Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Lithium-Ion Battery Simulator

Project Engineering

Year 4

Mark Veerasingam

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2024/2025

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Mark Veerasingam

Acknowledgements

I would like to express my sincere gratitude to all those who have supported and guided me throughout this project.

Firstly, I would like to extend my heartfelt thanks to my University Academic Supervisor, Michelle Lynch, for her continuous guidance, support, and insightful feedback, which were invaluable throughout the course of my project.

I am very grateful to my industry supervisors, Aloísio Kawakita de Souza and Jane Cornett from Analog Devices, for their mentorship and guidance. Their teachings and advice have been instrumental in shaping my understanding and approach to the project and I would not have been able to succeed in this project without their mentorship.

Special thanks to my manager, Sean Ó Connel, for his help in driving this opportunity for an industry-based final year project. His support and guidance have provided me with a unique and valuable experience throughout the course of this project.

Finally, I would like to acknowledge everyone who played a role in helping me achieve my goals throughout this journey. Their assistance, encouragement, and trust have been deeply appreciated.

Mark Veerasingam
06/04/2025

Table of Contents

1 Summary

The goal of this project was to develop a software platform for modeling **Equivalent Circuit Models** (**ECM**), a type of circuit model used to model lithium-ion batteries. This project creates battery models based on real-world test data as detailed in Onori et al. [1]. Offering users to simulate the batteries behavior under a custom user-defined conditions.

The projects scope includes data processing, identifying and optimizing the parameters of the equivalent circuit model to that of the test data, such that it models the real-world battery data and an API that allows user to simulate their own data-driven constructed model.

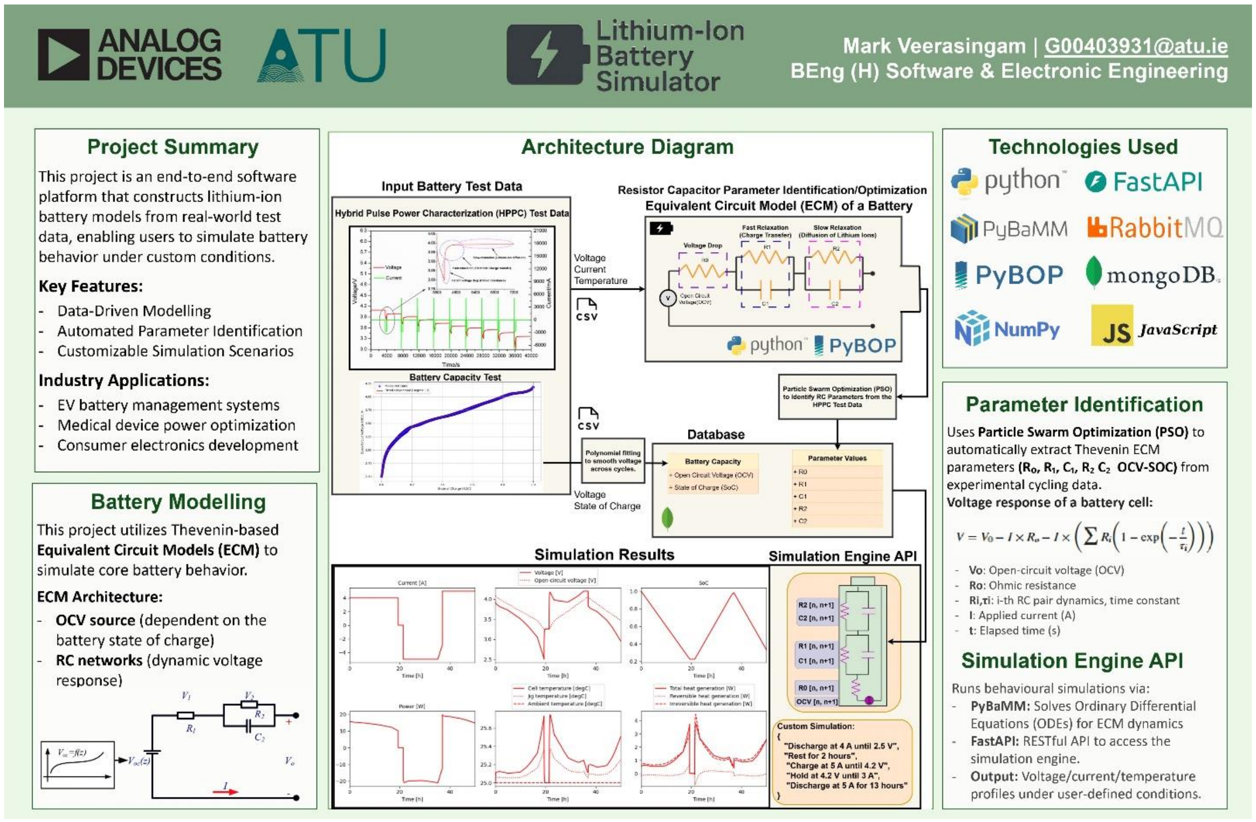
Key features of this project include **Data-Drive Modelling**, which uses real-world battery test data to construct a Thevenin ECM, to capture the battery behavior. **Parameter Extraction and Optimization** involves deriving ECM parameters from the fitted test data. The project's **Simulation API** is a dedicated API developed to solve the systems of differential equations that govern the Battery model. It is this that enables user-definable simulation scenarios.
The project approach began with processing battery test data, followed by fitting a Thevenin ECM through parameter extraction and optimization. A simulation API was then built on top of PyBaMM to solve the model's ODEs, enabling flexible, user-defined simulation scenarios.

Main Methods & Technologies Used: Python, PyBaMM (Python Battery Mathematical Modelling), PyBop (Python Battery Optimization), Numpy, FastAPI, Redis and MongoDB

What Was Accomplished:
The project successfully developed an end-to-end platform for data-driven battery modelling, automating parameterization and enabling intuitive scenario-based simulation engine allowing users to explicitly verbally describe the simulation.

Traditional battery modelling software is device constrained, for example MATLAB.
My project offers a scalable, solution to battery modelling, offering a modernized approach to scientific computing, benefiting industries like electric vehicle management, medical devices, and consumer electronics.

2 Poster



ANALOG DEVICES ATU

Lithium-Ion Battery Simulator

Mark Veerasingam | G00403931@atu.ie
BEng (H) Software & Electronic Engineering

Project Summary

This project is an end-to-end software platform that constructs lithium-ion battery models from real-world test data, enabling users to simulate battery behavior under custom conditions.

Key Features:

- Data-Driven Modelling
- Automated Parameter Identification
- Customizable Simulation Scenarios

Industry Applications:

- EV battery management systems
- Medical device power optimization
- Consumer electronics development

Battery Modelling

This project utilizes Thevenin-based **Equivalent Circuit Models (ECM)** to simulate core battery behavior.

ECM Architecture:

- **OCV source** (dependent on the battery state of charge)
- **RC networks** (dynamic voltage response)

Architecture Diagram

Input Battery Test Data
Hybrid Pulse Power Characterization (HPPC) Test Data
Voltage
Current
Battery Capacity Test
Voltage
Current
Temperature
CSV
CSV
Resistor Capacitor Parameter Identification/Optimization
Equivalent Circuit Model (ECM) of a Battery
Voltage Drop
Fast Relaxation (Charge Transfer)
Slow Relaxation (Diffusion of Lithium Ions)
R0
R1
R2
C1
C2
Open Circuit Voltage(OCV)
python
PyBOP
Particle Swarm Optimization (PSO) to identify RC Parameters from the HPPC Test Data
Polynomial fitting to smooth voltage across cycles.
Voltage
State of Charge
Database
Battery Capacity
+ Open Circuit Voltage (OCV)
+ State of Charge (SoC)
Parameter Values
+ R0
+ R1
+ C1
+ R2
+ C2
Simulation Results
Current [A]
Voltage [V]
Open-circuit voltage [V]
SoC
Power [W]
Cell temperature [degC]
Jig temperature [degC]
Ambient temperature [degC]
Total heat generation [W]
Reversible heat generation [W]
Irreversible heat generation [W]
Time [h]
Simulation Engine API
R2 [n, n+1]
C2 [n, n+1]
R1 [n, n+1]
C1 [n, n+1]
R0 [n, n+1]
OCV [n, n+1]

Custom Simulation:

```
{
  "Discharge at 4 A until 2.5 V",
  "Rest for 2 hours",
   "Charge at 5 A until 4.2 V",
   "Hold at 4.2 V until 3 A",
   "Discharge at 5 A for 13 hours"
}
```

Technologies Used

Parameter Identification

Uses **Particle Swarm Optimization (PSO)** to automatically extract Thevenin ECM parameters **(R_o, R_1, C_1, R_2 C_2 OCV-SOC)** from experimental cycling data.

Voltage response of a battery cell:

$$V = V_0 - I \times R_o - I \times \left(\sum R_i\left(1 - \exp\left(-\frac{t}{\tau_i}\right)\right)\right)$$

- **Vo**: Open-circuit voltage (OCV)
- **Ro**: Ohmic resistance
- **Ri,τi**: i-th RC pair dynamics, time constant
- **I**: Applied current (A)
- **t**: Elapsed time (s)

Simulation Engine API

Runs behavioural simulations via:

- **PyBaMM:** Solves Ordinary Differential Equations (ODEs) for ECM dynamics
- **FastAPI:** RESTful API to access the simulation engine.
- **Output:** Voltage/current/temperature profiles under user-defined conditions.

3 Github Repositories

- **Battery Simulator API [10]:**
    - https://github.com/MarkVeerasingam/Battery-Simulator
- **ECM Parameterisation [7]:**
    - https://github.com/MarkVeerasingam/BatteryECM_Identification_API
- **Jupyter notebook to prototype battery data processing [9]:**
    - https://github.com/MarkVeerasingam/LGM50t_cell_analysis/tree/main

4 Introduction

The aim of this project is to develop a software platform capable of simulating lithium-ion battery behavior using *equivalent circuit models (ECMs)* derived from real-world test data [1]. The simulator enables users to define custom testing scenarios, offering a virtual testbed to model a battery and explore the models' dynamics under varying operating conditions.

4.1 Project Goal

This project is motivated by the growing demand for not only fast but computationally efficient battery models in fields such as electric vehicles, energy storage systems, mobile technologies and *battery management systems (BMS).* By leveraging empirical data and modeling it through Thevenin-based ECMs; a popular and industry battery circuit model. The platform provides a practical and flexible tool for research and development purposes.

Most battery development software is written in *MATLAB*, with proprietary packages like *SIMSCAPE* are typically constrained to a device. My projects API approach with Python, decouples this and lays the foundation for scalable and modular battery development workflows.

4.2 Project Scope

The scope of the project includes processing and fitting real battery test data, extracting and optimizing ECM parameters with *PyBop (Python Battery Optimization),* and developing a Python-based simulation API using the *PyBaMM (Python Battery Mathematical Modelling)* framework to solve the governing ordinary differential equations. The final software allows users to run time-domain simulations and analyze voltage, current, and *state-of-charge (SoC)* behavior for different battery configurations.

5 Background

This project draws heavily from my internship experience at **Analog Devices**, a U.S.-based semiconductor company, where I worked on embedded software for electric *vehicles Battery Management Systems (BMS).*

5.1 Analog Devices Internship

After completing my second year of university, I was awarded an academic scholarship by Analog Devices and began interning with them in the summer of 2023, continuing part-time throughout my studies since. During this time, I was introduced to a variety of software engineering fields, including control systems and modelling. After my first internship, I knew I wanted to peruse a final year project in the realm of battery modeling and simulation.

5.2 Battery Modelling Experience

During my third-year placement, I worked closely with their battery modeling team. With guidance from my industry supervisor, we began brainstorming for a clear project proposal that would be ready, for when I return to university.

5.3 Project Motivation

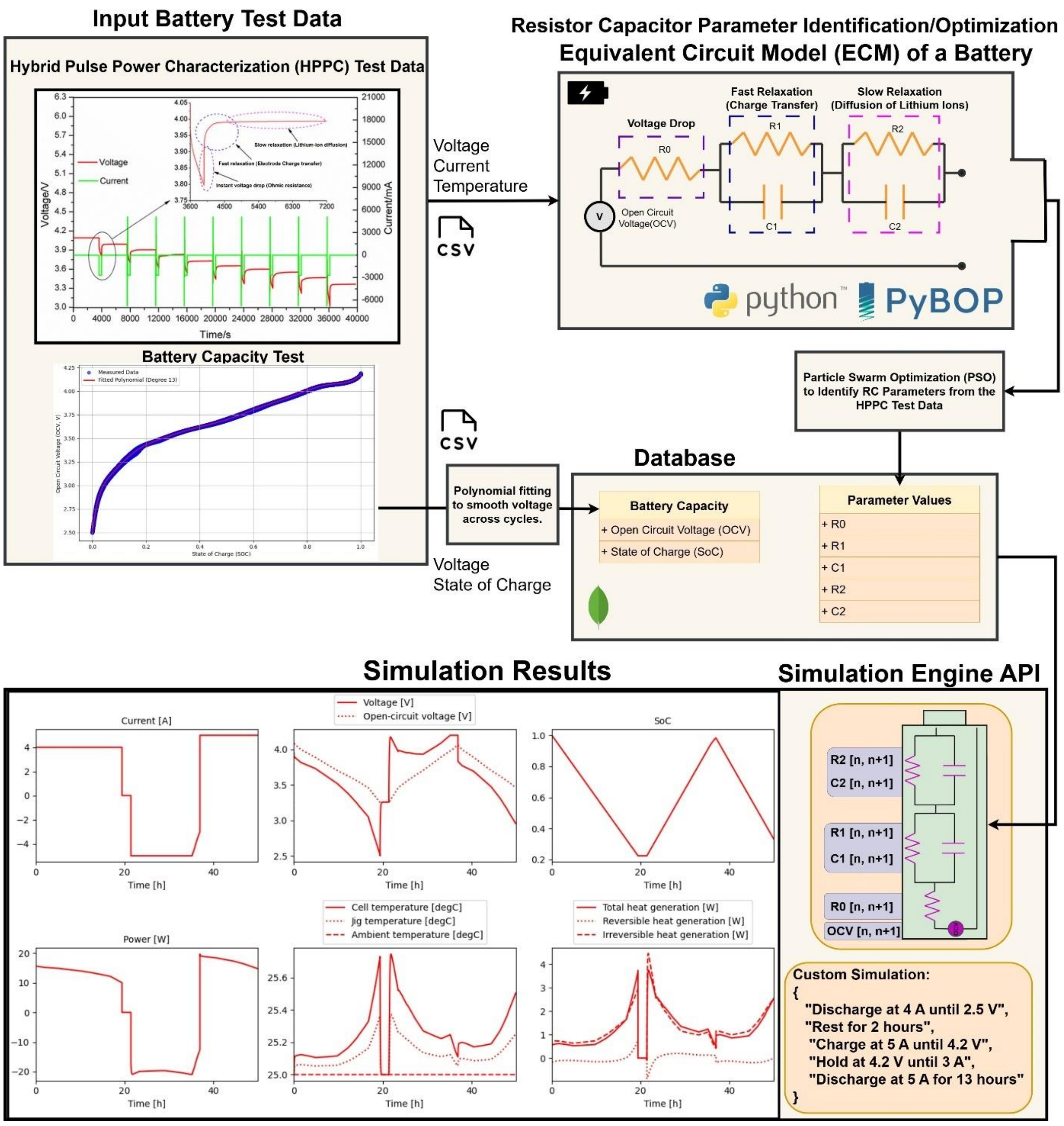
This project aims to take advantage of Python's growing ecosystem and open-source tools like *PyBaMM* and *PyBop* to build an API-driven approach to battery modelling, with the goal to lay the groundwork for one of the first cloud-based battery simulation platforms.

5.4 Project Research

The main research behind my project was in math-modelling. I have already been introduced to the primary technologies used in battery modelling, making the research of this project fall heavy on the theory behind battery modelling and in battery domain knowledge such as testing.

My projects research expanded upon this into researching the needed requirements to model a battery from its real-world test data, this included research algorithms like *Particle Swarm Optimization (PSO)* and understanding battery test data like *Hybrid Pulse Power Characterization (HPPC)* and capacity testing data.

6 Architecture Diagram

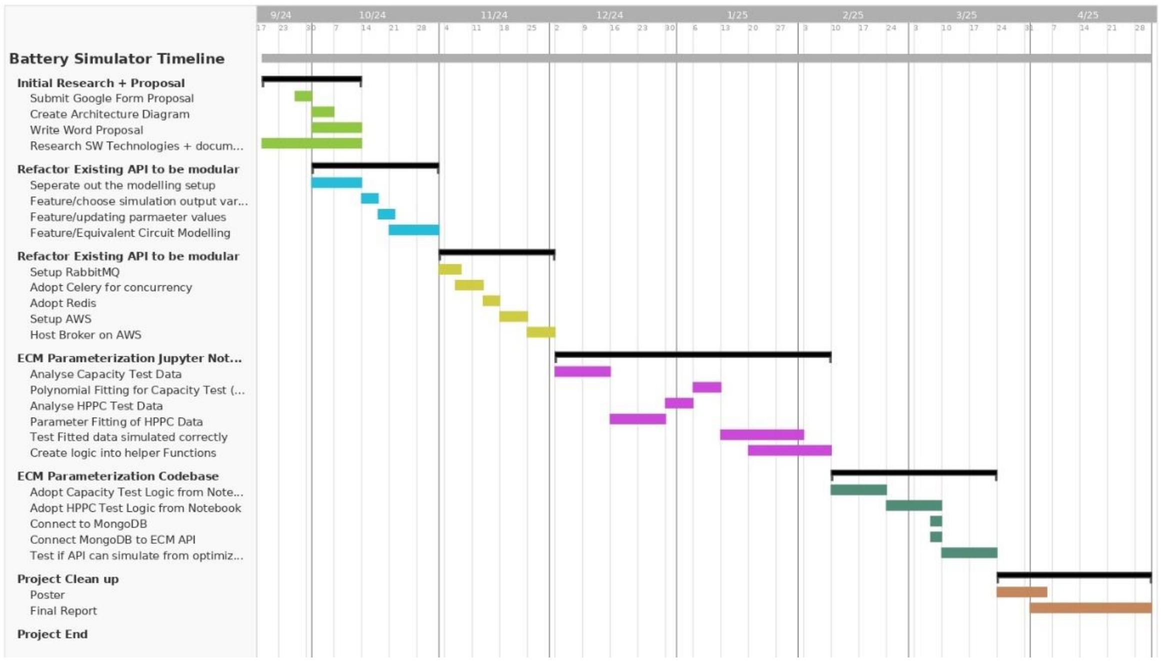

Input Battery Test Data
Hybrid Pulse Power Characterization (HPPC) Test Data
Voltage
Current
Slow relaxation (Lithium-ion diffusion)
Fast relaxation (Electrode Charge transfer)
Instant voltage drop (Ohmic resistance)
Voltage/V
Current/mA
Time/s
Battery Capacity Test
Measured Data
Fitted Polynomial (Degree 13)
Open Circuit Voltage (OCV, V)
State of Charge (SOC)
Voltage
Current
Temperature
CSV
Resistor Capacitor Parameter Identification/Optimization
Equivalent Circuit Model (ECM) of a Battery
Voltage Drop
R0
Fast Relaxation
(Charge Transfer)
R1
C1
Slow Relaxation
(Diffusion of Lithium Ions)
R2
C2
V
Open Circuit
Voltage(OCV)
python
PyBOP
Particle Swarm Optimization (PSO) to Identify RC Parameters from the HPPC Test Data
CSV
Polynomial fitting to smooth voltage across cycles.
Voltage
State of Charge
Database
Battery Capacity
+ Open Circuit Voltage (OCV)
+ State of Charge (SoC)
Parameter Values
+ R0
+ R1
+ C1
+ R2
+ C2
Simulation Results
Current [A]
Voltage [V]
Open-circuit voltage [V]
SoC
Power [W]
Cell temperature [degC]
Jig temperature [degC]
Ambient temperature [degC]
Total heat generation [W]
Reversible heat generation [W]
Irreversible heat generation [W]
Time [h]
Simulation Engine API
R2 [n, n+1]
C2 [n, n+1]
R1 [n, n+1]
C1 [n, n+1]
R0 [n, n+1]
OCV [n, n+1]
Custom Simulation:

```
{
  "Discharge at 4 A until 2.5 V",
  "Rest for 2 hours",
   "Charge at 5 A until 4.2 V",
   "Hold at 4.2 V until 3 A",
   "Discharge at 5 A for 13 hours"
}
```

7 Project Management

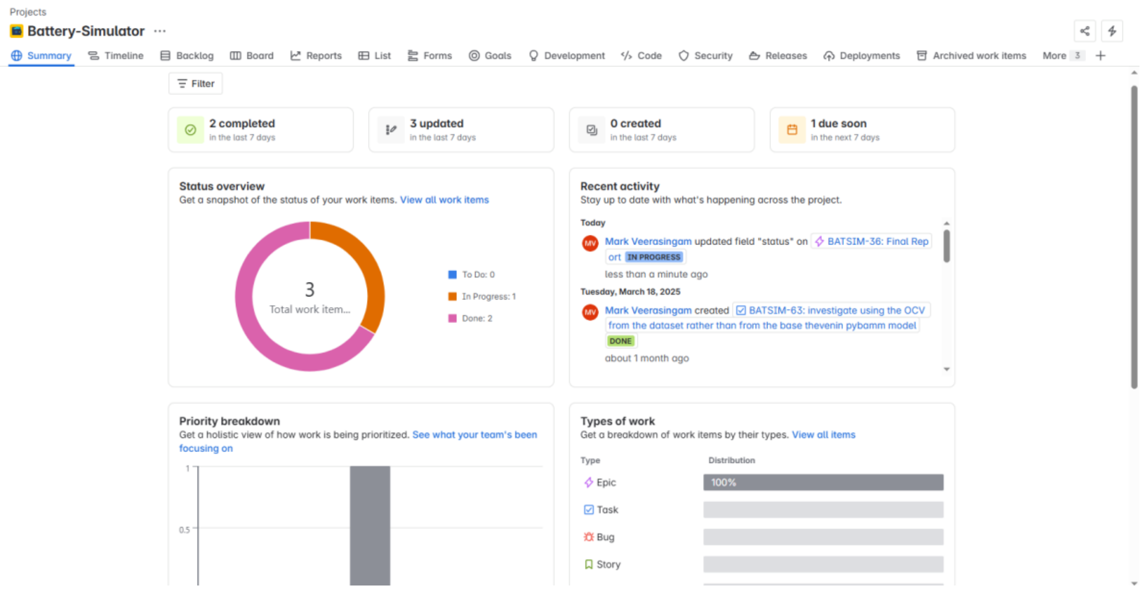
7.1 Project Timeline

Fig A: Project Timeline



7.2 Jira

Fig A.2: Jira Board page project summary


Projects

Battery-Simulator ···

Summary Timeline Backlog Board Reports List Forms Goals Development Code Security Releases Deployments Archived work items More 3 +

Filter

2 completed
in the last 7 days

3 updated
in the last 7 days

0 created
in the last 7 days

1 due soon
in the next 7 days

Status overview
Get a snapshot of the status of your work items. View all work items

3
Total work item...

To Do: 0
In Progress: 1
Done: 2

Recent activity
Stay up to date with what's happening across the project.

Today

Mark Veerasingam updated field "status" on BATSIM-36: Final Report IN PROGRESS
less than a minute ago

Tuesday, March 18, 2025

Mark Veerasingam created BATSIM-63: investigate using the OCV from the dataset rather than from the base thevenin pybamm model DONE
about 1 month ago

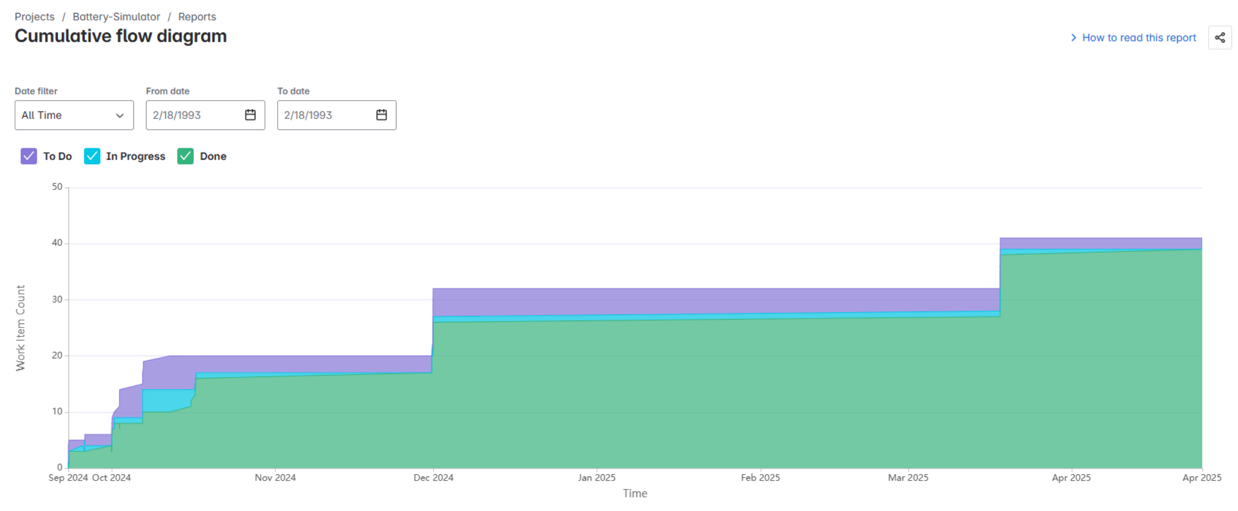
Priority breakdown
Get a holistic view of how work is being prioritized. See what your team's been focusing on

1
0.5

Types of work
Get a breakdown of work items by their types. View all items

Type	Distribution
Epic	100%
Task	
Bug	
Story	

Fig A.3: Jira Cumulative Flow diagram report



Project management was a requirement for this project, while relatively simple, my personal report was that it was the more challenging aspects of the project.

My timeline changed about 3-4 times throughout this project. Originally in my project proposal, the approach I took was that I wanted to create a generalist battery simulator leveraging the existing models native to PyBaMM, rather the create models from real world test-data. but in favour for a more academically challenging project, I chose the latter.

Between understanding the technologies and theory needed to undergo this endeavour, it took a considerable amount of time, far more than I expected. This took up majority of my time in my 2nd semester.

8 Lithium-Ion Battery Test and Data

8.1.1 Stanford University Dataset

For this project, I used the publicly available battery test dataset "*Lithium-ion battery aging dataset based on electric vehicle real-driving profiles*" published by *Stanford University* [1].

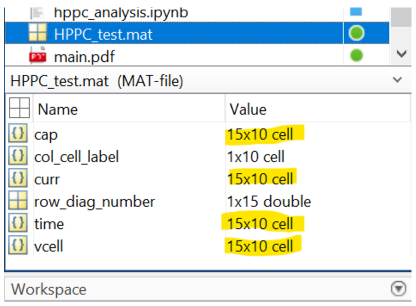
This dataset includes two primary types of tests: the *Hybrid Pulse Power Characterization (HPPC)* test and the *Capacity test*. These tests offer the data needed to model battery behavior over time and I will cover their methodologies in the following sections.

The HPPC and Capacity tests were conducted under a variety of experimental conditions and across different cycle counts to capture the effects of aging. These experiments were performed on *LGM50 lithium-ion cells*, a common 21700 cylindrical battery.

These conditions included changes in **temperature**, **charge/discharge profiles**, and **drive cycle patterns**, creating a wide range of test scenarios. Each condition generated a unique sequence of **cycles**, where one *cycle* refers to a full charge and discharge of the battery. The data collected from these cycles is key to understanding how the battery ages.
Each experiment is labeled with an identifier like **"W4," "G1," "V4,"** or **"V5,"** with no specific naming pattern. For instance, **"G1"** represents a specific test setup and includes multiple cycles of data. These labels group together all cycles collected under the same testing conditions.

Understandably, this level of organization made the heavily nested, on top of its datatype being in MATLAB .mat struct format, where each nest was a different discharge-cycle/experiment for either the HPPC or Capacity test. To work with this format requires me to use Scipy.io module to load the MATLAB data and Iterate through the nested data.

Fig B: Data Structure of the Stanford University Dataset



Name	Value
cap	15x10 cell
col_cell_label	1x10 cell
curr	15x10 cell
row_diag_number	1x15 double
time	15x10 cell
vcell	15x10 cell

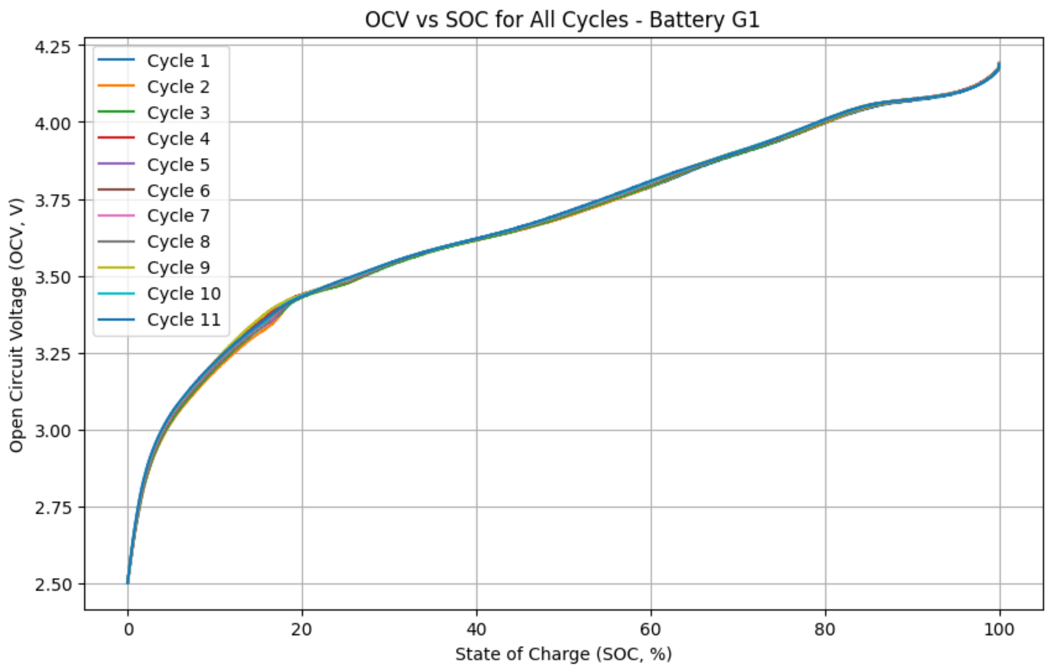
8.1.2 Capacity Test

A battery capacity test involves a constant current be applied to a discharge to a full charged battery until the battery is depleted. The purpose of this test is to determine a baseline measurement of how much usable capacity remains in the cell. This test is repeated under identical conditions to monitor the batteries aging process over time with respect to how the capacity fades over each cycle. [2]

In the dataset used for this project, several **LGM50 lithium-ion cells** were tested under different experimental conditions. From the dataset, one of the battery experiments labelled **"G1,"** the cell was charged at a **3C rate** with an ambient temperature of **23°C.**

The capacity test involved discharging the battery at a constant current of **1.5 A** down to **2.5 V**, following a full charge. This procedure was repeated multiple times under the same conditions, producing a series of discharge cycles used to evaluate capacity degradation over time.

