libsim

Release 1.0

A. Caldwell, A. Preston, A. Valkonen, J. Xiang, J. Yanez

Dec 20, 2021

CONTENTS:

libsim is a Python library that creates battery simulation models.

Check out the *Usage* section for further information, including how to *install* the project.

Note: This project is under active development.

CHAPTER

ONE

BACKGROUND

1.1 Models

One of the research challenges in the development of lithium-ion batteries (LIBs) is to predict their behavior under different operating modes, which is useful for estimating state of charge and state of health of batteries in electric vehicles (EVs). There exist empirical models, mostly equivalent circuit-based, widely used in the Battery Management Systems (BMS) of electronics and EVs. These types of models use past experimental data of a battery to anticipate its future states. Most of the experimental data used to find charge/discharge characteristics rely on the current or cell potential. On one hand, these empirical models are relatively fast and simple computationally, but they have drawbacks. For example, the physics-based parameters are not able to be predetermined. The battery characteristics are not updated as the battery ages and a battery's model is unique to itself - it does not apply to all batteries but only a specific type.

Electrochemical models are, on the other hand, more sophisticated. These models are based on chemical/electrochemical kinetics and transport equations. They may be used to simulate the LIB's characteristics and reactions. Popular electrochemical models are the Pseudo-two-Dimensional (P2D) Model and Single Particle Model (SPM). The P2D model is commonly used in lithium-ion battery studies, and the predicted behavior of this model matches experimental data q uite accurately. A significant drawback with this model is the difficulty to use in real-time due to its computationally expensive nature. The SPM model simplifies anode/cathode interactions and reduces the dimensionality down to one dimension, which greatly enhances its computational capabilities. However, it places greater importance on the parameters of the anodes and cathodes.

CHAPTER

TWO

PROJECT GOALS

This project focuses on estimating and predicting the state of charge (SOC) and state of health (SOH) for lithium-ion batteries (LIBs) using a Single Particle Model (SPM) in order to reduce the computational cost and allow for the model to be implemented in real-time EV LIBs modeling. It accounts for the impact of complex parameters such as ion diffusivity, ion particle radius, and maximum ion concentration at the ion's surface on the performance of the battery.

Modelling these parameters is useful to compare to experimental results.

Eventually, this project can be expanded to estimate and predict the state of charge (SOC) and state of health (SOH) of LIBs, in order to reduce the computational cost and allow for the model to be implemented in real-time EV LIBs modeling.

CHAPTER

THREE

DESIGN PROCESS

In this code suite, abstracting battery behavior was not a trivial task. LIBs can be designed with many different types of anodes and cathodes which directly affect the electro-chemical properties and electrolyte interactions. To capture this variability, we decided to create a dictionary of different lithium-ion battery types that each have their own unique properties regarding diffusivity, particle radius, and ion concentration. This increases the versatility of the code suite to make the simulations widely applicable should an end user decide to test through various types of lithium-ion batteries of their choosing.

Finding a way to solve the first and second order derivatives of the SPM model was also not as simple as just plugging values into an equation. To remedy this, we decided to use a finite element method that involves using a mesh that is composed of nodes, which was necessary to analytically solve these governing equations.

The SPM model was used to simulate battery cycling behaviors. For generating solutions a finite element method was chosen. Architectural choices were made to allow for future implementation of various different model types, though we had only implemented the finite element method.

Our initial design was less modularized than the final version, as many of the tests we created to verify integration results relied on hard coded constants and input parameters that could not be generalized at the conception. Once testing was complete for one case of hard coded constants, we were able to allow command line arguments to be passed into the program that allowed for flexibility of the simulation. For another example, the electrode.py class was very overloaded with a multitude of functions nested inside of it, but upon further inspection these functions could easily exist in a separate file that doesn't need to be coupled with only the electrode class.

3.1 UML DIAGRAM

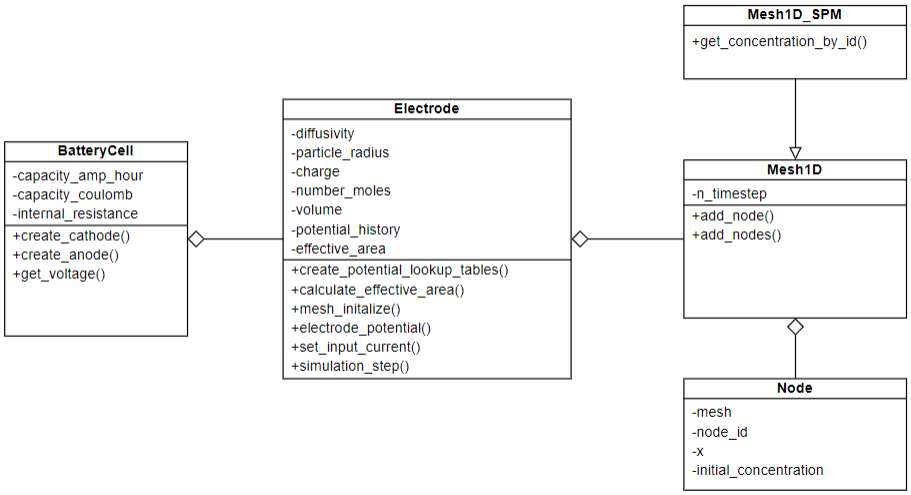

Fig. 1: UML Diagram for libsim.

CHAPTER

FOUR

LIBSIM

4.1 arguments module

4.2 batterycell module

4.3 derivative module

Derivative

derivative.**first_derivative**(*Mesh*, *coefficient*, *timestep*)

Calculates the first derivative in Fick's Law using a "phantom node".

