Single Particle Model - Introduction and Implementation in Python

Moin Ahmed

August 13, 2023
Version 1.0

Contents

i List of Acronyms and Abbreviations

BV	Butler-Volmer Equation
CC	Constant Current Cycler
PDE	Partial Differential Equations
OCV	Open-circuit potential
rk4	Fourth-order version of Runge Kutta Method
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 $[mol/(m^2 s)]$
$i_{0,j}$	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
α_a	Anodic charge transfer co-efficient
α_c	Cathodic charge transfer co-efficient
η	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

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[4][6]

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

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

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

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 η_j is given by

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

After applying the values of α 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[3]. 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.

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

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

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

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'Hôpital's rule [5]

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

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

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

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

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

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

This tridiagonal matrix can be solved using the Thomas algorithm (refer to the Appendix B). Thomas algorithm needs the matrix to be diagonally dominant [8], 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 [3]. 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, λ_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 1).
A more detailed commentary on the code for the various class objects are discussed in the subsequent sections.

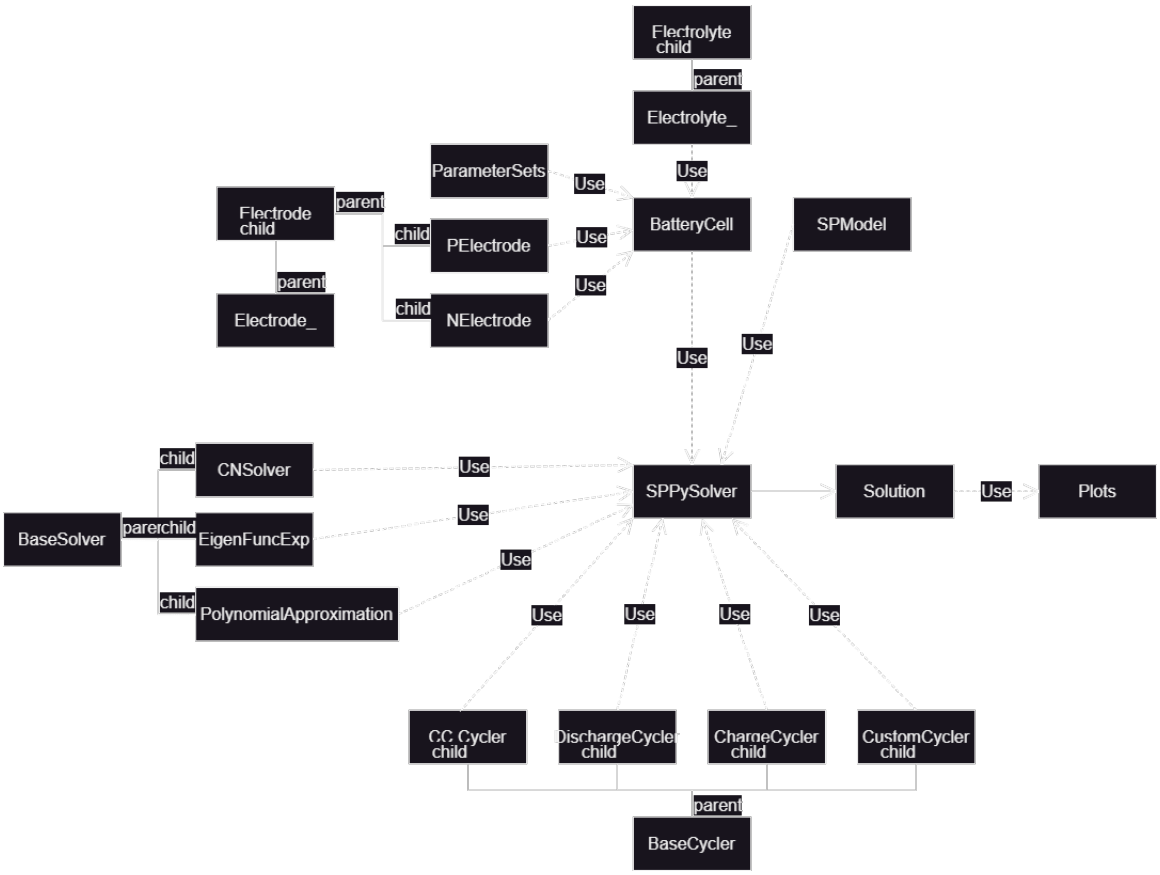

Figure 1: Conceptual Perspective of the Class Diagram in UML

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 Python's `@dataclass` decorator is used before defining the class name, and this decorator will automatically generate the class's `__init__` and `__repr__` methods[2]. This class is intended to store parameters such as the electrode's 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 electrolyte's initial concentration, conductivity, volume fraction, Bruggerman's 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 electrode's 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.

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

4.4.3 Reduced Order Model - Volume Averaging Technique

4.4.4 Performance of the Different Numerical Schemes

4.5 SPPySolver

5 Simulation Examples

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)[1]

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

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.

```
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 [7].

```
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
```

References

[1] Steven Chapra. "Applied Numerical Methods with MATLAB for Engineers and Scientists". In: McGraw Hill Education, 2012. Chap. Initial Value Problems.

[2] *dataclasses — Data Classes — Python 3.11.4 documentation*. NaN. URL: https://docs.python.org/3/library/dataclasses.html (visited on 08/12/2023).

[3] Meng Guo, Godfrey Sikha, and Ralph E. White. "Single-Particle Model for a Lithium-Ion Cell: Thermal Behavior". In: *Journal of The Electrochemical Society* 158 (2 Dec. 2011), A122. ISSN: 00134651. DOI: 10.1149/1.3521314/XML. URL: https://iopscience.iop.org/article/10.1149/1.3521314%20https://iopscience.iop.org/article/10.1149/1.3521314/meta.

[4] Hariharan Krishnan, Tagade Piyush, and Sanoop Ramachandran. "Mathematical Modeling of Lithium Batteries". In: Springer International Publishing 2018, 2018. Chap. Theoritical Framework of the Reduced Order Models.

[5] Jules Thibault, Simon Bergeron, and Hugues W. Bonin. "ON FINITE-DIFFERENCE SOLUTIONS OF THE HEAT EQUATION IN SPHERICAL COORDINATES". In: *Numerical Heat Transfer* 12.4 (Dec. 1987), pp. 457–474. DOI: 10.1080/10407788708913597. URL: https://doi.org/10.1080%2F10407788708913597.

[6] Marcello Torchio et al. "LIONSIMBA: A Matlab Framework Based on a Finite Volume Model Suitable for Li-Ion Battery Design, Simulation, and Control". In: *Journal of The Electrochemical Society* 163.7 (2016), A1192–A1205. DOI: 10.1149/2.0291607jes. URL: https://doi.org/10.1149%2F2.0291607jes.

[7] *Tridiagonal Matrix Algorithm solver in Python*. NaN. URL: https://gist.github.com/cbellei/8ab3ab8551b8dfc8b081c518ccd9ada9?permalink_comment_id=3109807 (visited on 08/13/2023).

[8] Yumpu.com. *Tridiagonal Matrices: Thomas Algorithm - University of Limerick*. NaN. URL: https://www.yumpu.com/en/document/read/4721668/tridiagonal-matrices-thomas-algorithm-university-of-limerick (visited on 08/12/2023).