Fig B.2: Voltage vs State of Charge Plotting for all cycles across Capacity Test G1



The graph above shows the results of the "G1" test, showing the battery's behavior across those 11 discharge cycles. From this test data, our goal is to capture the degradation mechanism affecting the battery over time. To capture this, we record what is called the batteries **State of Charge (SoC)** vs **Open Circuit Voltage (OCV)** relationship

- **Open Circuit Voltage (OCV):**
  - OCV is the voltage measured across the battery terminals when no current is flowing.
- **State of Charge (SoC):**

- SoC indicates how full a battery is, usually expressed as a percentage (%) i.e. 100% SoC = fully charged, 0% SoC = completely discharged.
- SoC is not a parameter that is available from the dataset, but something I must calculate from the available columns, "current", "voltage" and the "capacity" of the battery. [5]

  Fig B.3: Coulomb Counting State of Charge Estimation Algorithm

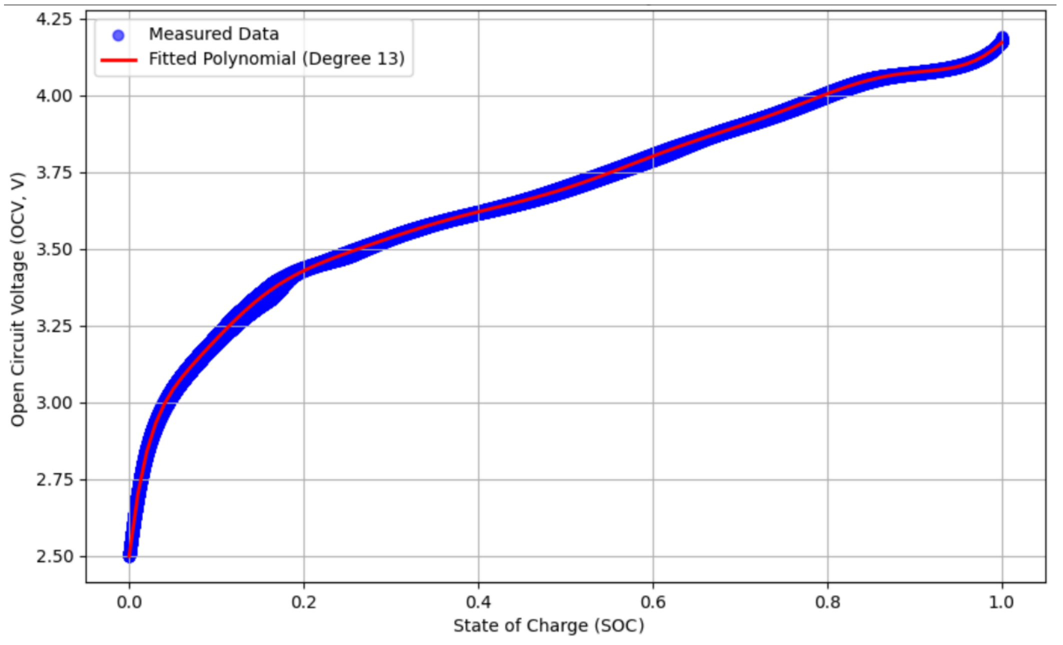
  $$SoC(t) = SoC(t-1) + \frac{I(t)}{Q_n}\Delta t$$
- Where…
  - SoC(t-1) = previous charge index. This would need to start at 100, to represent the battery fully charged before calculating.
  - I(t) = current at the given time
  - Qn = the maximum charge of the battery, in my instance as described in the dataset, the maximum charge of the LGM50 battery is 4.85 Ah (ampere-hours). However, in my calculation I took the highest capacity value from the cycle in the dataset.

Since OCV is not directly measurable during charging/discharging (because of internal resistance and voltage drop), models are used to estimate SOC using known OCV vs SOC curves. Capturing this behavior is crucial for understanding the battery's capacity loss and aging characteristics over time.

8.1.3 Capacity Test – Polynomial Fitting

To capture the aging characteristics of the battery, the relationship between the Open Circuit Voltage (OCV) and State of Charge (SoC), a **polynomial fitting** will be performed across the 11 discharge cycles. Depending on the chosen degree of the polynomial, this method smooths out the voltage and captures a consistent OCV-SOC trend over the cycles.

Fig B.4: OCV-SoC curve plotting from fitting all cycles across Capacity Test G1



In the graph above, the **red line, "Fitted Polynomial"** is the fitted curve, serving as a representation of the battery's voltage response and capacity behavior over time.

I save this in a .csv database with the columns "State of Charge" and "Open Circuit Voltage". This data will be used as a key input for modelling the voltage response in the model.

Before implementing the capacity test analysis, fitting and State of Charge estimation into my main codebase for the parameter identification, I used a Jupiter notebook to protype out all the capacity test analysis. [9]

8.1.4 Hybrid Pulse Power Characterization (HPPC)

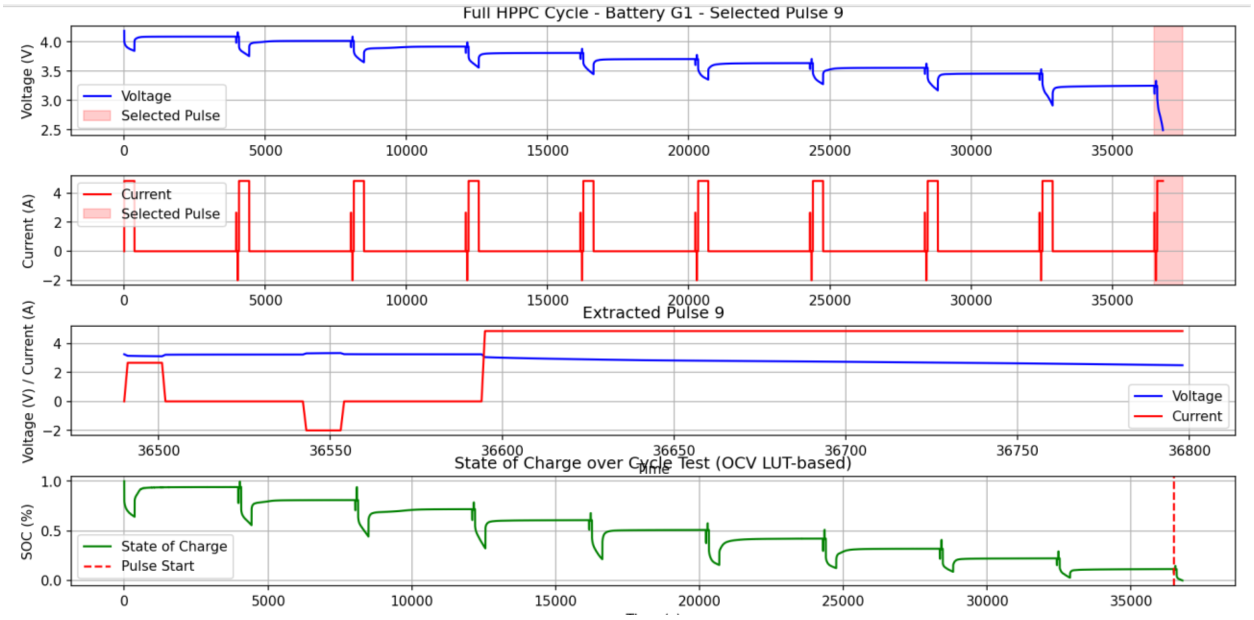
The **Hybrid Pulse Power Characterization (HPPC)** test assesses a battery's **internal resistance** and its dynamic voltage response to pulsed loads.