Parameters

- **Mesh** (*[Mesh]*) – Mesh for which the derivative is to be evaluated.
- **coefficient** (*[double]*) – Scalar coefficient for the derivative.
- **timestep** (*[int]*) – Index for the timestep of the current derivative.

Returns The first derivative of Fick's Law for each of the nodes in the Mesh.

Return type [double]

derivative.**second_derivative**(*Mesh*, *coefficient*, *timestep*)

Calculates the second derivative in Fick's Law using a "phantom node".

param Mesh Mesh for which the derivative is to be evaluated.

type Mesh [Mesh]

param coefficient Scalar coefficient for the derivative.

type coefficient [double]

param timestep Index for the timestep of the current derivative.

type timestep [int]

return The second derivative of Fick's Law for each of the nodes in the Mesh.

rtype [double]

4.4 electrode module

4.5 main module

4.6 mesh module

Mesh

Parent class for classes that describe the computational mesh for each different type of model/problem.

class mesh.**Mesh1D**(*n_timestep*)

Bases: object

Mesh 1D is a 1-dimensional mesh composed of Nodes complete with functions to create Nodes. It includes an indexing system in order to be able to access individual nodes as needed (post-processing).

In order to initialize a Mesh1D, a variable n_timestep is needed.

Parameters **n_timestep** (*[int]*) – Number of timesteps to be taken.

add_node(*x*)

Adds a node based on x location, and returns a new node.

Parameters **x** (*[int]*) – Index identifier for location of the node.

Returns x

Return type [int]

add_nodes(*length*, *n_elements*)

Add nodes of a specified length with n_elements.

Parameters

- **length** (*[int]*) – Length of nodes to add.
- **n_elements** (*[int]*) – Number of nodes to add.

class mesh.**Mesh1D_SPM**(*n_timestep*)

Bases: *mesh.Mesh1D*

Mesh1D_SPM class inherits Mesh1D. Here the one dimensional Mesh is implemented for the use with SPM model.

add_node(*x*, *initial_concentration*)

Adds a node based on x location.

Parameters

- **x** (*[int]*) – Index identifier for location of the node.
- **initial_concentration** (*[double]*) – Initial concentration of ions at each node. [mol/(m^3)]

add_nodes(*length*, *n_elements*, *initial_concentration*)

Add nodes of a length with n_elements

Parameters

- **length** (*[int]*) – Length of nodes to add.
- **n_elements** (*[int]*) – Number of nodes to add.

- **initial_concentration** (*[double]*) – Initial concentration of ions at each node. [mol/(m^3)]

get_concentration_by_id(*node_id*, *timestep*)

Get concentration at a node, and return

Add nodes of a specified length with n_elements.

Parameters

- **node_id** (*[int]*) – Unique identifier for a node.
- **timestep** (*[int]*) – Current timestep.

Returns Node concentration

Return type [double]

4.7 node module

Node

Node is a parent class for classes that describe the points in the mesh for each type of problem. A Node is a general representation of a point in space. This parent class does not define the dimensionality or any other attributes.

class node.**Node**(*mesh*, *node_id*, *x*)

Bases: object

Node class

Parameters

- **mesh** (*mesh [Mesh]*) – Mesh
- **node_id** (*[int]*) – Unique identifier for a node.
- **x** (*x [int]*) – Index identifier for location of the node.

class node.**Node_SPM**(*mesh*, *node_id*, *x*, *initial_concentration*)

Bases: *node.Node*

Node_SPM class, subset of Node

Inherits Node. This is the implementation of Node for SPM modeling.

Parameters

- **mesh** (*[Mesh]*) – Mesh
- **node_id** (*[int]*) – Unique identifier for a node.
- **x** (*[int]*) – Index Identifier for location of the node.

4.8 plot module

4.9 solver module

CHAPTER

FIVE

USAGE

5.1 Installation

The third-party packages required are: *python3*, *numpy*, *scipy.interpolate*, *math*, and *matplotlib*. These packages can all be installed via *pip3*.

The repo can be cloned from *https://github.com/jerryzxiang/libsim.git* or installed via

5.2 Modeling Batteries

This program is run through the driver code file, *main.py*, which takes 8 command line arguments. As of now, there is no GUI. The inputs are the cathode, anode, input current, capacity of the cell in amp hours, the number of radial segments, the simulation time in seconds, and the time steps. An example is shown here for an LIB with an LFP cathode and graphite anode:

To get help, use *python3 main.py -h'*.

```
$ git clone https://github.com/jerryzxiang/libsim.git
```

CHAPTER

SIX

PROFILING

TO DO

CHAPTER

SEVEN

LESSONS LEARNED

Figured out good ways to verify tests when using a variety of parameters, as most of our work is looking at experimental data which is difficult to source.

Pair programming was very useful for catching bugs early on.

Commit often, establish workflow early

CHAPTER

EIGHT

FUTURE WORK

8.1 Graphic User Interface (GUI)

In order to simplify the utilization of libsim, we hope to integrate a Graphic User Interface (GUI) to allow for an intuitive user experience as opposed to terminal commands.

8.2 Tests

With the number of models implemented into libsim, the expected number of edge cases and foreseeable issues will also grow. We hope to increase the number of tests built into the code in order to reduce the possibility for inaccuracies to be generated.

8.3 Expanding Models Library

The final result of this project allowed for a computationally inexpensive model of a reduced order that adapts well for real-time applications. In future versions of libsim, we will look to implement PSeudo 2 Dimensional (P2D) model aspects to improve accuracy while maintaining time cost in mind.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX