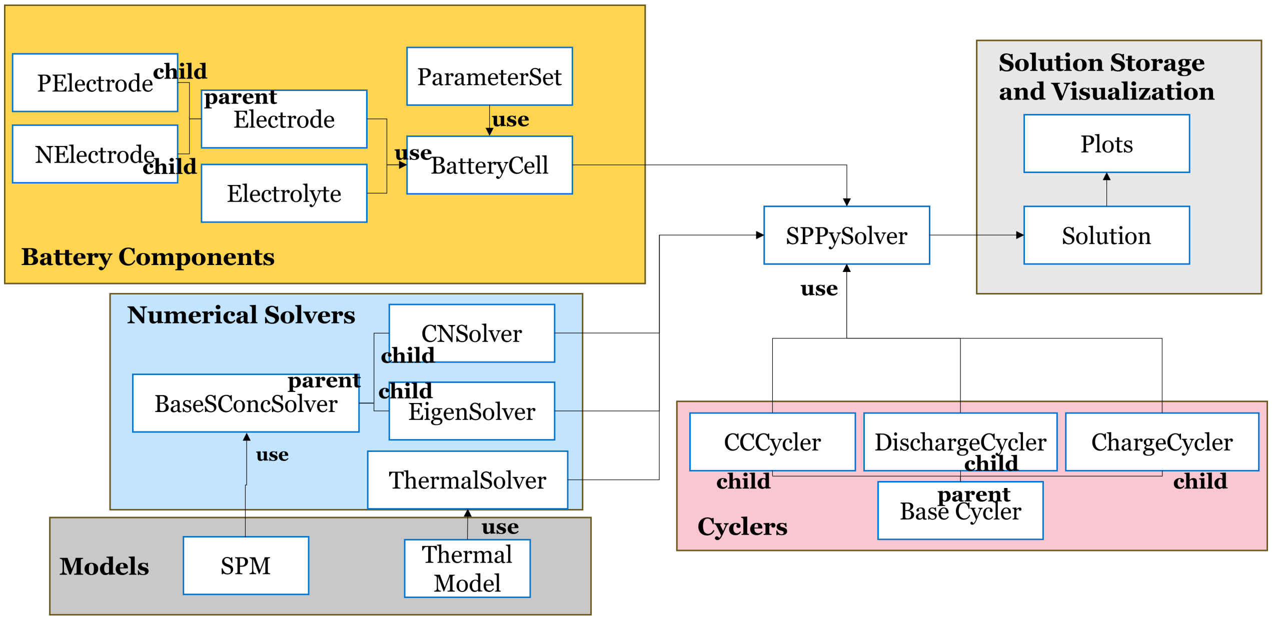

Figure 4: Conceptual Perspective of the Class Diagram in UML

## 4.2 Battery Component Parameters Extraction and Storage

The battery electrode and electrolyte parameters are stored in class objects `Electrode` and `Electrolyte` , respectively. At the time of this document writing, the parameters are stored and extracted from the .csv file. However, the author is open to other options, such as .json or .xml files, in the future. The code snippet below shows the initialization of the Electrode class. Note that Pythons `@dataclass` decorator is used before defining the class name, and this decorator will automatically generate the classs `__init__` and `__repr__` methods[**Python˙dataclass**]. This class is intended to store parameters such as the electrodes thickness, cross-sectional area, diffusivity, rate constant, etc.

```
from typing import Optional
from dataclasses import dataclass, field
import collections


@dataclass
class Electrode_:
    """
    This class is used to create an electrode object. This object is used to store
         the electrode parameters and provide
    relevant electrode methods. It takes input parameters from the csv file. The
        electrode parameters are stored as
    attributes of the object.
    """
    L: float # Electrode Thickness [m]
    A: float # Electrode Area [m^2]
    kappa: float # Ionic Conductivity [S m^-1]
    epsilon: float # Volume Fraction
    max_conc: float # Max. Conc. [mol m^-3]
    R: float # Radius [m]
    S: Optional[float] # Electro-active Area [m2]
    T_ref: float # Reference Temperature [K]
    D_ref: float # Reference Diffusivity [m2/s]
    k_ref: float # Reference Rate Constant [m2.5 / (mol0.5 s)
    Ea_D: float # Activation Energy of Diffusion [J / mol]
    Ea_R: float # Activation Energy of Reaction [J / mol]
    alpha: float # Anodic Transfer Coefficient
    brugg: float # Bruggerman Coefficient
    SOC_init: float # initial SOC
    func_OCP: collections.abc.Callable[[float], [float]] # electrode open-circuit
        potential function that takes SOC as
    # its arguments
    func_dOCPdT: collections.abc.Callable[[float], [float]] # the function that
        represents the change of open-curcuit
    # potential with SOC
    T: float # electrode temperature, K
    electrode_type: str = field(default='none')  # electrode type: none, 'p', or '
        n'
```

Similarly, the following code snippet shows the `Electrolyte` class, which stores the electrolytes initial concentration, conductivity, volume fraction, Bruggermans coefficient, along with the separator thickness.

```
from dataclasses import dataclass


@dataclass
class Electrolyte_:
    """
    Class for the Electrolyte object and contains the relevant electrolyte
        parameters.
    """
    L: float  # seperator thickness, m3
    conc: float  # electrolyte concentration, mol/m3
    kappa: float  # ionic conductivity, S/m
    epsilon: float  # electrolyte volume fraction
    brugg: float  # Bruggerman coefficient for electrolyte

    def __post_init__(self):
        # Check for types of the input parameters
        if not isinstance(self.conc, float):
            raise "Electrolyte␣conc.␣needs␣to␣be␣a␣float."
        if not isinstance(self.L, float):
            raise "Electrolyte␣thickness␣needs␣to␣be␣a␣float."
        if not isinstance(self.kappa, float):
            raise "Electrolyte␣conductivity␣needs␣to␣be␣a␣float."
        if not isinstance(self.epsilon, float):
            raise "Electrolyte␣volume␣fraction␣needs␣to␣be␣a␣float."
        if not isinstance(self.brugg, float):
            raise "Electrolyte's␣bruggerman␣coefficient␣needs␣to␣be␣a␣float."

    @ property
    def kappa_eff(self):
        return self.kappa * (self.epsilon ** self.brugg)
```

## 4.3 Single Particle Model

Python class `SPM` contains the methods for calculating electrode molar flux and cell terminal voltage, as shown below. While separate methods for calculating $m_j$ (Eqs 2.3.7 and 2.3.8 ) and cell terminal voltages (Eq 2.3.6) are declared, the declared `__call__` method calculates the $m$ and (and returns) the cell terminal voltage. Hence, when the `SPM` class instance is called, it can return the cell terminal voltage without calculating the $m$ for each electrode.

```
class SPM:
    """
    This class contains the methods for calculating the molar lithium flux, cell
        terminal voltage according to the
    single particle model.
    """
    @classmethod
    def molar_flux_electrode(cls, I: float, S: float, electrode_type: str) ->
        float:
        """
        Calculates the model lithium-ion flux [mol/m2/s] into the electrodes.
        :param I: (float) Applied current [A]
        :param S: (float) electrode electrochemically active area [m2]
        :param electrode_type: (str) positive electrode ('p') or negative
            electrode ('n')
        :return: (float) molar flux [mol/m2/s]
        """
        if electrode_type == 'p':
            return I / (Constants.F * S)
        elif electrode_type == 'n':
            return -I / (Constants.F * S)
        else:
            raise InvalidElectrodeType
    @staticmethod
    def flux_to_current(molar_flux: float, S: float, electrode_type: str) -> float
        :
        """
        Converts molar flux [mol/m2/s] to current [A].
        :param molar_flux: molar lithium-ion flux [mol/m2/s]
        :param S: (float) electrode electrochemically active area [m2]
        :param electrode_type: (str) positive electrode ('p') or negative
            electrode ('n')
        :return: (float) current [A]
        """
        if electrode_type == 'p':
            return molar_flux * Constants.F * S
        elif electrode_type == 'n':
            return -molar_flux * Constants.F * S
        else:
            raise InvalidElectrodeType

    @staticmethod
    def m(I, k, S, c_max, SOC, c_e) -> float:
        return I / (Constants.F * k * S * c_max * (c_e ** 0.5) * ((1 - SOC) **
            0.5) * (SOC ** 0.5))

    @staticmethod
    def calc_cell_terminal_voltage(OCP_p, OCP_n, m_p, m_n, R_cell, T, I) -> float:
        V = OCP_p - OCP_n
        V += (2 * Constants.R * T / Constants.F) * np.log((np.sqrt(m_p ** 2 + 4) +
             m_p) / 2)
        V += (2 * Constants.R * T / Constants.F) * np.log((np.sqrt(m_n ** 2 + 4) +
             m_n) / 2)
        V += I * R_cell
        return V

    def __call__(self, OCP_p, OCP_n, R_cell,
                 k_p, S_p, c_smax_p, SOC_p,
                 k_n, S_n, c_smax_n, SOC_n,
                 c_e,
                 T, I_p_i, I_n_i) -> float:
        """
        Calculates the cell terminal voltage.
        :param OCP_p: Open-circuit potential of the positive electrode [V]
        :param OCP_n: Open-circuit potential of the negative electrode [V]
        :param R_cell: Battery cell ohmic resistance [ohms]
        :param k_p: positive electrode rate constant [m2 mol0.5 / s]
        :param S_p:  positive electrode electro-active area [mol/m2]
        :param c_smax_p: positive electrode max. lithium conc. [mol]
        :param SOC_p: positive electrode SOC
        :param k_n: negative electrode rate constant [m2 mol0.5 / s]
        :param S_n: negative electrode electrochemical active area [m2/mol]
        :param c_smax_n: negative electrode max. lithium conc. [mol]
        :param SOC_n: negative electrode SOC
        :param c_e: electrolyte conc. [mol]
        :param T: Battery cell temperature [K]
        :param I_p_i: positiive electrode intercalation applied current [A]
        :param I_n_i: negative electrode intercalation applied current [A]
        :return: Battery cell terminal voltage [V]
        """
        m_p = self.m(I=I_p_i, k=k_p, S=S_p, c_max=c_smax_p, SOC=SOC_p, c_e=c_e)
        m_n = self.m(I=I_n_i, k=k_n, S=S_n, c_max=c_smax_n, SOC=SOC_n, c_e=c_e)
        return self.calc_cell_terminal_voltage(OCP_p=OCP_p, OCP_n=OCP_n, m_p=m_p,
            m_n=m_n, R_cell=R_cell, T=T, I=I_p_i)
```

## 4.4 Numerical Method Solvers for Solid Electrode Diffusion

Various numerical schemes for solid electrode diffusion have been discussed above, and Python solvers for these schemes are discussed below. Each of the schemes has its class. While each class has unique attributes and methods, they are all programmed to solve for the electrodes state-of-charge (SOC) for the successive time step. Most, if not all, solvers inherit from the `BaseSolver` class that primarily checks for the type of certain input parameters. Furthermore, the parameter `electrode_type` is required by all relevant solvers since the electrode flux depends on it (Egs 2.1.2 and 2.1.1).

### 4.4.1 Crank Nicolson Scheme

This section explains the various methods of the `CNSolver` class. First, the `__init__` method below initializes the class instance. The `__init__` method takes the `c_init` parameter that denotes the solid lithium concentration at the current time step. The radial coordinate is discretized in 100 spatial points by default, though it can be changed by the user.

```
class CNSolver(BaseElectrodeConcSolver):
    """
    Crank Nickelson Scheme for solving for spherical diffusion in solid electrode
        in lithium-ion batteries models. The
    associated PDE uses the mass transport with symmetry condition imposed at r=0
        and flux boundary condition at r=R.

    The PDE is:
    dx/dt = (D/(R^2 * r_scaled^2)) * d(r_scaled^2 * dx/dr_scaled)/dr_scaled

    with BC:
    dx/dr_scaled = 0 at r_scaled=0
    dx/dr_scaled = -jR/D*c_smax at r_scaled=1
    """
    def __init__(self, c_init: float, electrode_type: str, spatial_grid_points:
       int = 100):
        super().__init__(electrode_type=electrode_type)
        self.K = spatial_grid_points  # number of spatial grid points
        self.c_prev = c_init * np.ones(self.K).reshape(-1, 1)  # column vector
            used for storing concentrations at t_prev
```

Next, the constants (including constants from Eqs 3.1.2 and 3.1.3) are defined.

```
    def dr(self, R: float) -> float:
        """
        Difference in radial coordinate [m].
        :param R: electrode particle radius
        :return:
        """
        return R / self.K

    def A(self, dt: float, R: float, D: float) -> float:
        """
        Value of the constant A (delta_t * D / delta_r^2)
        :return: Returns the value of the A constant, used for the forming the
            matrices
        """
        return dt * D / (self.dr(R) ** 2)

    def B(self, dt: float, R: float, D: float) -> float:
        """
        Value of constant B (delta_t * D / (2 * delta_r))
        :return:
        """
        return dt * D / (2 * self.dr(R))

    def array_R(self, R: float) -> npt.ArrayLike:
        """
        Array containing the values of r at every grid point.
        :return: Array containing the values of r at every grid point.
        """
        return np.linspace(0, R, self.K)
```

The following methods store and return the lower, main, and upper diagonal elements in arrays (Eq 3.1.7). Moreover, a method to create an array is created in case the equation needs to be solved using matrix multiplication. The column vector in the RHS of the Eq 3.1.7 is also defined below.

```
    def _LHS_diag_elements(self, dt: float, R: float, D: float) -> npt.ArrayLike:
        A_ = self.A(dt=dt, R=R, D=D)
        array_elements = (1 + A_) * np.ones(self.K)
        array_elements[0] = 1 + 3 * A_  # for symmetry boundary condition at r=0
        array_elements[-1] = 1 + A_
        return array_elements
    def _LHS_lower_diag(self, dt: float, R: float, D: float) -> npt.ArrayLike:
        A_ = self.A(dt=dt, R=R, D=D)
        B_ = self.B(dt=dt, R=R, D=D)
        array_elements = -(A_/2 - B_/self.array_R(R)[1:]) * np.ones(self.K-1)
        array_elements[-1] = -A_  # for the flux at r=R
        return array_elements

    def _LHS_upper_diag(self, dt: float, R: float, D: float) -> npt.ArrayLike:
        A_ = self.A(dt=dt, R=R, D=D)
        B_ = self.B(dt=dt, R=R, D=D)
        array_elements = -(A_ / 2 + B_ / self.array_R(R)[1:-1]) * np.ones(self.K -
             2)
        array_elements = np.insert(array_elements, 0, -3 * A_)  # for symmetry
            boundary condition at r=0
        return array_elements

    def M(self, dt: float, R: float, D: float) -> npt.ArrayLike:
        return np.diag(self._LHS_diag_elements(dt=dt, R=R, D=D)) + \
               np.diag(self._LHS_lower_diag(dt=dt, R=R, D=D), -1) + \
               np.diag(self._LHS_upper_diag(dt=dt, R=R, D=D), 1)

    def _RHS_array(self, j: float, dt: float, R: float, D: float):
        A_ = self.A(dt=dt, R=R, D=D)
        B_ = self.B(dt=dt, R=R, D=D)
        array_c_temp = np.zeros(self.K).reshape(-1,1)
        array_c_temp[0][0] = (1-3*A_)*self.c_prev[0][0] + 3*A_*self.c_prev[1][0]
            # for the symmetry boundary condition
        # at r=0
        array_c_temp[-1][0] = (1-A_) * self.c_prev[-1][0] - (A_+B_/R) * (2*self.dr
            (R=R)*j/D) + \
                              A_ * self.c_prev[-2][0]  # for the boundary
                                condition at r=R
        for i in range(1, len(array_c_temp) - 1):
            array_c_temp[i][0] = (1 - A_) * self.c_prev[i][0] + \
                                 (A_ / 2 + B_ / self.array_R(R=R)[i]) * self.
                                     c_prev[i + 1][0] + \
                                 (A_ / 2 - B_ / self.array_R(R=R)[i]) * self.
                                     c_prev[i - 1][0]
        return array_c_temp
```

Finally, the `solve` method solves the lithium concentration for the current time step using the matrix multiplication or Thomas algorithm method (refer to Appendix B).

```
    def solve(self, dt: float, i_app: float, R: float, S: float, D: float,
       solver_method:str):
        """
        Solves for the lithium-ion concentration after dt. It then updates the
            class instance's c_prev attribute.
        :param c_prev: (numpy array) matrix (Kx1) containing the concentrations at
```

```
            t_prev [mol/m3]
        :param j: (float) lithium flux at r=R [mol/m2/s]
        :param dt: (float) time difference [s]
        :param R: (float) electrode particle radius [m]
        :param D: (float) electrode diffusivity [m2/s]
        :return:
        """
        j = SPM.molar_flux_electrode(I=i_app, S=S, electrode_type=self.
            electrode_type)
        if solver_method == "inverse":
            self.c_prev = np.linalg.inv(self.M(dt=dt, R=R, D=D)) @ self._RHS_array
                (j=j, dt=dt, R=R, D=D)
        elif solver_method == "TDMA":
            self.c_prev = ode_solvers.TDMAsolver(l_diag=self._LHS_lower_diag(dt=dt
                , R=R, D=D),
                                                  diag=self._LHS_diag_elements(dt=
                                                      dt, R=R, D=D),
                                                  u_diag=self._LHS_upper_diag(dt=dt
                                                      , R=R, D=D),
                                                  col_vec=self._RHS_array(j=j, dt=
                                                      dt, R=R, D=D)).flatten().
                                                      reshape(-1, 1)

    def __call__(self, dt: float, t_prev: float, i_app:float, R: float, S:float,
        D_s: float, c_smax: float,
                 solver_method: str = "TDMA") -> float:
        """
        Returns the electrode surface SOC
        """
        self.solve(dt=dt, i_app=i_app, R=R, S=S, D=D_s, solver_method=
            solver_method)
        return self.c_prev[-1][0] / c_smax
```

### 4.4.2 Eigen Expansion Method

The `__init__` method requires the initial electrode SOC, the number of $N$ terms in the equation (Eq 3.2.1), and the electrode type.

```
class EigenFuncExp(BaseElectrodeConcSolver):
    """
    This solver uses the Eigen Function Expansion method as detailed in ref 1 to
        calculate the electrode
    surface SOC. As such, it stores all the necessary variables required for the
        iterative calculations for all time
    steps. After initiating the class, the object instance can be called
        iteratively to solve for electrode surface SOC
    for all time steps.
    The equation solved is
        x_j_surf = x_ini + (1/5)*j_scaled + 3 * integration{D_sj * j_scaled / R_j
            ** 2} * dt +
                    summation{u_jk - 2 * scaled_j / lambda_k ** 2}
    In the above equation, the integration is performed from t=0 to the current
        time. Moreover, the summation is
    performed from k=1 to k=N. Here k represents the kth term of the solution
        series.

    Reference:
    1. Guo, M., Sikha, G., & White, R. E. (2011). Single-Particle Model for a
        Lithium-Ion Cell: Thermal Behavior.
    Journal of The Electrochemical Society, 158(2), A122. https://doi.org
        /10.1149/1.3521314/XML
    """

    def __init__(self, x_init: float, n: int, electrode_type: str):
        self.x_init_ = x_init  # initial electrode SOC
        self.N_ = n  # the number of terms in the solution series

        self.integ_term = 0  # the integration term, which is initialized as zero
        self.lst_u_k = [0 for i in range(self.N)]  # a list of solved values of
            eigenfunctions, the values of which are
        # all initialized to zero.

        super().__init__(electrode_type=electrode_type)
```

The following methods determine the eigenvalues $\lambda_k$, which is stored as the class property.

```
    @staticmethod
    def lambda_func(lambda_k) -> float:
        """
        Algebraic equation from which the eigenvalues can be calculated.
        :param lambda_k: (float) eigen value ot the kth term.
        :return: (float) the value of the elgebraic equation.
        """
        return np.sin(lambda_k) - lambda_k * np.cos(lambda_k)

    def lambda_bounds(self) -> npt.ArrayLike:
        """
        The list which contains the tuple containing the lower and upper bounds of
             the eigenvalues.
        :return: (list) list containing the tuple of bounds for the eigenvalues.
        """
        return [(np.pi * (1 + k), np.pi * (2 + k)) for k in range(self.N)]  # k
            refers to the kth term of the series

    @property
    def lambda_roots(self) -> npt.ArrayLike:
        """
        Uses the bisect method to solve for the eigenvalue algebraic equation
            within the bounds.
        :return: (list) list containing the eigenvalues for all solution terms.
        """
        bounds = self.lambda_bounds()
        return [bisect(self.lambda_func, bounds[k][0], bounds[k][1]) for k in
            range(self.N)]  # k refers to the kth
        # term of the series
```

Moreover, the following methods are used to solve for the eigenfunction values, the summation, and the integration terms in Eq. 3.2.1.

```
    def update_integ_term(self, dt, i_app, R, S, D_s, c_smax) -> float:
        """
        Updates the integration term. Integration is performed using simple
            algebraic integration.
        :return: (float) updated integration term.
        """
        self.integ_term += 3 * (D_s * self.j_scaled(i_app=i_app, R=R, S=S, D_s=D_s
            , c_smax=c_smax) / (R ** 2)) * dt

    @staticmethod
    def u_k_expression(lambda_k, D, R, scaled_flux) -> Callable:
        """
        Returns a function that represents the eigenfunction ode.
        :param lambda_k: eigenvalue of the kth term
        :param D: diffusivity [m2/s]
        :param R: electrode particle radius [m]
        :param scaled_flux: dimensionless scaled lithium-ion flux
        :return: (func) a function representing the eigenfunction ode
        """
        def u_k_odeFunc(x, t):
            return -(lambda_k ** 2) * D * x / (R ** 2) + 2 * D * scaled_flux / (R
                ** 2)
        return u_k_odeFunc

    def solve_u_k(self, root_value, t_prev, dt, u_k_prev, i_app, R, S, D_s, c_smax
       ) -> float:
        """
        Calculates the eigenfunction value from the eigen values using the ode
            function. This ode function is solved
        using the rk4 ode solver.
        :param root_value:
        :param t_prev:
        :param u_k_j_prev:
        :param dt:
        :return:
        """
        j_scaled_ = self.j_scaled(i_app=i_app, R=R, S=S, D_s=D_s, c_smax=c_smax)
        u_k_p_func = self.u_k_expression(lambda_k=root_value, D=D_s, R=R,
            scaled_flux=j_scaled_)
        return ode_solvers.rk4(func=u_k_p_func, t_prev=t_prev, y_prev=u_k_prev,
            step_size=dt)

    def get_summation_term(self, dt, t_prev, i_app, R, S, D_s, c_smax) -> float:
        """
        Calculates and returns the summation term of the Eigen Expansion equation.
        :param t_prev:
        :param dt:
        :param j_scaled:
        :return:
        """
        sum_term = 0
        # Solve for the eigenfunction for all roots using the iteration below:
        j_scaled_ = self.j_scaled(i_app=i_app, R=R, S=S, D_s=D_s, c_smax=c_smax)
        for iter_root, root_value in enumerate(self.lambda_roots):
            self.lst_u_k[iter_root] = self.solve_u_k(root_value=root_value, t_prev
                =t_prev,
                                                      u_k_prev=self.lst_u_k[
                                                          iter_root], dt=dt,
                                                      i_app=i_app, R=R, S=S, D_s=
                                                          D_s, c_smax=c_smax)
            sum_term += self.lst_u_k[iter_root] - (2 * j_scaled_ / (root_value **
                2))
        return sum_term
```

Finally, the SOC is solved using the method below, and the classs `__call__` runs this method once the class instance is called.

```
    def calc_SOC_surf(self, dt, t_prev, i_app, R, S, D_s, c_smax) -> float:
        """
        Calculates the electrode surface SOC using the Eigen Expansion method.
        :param dt: The time difference between the current and the previous time
            steps [s].
        :param t_prev: Time value of the previous time step [s].
        :return: (float) The electrode's surface SOC.
        """
        j_scaled_ = self.j_scaled(i_app=i_app, R=R, S=S, D_s=D_s, c_smax=c_smax)
        sum_term = self.get_summation_term(t_prev=t_prev, dt=dt, i_app=i_app, R=R,
             S=S, D_s=D_s, c_smax=c_smax)
        self.update_integ_term(dt=dt, i_app=i_app, R=R, S=S, D_s=D_s, c_smax=
            c_smax)
        return self.x_init + j_scaled_ / 5 + self.integ_term + sum_term

    def __call__(self, dt, t_prev, i_app, R, S, D_s, c_smax) -> float:
        """
        This method calculates the electrode surface SOC
        :param dt: Time difference between current and previous time steps [s].
        :param t_prev: Time value at the previous time step [s].
        :param i_app: Applied current at the current time step [A].
        :param R:  Electrode particle radius [m]
        :param S: Electrode electroactive area [m2]
        :param D_s: Electrode diffusivity [m2/s]
        :param c_smax: Electrode max. conc [mol/m3]
        :return: (float) The electrode's surface SOC.
        """
        return self.calc_SOC_surf(dt=dt, t_prev=t_prev, i_app=i_app, R=R, S=S, D_s
            =D_s, c_smax=c_smax)
```

### 4.4.3 Reduced Order Model - Volume Averaging Technique

The following class solves the two ODEs and one algebraic equation (Eqs 2.2.4, 2.2.5, and 2.2.6) that results from the reduced order model (ROM) from volume averaging the solid diffusion PDE.

```
class PolynomialApproximation(BaseElectrodeConcSolver):
    """
    Two=parameter polynomial approximation for the spherical diffusion using two
        parameters [1].

    Reference:
    [1] Torchio, M., Magni, L., Gopaluni, R. B., Braatz, R. D., & Raimondo, D. M.
        (2016).
    LIONSIMBA: A Matlab Framework Based on a Finite Volume Model Suitable for Li-
        Ion Battery Design, Simulation,
    and Control.
    Journal of The Electrochemical Society, 163(7), A1192 A1205.
    https://doi.org/10.1149/2.0291607JES/XML
    """
    def __init__(self, c_init: float, electrode_type: str, type: str = 'two'):
        super().__init__(electrode_type=electrode_type)
        self.c_s_avg_prev = c_init
        self.c_surf = c_init
        if type=='two' or type == 'higher':
            self.type = type
        else:
            raise ValueError(f"{type}␣is␣not␣recognized␣as␣a␣solver␣type")
        if self.type == 'higher':
            self.q = 0

    def func_c_s_avg(self, j: float, R: float) -> Callable:
        def wrapper(r, t):
            return -3 * j / R
        return wrapper

    def func_q(self, j: float, R: float, D: float) -> Callable:
        def wrapper(x, t):
            return -30 * (D/R**2) * x - (45*j/(2 * R**2))
        return wrapper

    def solve(self, dt: float, t_prev: float, i_app: float, R: float, S: float, D:
         float):
        j = SPM.molar_flux_electrode(I=i_app, S=S, electrode_type=self.
            electrode_type)
        self.c_s_avg_prev = ode_solvers.rk4(func=self.func_c_s_avg(j=j, R=R),
            t_prev=t_prev,
                                            y_prev=self.c_s_avg_prev, step_size=
                                                dt)
        if self.type != 'two':
            self.q = ode_solvers.rk4(self.func_q(j=j, R=R, D=D), t_prev=t_prev,
                y_prev=self.q, step_size=dt)
            self.c_surf = -(j*R)/(35*D) + 8 * R * self.q / 35 + self.c_s_avg_prev
        else:
            self.c_surf = -(R/D) * (j/5) + self.c_s_avg_prev

    def __call__(self, dt: float, t_prev: float, i_app: float, R: float, S: float,
         D_s: float, c_smax: float) -> float:
        self.solve(dt=dt, i_app=i_app, t_prev=t_prev, R=R, S=S, D=D_s)
        return self.c_surf / c_smax
```

### 4.4.4 Performance of the Different Numerical Schemes

The electrodes surface SOC evolution results during a battery discharge were graphed (Figure 5), and the solution time were noted (Table 1) on an system with AMD Ryzen 5600X CPU and 32GB RAM.

Table 1: Solution Times for Different Solid Electrode Conc. Numerical Solvers.

| Solver | Positive Electrode Solution time [s] | Negative Electrode Solution time [s] |
|---|---|---|
| CNSolver | 91.6 | 91.6 |
| EigenExpFunc | 13.1 | 13.0 |
| PolySolver | 0.141 | 0.145 |

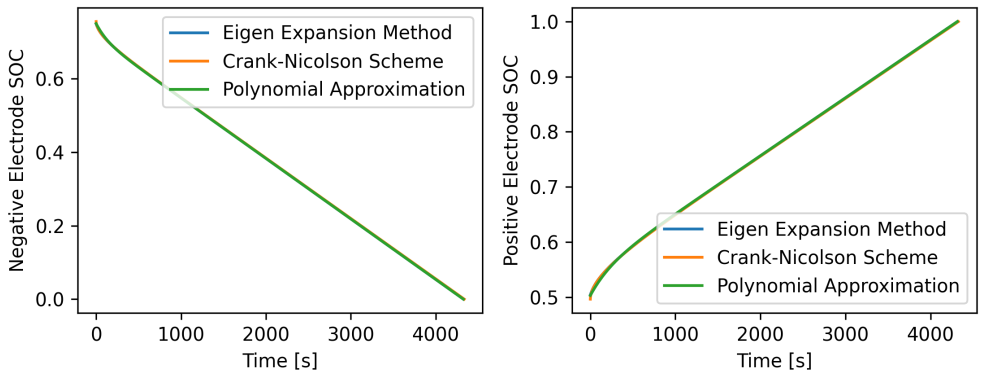

Figure 5: The negative (left) and positive(right) electrode SOC evolution.

While simulation solution time was achieved with ROM solver, the information of the spatial concentration profile in a time step is lost. This loss in information can be seen when the battery cell is rested after charge or discharge. The lithium concentration variation within the electrode particle results in diffusion voltages right after the charge or discharge operations. However, there is no change in the average lithium-ion volume concentration; therefore, the ROM solver is not able to model the cell terminal potential resulting from diffusion voltages (Figure 6).

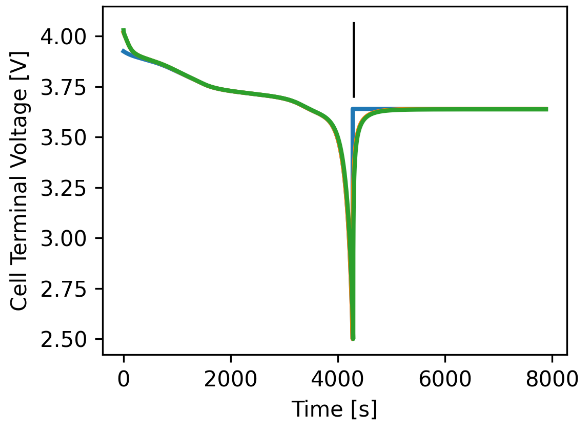

Figure 6: Volume averaging technique is not able to capture the diffusion voltages after battery discharge operations (time indicated by the vertical line).

## 4.5 SPPySolver

The `SPPySolver` is responsible for carrying out the simulation at all time steps and returning the `Solution` object, which contains the simulated results. Here, a method of `SPPySolver` is shown that carries out the steps in a single time iteration.

```
    def solve_iteration_one_step(self, t_prev: float, dt: float, I: float) ->
        float:
        # Account for SEI growth
        if self.bool_degradation:
            I_i, I_s, delta_R_SEI = self.SEI_model(SOC_n=self.b_cell.elec_n.SOC,
                OCP_n=self.b_cell.elec_n.OCP,
                                                   dt=dt,
                                                   temp=self.b_cell.elec_n.T,
                                                   I=I)  # update the
                                                       intercalation current (
                                                       negative electrode
            # only)
            self.b_cell.R_cell += delta_R_SEI  # update the cell resistance
            self.b_cell.electrolyte.conc -= -self.SEI_model.J_s * dt  # update the
                 electrolyte conc. to account
            # for mass balance.
        else:
            I_i = I  # intercalation current is same at the input current

        # Calc. electrode surface SOC below and update the battery cell's instance
             attributes.
        # if self.electrode_SOC_solver == 'eigen':
        self.b_cell.elec_p.SOC = self.SOC_solver_p(dt=dt, t_prev=t_prev, i_app=I,
                                                   R=self.b_cell.elec_p.R,
                                                   S=self.b_cell.elec_p.S,
                                                   D_s=self.b_cell.elec_p.D,
                                                   c_smax=self.b_cell.elec_p.
                                                       max_conc)  # calc p surf SOC
        self.b_cell.elec_n.SOC = self.SOC_solver_n(dt=dt, t_prev=t_prev, i_app=I_i
            ,
                                                   R=self.b_cell.elec_n.R,
                                                   S=self.b_cell.elec_n.S,
                                                   D_s=self.b_cell.elec_n.D,
                                                   c_smax=self.b_cell.elec_n.
                                                       max_conc)  # calc n surf SOC

        V = self.calc_terminal_potential(I_p_i=I, I_n_i=I_i)  # calc battery cell
            terminal voltage

        # Calc temp below and update the battery cell's temperature attribute.
        if not self.bool_isothermal:
            self.b_cell.T = self.calc_cell_temp(t_model=self.t_model, t_prev=
                t_prev, dt=dt,
                                                temp_prev=self.b_cell.T, V=V, I=I)
        return V
```

## 5 Simulation Examples

The subsections of this section contain the simulation script and the subsequent visualization as obtained from the `Solution` or `Plots` Object.

## 5.1 Battery Discharge in Isothermal Environment

```python
import SPPy


# Operating parameters
I = 1.656
T = 298.15
V_min = 3
SOC_min = 0.1
SOC_LIB = 0.9

# Modelling parameters
SOC_init_p, SOC_init_n = 0.4956, 0.7568  # conditions in the literature source.
    Guo et al

# Setup battery components
cell = SPPy.BatteryCell(parameter_set_name='test', SOC_init_p=SOC_init_p,
    SOC_init_n=SOC_init_n, T=T)

# set-up cycler and solver
dc = SPPy.Discharge(discharge_current=I, V_min=V_min, SOC_min=SOC_min, SOC_LIB=
    SOC_LIB)
solver = SPPy.SPPySolver(b_cell=cell, N=5, isothermal=True, degradation=False,
    electrode_SOC_solver='poly')

# simulate
sol = solver.solve(cycler_instance=dc)

print(sol.cycle_summary)

# Plot
sol.comprehensive_plot()
```

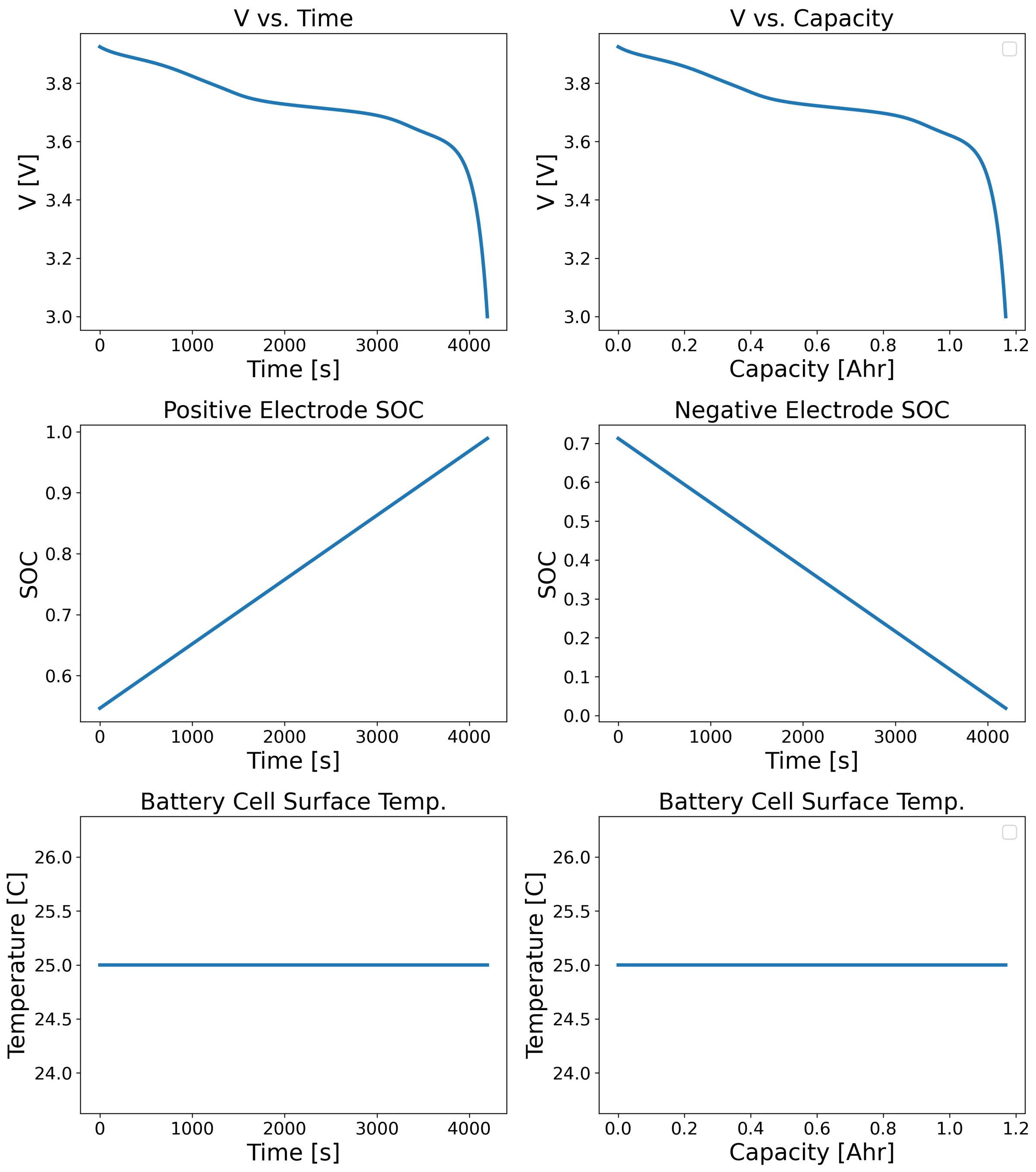

Figure 7: Simulation Results of Battery Discharge under Iosthermal Conditions.

# A Runge Kutta Methods

For an ODE of the form

$$\frac{dy}{dt} = f(t, y)$$

The forth order version of Runge-Kutta method (rk4) can be written as (where $h = t_{i+1} - t_i$ is the time step)[**Chapra2012**]

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) h) \tag{A.0.1}$$

Where

$$k_1 = f(t_i, y_i) \tag{A.0.2}$$

$$k_2 = f\left(t_i + \frac{1}{2}h,\, y_i + \frac{1}{2}k_1 h\right) \tag{A.0.3}$$

$$k_3 = f\left(t_i + \frac{1}{2}h,\, y_i + \frac{1}{2}k_2 h\right) \tag{A.0.4}$$

$$k_4 = f\left(t_i + h,\, y_i + k_3 h\right) \tag{A.0.5}$$

## A.1 Implementation in Python

A sample code for rk4 in Python is presented below. Note that this implementation uses the Python function as one of its input arguments.

```python
from typing import Callable


def rk4(func: Callable, t_prev: float, y_prev: float, step_size: float):
    """
    Solves for the value of y in the next time step for a ODE
                dy/dt = f(y,t)
    :param func: (Callable) function that takes y and t as its input arguments (in
         that order).
    :param t_prev: The value of time in the previous time step [s]
    :param y_prev: The value of y in the previous time step
    :param step_size: the difference in time between the current and previous time
         steps [s]
    :return: The value of y at the next time step
    """
    k1 = func(y_prev, t_prev)
    k2 = func(y_prev + 0.5*k1*step_size, t_prev + 0.5*step_size)
    k3 = func(y_prev + 0.5*k2*step_size, t_prev + 0.5*step_size)
    k4 = func(y_prev + k3*step_size, t_prev + step_size)
    return y_prev + (1/6.0) * (k1 + 2*k2 + 2*k3 + k4) * step_size
```

# B Thomas Algorithm for Tridiagonal Matrices

## B.1 Implementation in Python

This code was obtained from ref [**func˙TDMA**].

```python
import numpy as np
import numpy.typing as npt


def TDMAsolver(l_diag: npt.ArrayLike, diag: npt.ArrayLike, u_diag: npt.ArrayLike,
   col_vec: npt.ArrayLike) \
        -> npt.ArrayLike:
    '''
    TDMA (a.k.a Thomas algorithm) solver for tridiagonal system of equations.
    Code Modified from:
    https://gist.github.com/cbellei/8ab3ab8551b8dfc8b081c518ccd9ada9?
       permalink_comment_id=3109807
    '''
    nf = len(col_vec)  # number of equations
    c_l_diag, c_diag, c_u_diag, c_col_vec = map(np.array, (l_diag, diag, u_diag,
       col_vec))  # copy arrays
    for it in range(1, nf):
        mc = c_l_diag[it - 1] / c_diag[it - 1]
        c_diag[it] = c_diag[it] - mc * c_u_diag[it - 1]
        c_col_vec[it] = c_col_vec[it] - mc * c_col_vec[it - 1]

    xc = c_diag
    xc[-1] = c_col_vec[-1] / c_diag[-1]

    for il in range(nf - 2, -1, -1):
        xc[il] = (c_col_vec[il] - c_u_diag[il] * xc[il + 1]) / c_diag[il]
    return xc
```