In this test, a series of **charge and discharge pulses** are applied to the battery at various **State of Charge (SOC)** levels for example at 100% to 90%, 90% to 80% etc… Each pulse typically lasts for a short duration (e.g., 10 seconds), and is followed by a rest period to allow the battery to return to "rest", allowing the battery to stabilize, approaching its **Open Circuit Voltage (OCV)**. [3]

It is with the HPPC Test that allows us to test the dynamic response of the battery at various charge levels, to capture how the batteries voltage response is at these varying charge levels.

Below is a graph of the "G1" battery HPPC Test Results.

Fig B.5: HPPC Test Results for Cycle 1 from the G1 Battery Experiment



These pulse-rest sequences help to:

- Observe **voltage drop** during high current discharge or charge events,
- Track **voltage recovery** after the pulse ends,
- Measure **internal resistance** based on the voltage change and applied current,
- Understand how the **dynamic response** varies with SOC and aging.

From the HPPC test, we can extract important parameters such as:

- **DC internal resistance**, by measuring the instantaneous voltage drop during the pulse,
- **Open Circuit Voltage (OCV)** at various SOC levels, after rest periods,
- **Dynamic behaviour** under load, which is useful for constructing equivalent circuit models (e.g., Thevenin Equivalent Circuit Models).

Before implementing this analysis into my main codebase for the parameter identification, I used a Jupyter notebook to protype out all the HPPC analysis and pulse extraction. [9]

9 Battery Modelling

9.1 Equivalent Circuit Models (ECM)

This section will cover the core modelling methodology and technical detail to simulate and model lithium-ion batteries.

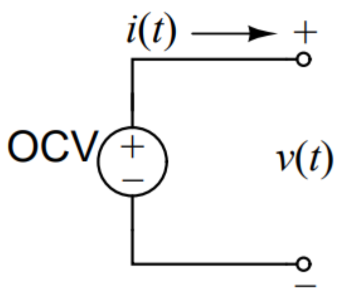
In the modelling world, circuits are used to model a variety of things that represent real world behaviour, from neurons in the brain to battery behaviour. This is because circuits can demonstrate how energy transforms or moves over time by manipulating their components.

9.1.1 Open Circuit Voltage (OCV)

This is the simplest model of a battery. [6]

- Voltage is not a function of current
- Voltage is not a function of past usage
- Voltage is a constant

Fig C: Open Circuit Voltage Circuit, referenced from G. Plett, "ECE4710/5710" [6]

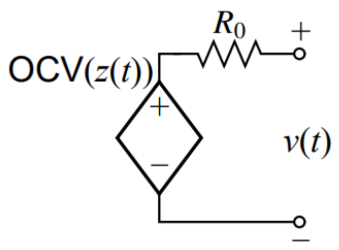


The goal of this model is to be a building block towards a more complete model. When a battery is unloaded and in rest otherwise known as "open circuit", the voltage is predictable. An ideal voltage source will be part of our equivalent circuit model.

9.1.2 Equivalent series resistance

A cell's voltage drops when it is under load. This can be modeled, in part, as a resistance in series with the ideal voltage source. As shown in the diagram below.

Fig C.1: Series Resistor in an Open Circuit Voltage Circuit, referenced from G. Plett, "ECE4710/5710"[6]



Where Z is State of Charge

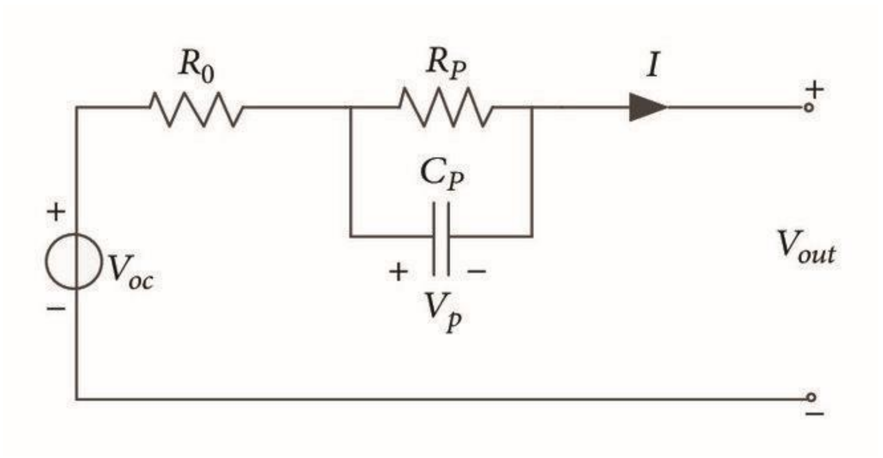
9.1.3 Thevenin Model

While the Open Circuit Voltage and Equivalent Series Resistance capture basic steady-state behaviour, real lithium-ion batteries exhibit **dynamic voltage responses** due to internal electrochemical processes.

To account for this behaviour, we introduce an RC (Resistor-Capacitor) pair, commonly known as a **Thevenin pair**—into the model.

Below is a diagram of the Thevenin Equivalent Circuit Model of a Lithium-Ion Battery.

Fig C.2: 1 Resistor-Capacitor Pair Thevenin Equivalent Circuit Model of a Lithium-Ion Battery



The Thevenin model builds on the OCV-based approach by simulating how the battery voltage deviates from the OCV during dynamic operation when current is applied.

- R0 = represents the instantaneous voltage drop because of internal resistance.
- R1/C1 = models the slower, delayed response of the terminal voltage. This RC network captures diffusion and relaxation effects caused by lithium-ion reaction within the cell.

By using a Thevenin Equivalent Circuit Model, it bridges the “gap” between OCV and real terminal voltage under load.

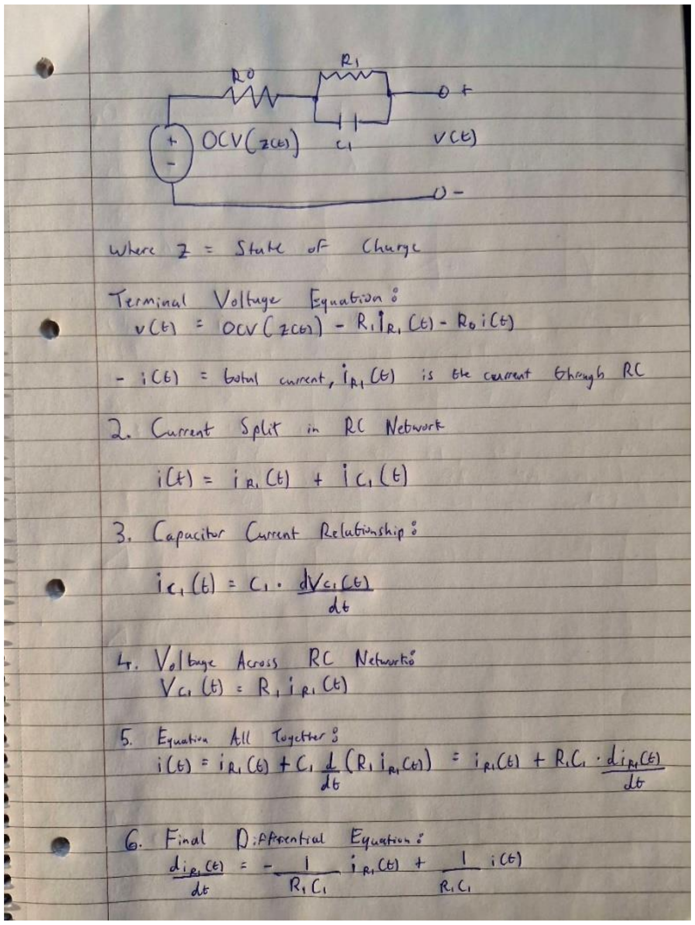
When a battery is at rest, its voltage does not instantly return to OCV, but rather gradually recovers to OCV over time.

This slow voltage change is due to internal diffusion processes within the cell, a behavior. We use the Thevenin RC network pair to model this slower voltage response. [6]

Below is the mathematics to model the dynamic behavior of a Thevenin Equivalent Circuit Model for a lithium-ion battery. Specifically focus on how the RC network models the voltage and current response over time in response to an applied current.

9.1.4 Expressing an ECM as a Differential Equation

Fig C.3: Pen and paper mathematic expression of an ECM as a Differential Equation.


R0
R1
C1
OCV(z(t))
+
-
v(t)
+
-

Where z = State of Charge

Terminal Voltage Equation:

$v(t) = OCV(z(t)) - R_1 i_{R_1}(t) - R_0 i(t)$

- $i(t)$ = total current, $i_{R_1}(t)$ is the current through RC

2. Current Split in RC Network

$i(t) = i_{R_1}(t) + i_{C_1}(t)$

3. Capacitor Current Relationship:

$i_{C_1}(t) = C_1 \cdot \frac{dV_{C_1}(t)}{dt}$

4. Voltage Across RC Network:

$V_{C_1}(t) = R_1 i_{R_1}(t)$

5. Equation All Together:

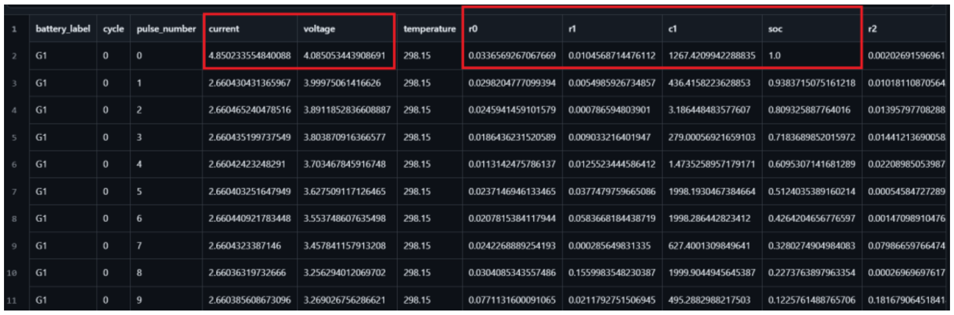
$i(t) = i_{R_1}(t) + C_1 \frac{d}{dt}(R_1 i_{R_1}(t)) = i_{R_1}(t) + R_1 C_1 \cdot \frac{di_{R_1}(t)}{dt}$

6. Final Differential Equation:

$\frac{di_{R_1}(t)}{dt} = -\frac{1}{R_1 C_1} i_{R_1}(t) + \frac{1}{R_1 C_1} i(t)$

Below is an example of some final parameterized results of a 10 pulse HPPC test found in my GitHub Repository, where I performed the parameter identification and optimization. [7]

Fig C.4: Results of an already optimized RC pair values from test data



battery_label	cycle	pulse_number	current	voltage	temperature	r0	r1	c1	soc	r2
G1	0	0	4.850233554840088	4.085053443908691	298.15	0.0336569267067669	0.0104568714476112	1267.4209942288835	1.0	0.00202691596961
G1	0	1	2.660430431365967	3.99975061416626	298.15	0.0298204777099394	0.0054985926734857	436.4158223628853	0.9383715075161218	0.01018110870564
G1	0	2	2.660465240478516	3.8911852836608887	298.15	0.0245941459101579	0.000786594803901	3.186448483577607	0.809325887764016	0.01395797708288
G1	0	3	2.660435199737549	3.803870916366577	298.15	0.0186436231520589	0.009033216401947	279.00056921659103	0.7183689852015972	0.01441213690058
G1	0	4	2.66042423248291	3.703467845916748	298.15	0.0113142475786137	0.0125523444586412	1.4735258957179171	0.6095307141681289	0.02208985053987
G1	0	5	2.660403251647949	3.627509117126465	298.15	0.0237146946133465	0.0377479759665086	1998.1930467384664	0.5124035389160214	0.00054584727289
G1	0	6	2.660440921783448	3.553748607635498	298.15	0.0207815384117944	0.0583668184438719	1998.286442823412	0.4264204656776597	0.00147098910476
G1	0	7	2.6604323387146	3.457841157913208	298.15	0.0242268889254193	0.000285649831335	627.4001309849641	0.3280274904984083	0.07986659766474
G1	0	8	2.66036319732666	3.256294012069702	298.15	0.0304085343557486	0.1559983548230387	1999.9044945645387	0.2273763897963354	0.00026969697617
G1	0	9	2.660385608673096	3.269026756286621	298.15	0.0771131600091065	0.0211792751506945	495.2882988217503	0.1225761488765706	0.18167906451841

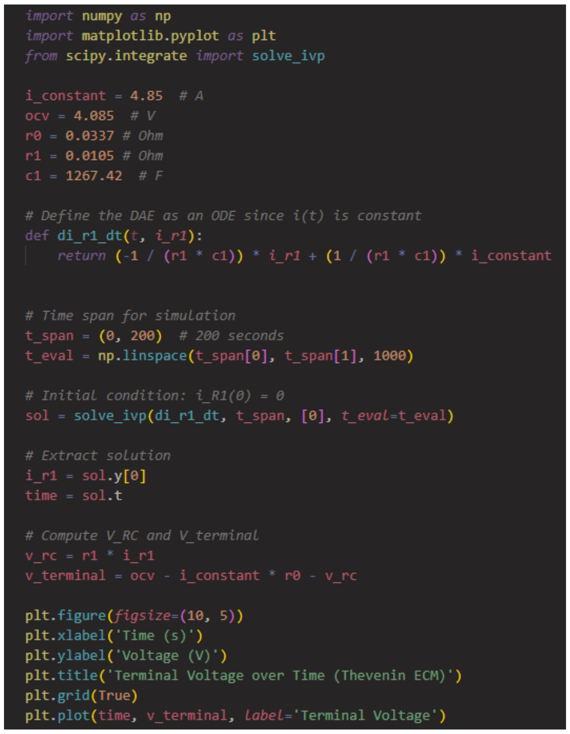
I will be mathematically proofing the DAE equation.

NOTE: This is for demo purposes. In a complete ECM implementation, Voltage, Current and the RC values would be a function of State of Charge (SoC), as well as an Open Circuit Voltage–State of Charge (OCV-SoC) curve derived from a prior capacity test.

Fig C.5: 1RC Thevenin ECM Governing Equation

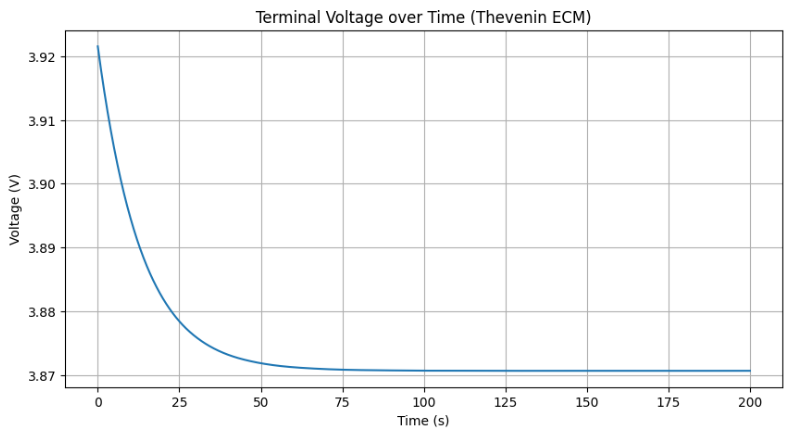
$$\frac{di_{R_1}(t)}{dt} = -\frac{1}{R_1C_1} i_{R_1}(t) + \frac{1}{R_1C_1} i(t)$$

Fig C.6: Python proofing of 1RC Thevenin ECM Governing Equation



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

i_constant = 4.85  # A
ocv = 4.085  # V
r0 = 0.0337 # Ohm
r1 = 0.0105 # Ohm
c1 = 1267.42  # F

# Define the DAE as an ODE since i(t) is constant
def di_r1_dt(t, i_r1):
    return (-1 / (r1 * c1)) * i_r1 + (1 / (r1 * c1)) * i_constant


# Time span for simulation
t_span = (0, 200)  # 200 seconds
t_eval = np.linspace(t_span[0], t_span[1], 1000)

# Initial condition: i_R1(0) = 0
sol = solve_ivp(di_r1_dt, t_span, [0], t_eval=t_eval)

# Extract solution
i_r1 = sol.y[0]
time = sol.t

# Compute V_RC and V_terminal
v_rc = r1 * i_r1
v_terminal = ocv - i_constant * r0 - v_rc

plt.figure(figsize=(10, 5))
plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.title('Terminal Voltage over Time (Thevenin ECM)')
plt.grid(True)
plt.plot(time, v_terminal, label='Terminal Voltage')
```

Fig C.7: Plotting response of 1RC Thevenin ECM Governing Equation



This simple model highlights the non-linear characteristic voltage exhibits during the pulse.

9.1.5 PyBaMM (Python Battery Mathematical Modelling)

PyBaMM (Python Battery Mathematical Modelling) is an open-source python framework spun out of Oxford University; it's used to simulate the behaviour of batteries through solving systems of differential equations [4]. PyBaMM itself, is a self-contained framework that contains of several battery models and their respective parameter sets.

I am using PyBaMM to solve for the Thevenin Equivalent Circuit Model (ECM).

The typical workflow of PyBaMM follows:

- Define a model (e.g. ECM Thevenin Model)
- Define the models' parameter values (e.g. RC pairs, OCV, Current)
- Define the Differential Algebraic Solver (DAE) (e.g. the solver tolerances)
- Define the Experiment (e.g. "*Discharge at 3A until 2.5V, rest for 1 hours, Charge at 5A for 2 hours or until 4.2V*")
    - PyBaMM has a text definable simulation mechanism that allows users to verbally define the semantics of the experiment that will then be applied to the solver. The framework also offers a time-based simulation like 1-hour, 2-hour etc…
- The solver will then compute the Ordinary Differential Equate (ODE) and solve for the simulation results

9.2 PyBop (Python Battery Optimization and Parameterization)

PyBop (Python Battery Optimization and Parameterization) is a library that offers tools for parameterization and optimization of battery models built on top of PyBaMM. I will be using PyBop to perform the parameter identification process of extracting parameters from the battery tests and fitting them onto the Thevenin Equivalent Circuit Model built with PyBaMM.

9.2.1 Parameter Identification

Parameter Identification or "parameterization" refers to the process of determining the values of a given models' parameter values such that the models output closely match the expected behavior of real experimental data.

In the context of the Thevenin ECM, this involves determining the values of the following [3] parameters:

- **R0**: Internal resistance, representing instantaneous voltage drop. Covered in section *8.1.2*
- **R1, C2**: First RC pair that captures the mid-range dynamic behavior. Covered in section *8.1.3*
- **R2, C2**: An optional RC pair in the ECM that captures higher range transient effects in the behavior for higher fidelity models.

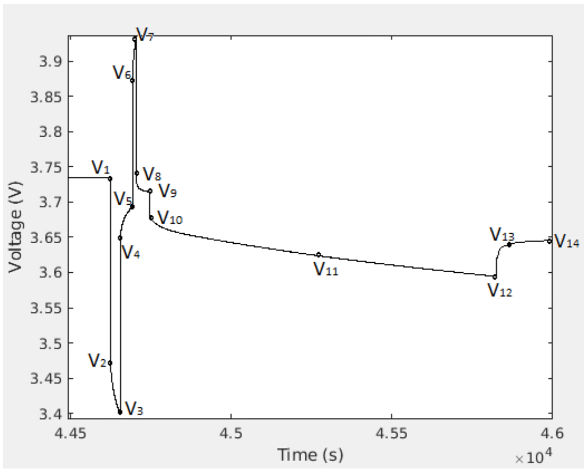
Section *8.1.4* covers the mathematics of how the Thevenin *ECM* can be expressed as a Ordinary Differential Equation (ODE) to be solved.

The goal of parameter identification is to find values of *R0, R1, C1* that minimize the error between the measured voltage from the experimental test data and the model-predicted voltage.

If you recall to section *7.1.4 "Hybrid Pulse Power Characterization (HPPC)"*, HPPC is a battery test that provides exactly the kind of input-output data needed for accurate parameter identification. The test consists of applying a sequence of current pulses at various states of charge (e.g., from 100% to 90%), followed by a rest period.

Below is an image of a singular pulse form a series of pulses from a HPPC test [3]

Fig C.8: Singular Pulse extracted from a HPPC Test. Referenced from [3]



Each pulse of the HPPC sequence offers insight into battery behaviour:

- **R0**: The instantaneous voltage drop response.
- **R1/C1**: This is the voltage relaxation curve during the rest period, representing the slower dynamic response.

9.2.2 Parameter Optimization

Once the model structure has been defined and the experimental data collected, the next step is **parameter optimization**. This involves adjusting the model parameters, in the case of Thevenin Equivalent Circuit Model, it means adjusting the RC values, so that the simulated ECM output closely matches the measured data.

In PyBop, this process is formulated as a **mathematical optimization problem**, where the user defines a cost-function with the objective is to minimize the error between the model-predicted voltage and the experimentally observed voltage over time.

The cost function includes setting up:

- **Parameter Bounds**: User defined limits for each parameter (e.g. Resistor, Capacitor) to ensure the optimization remains within realistic and valid ranges.
- **Gaussian Noise**: used to simulate measurement uncertainty within the optimization process.

The user defines this optimization problem in PyBOP, applies the necessary constraints, and selects an optimization algorithm. In this project, I used **Particle Swarm Optimization (PSO)**.

9.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based technique, much like genetic algorithms. It operates by having a group of individuals, known as “particles”, these particles explore a search space in discrete steps. [8]

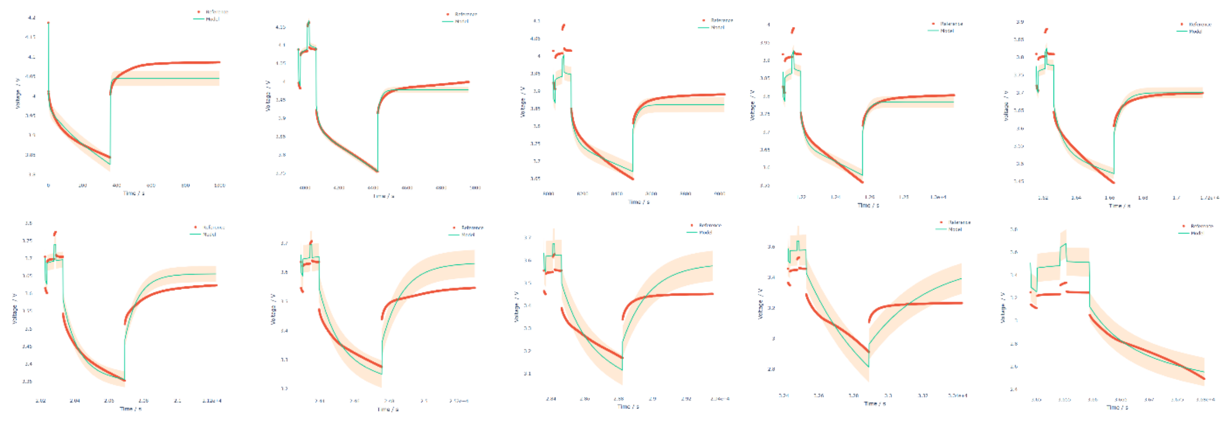
At each step, the algorithm evaluates the Cost Function (in this case, the voltage fitting error) for every particle. Each particle then updates its position and velocity by the best-known position computed. This allows the swarm of particles to collective converge towards an ideal solution over time.

Due to its computational efficiency and the recommendations from my industry supervisors.

Before adopting all this work into its own condensed codebase and library, I prototyped the optimization process in a jupyter notebook [9]

9.2.4 Optimization Results

Fig C.9: Optimized Results, Voltage Model-Reference fitting of pulses from the G1 battery



Not all voltage convergences fit well due to the same cost function being applied to each pulse.

10 ECM Parameterization + Optimization Codebase

In this section, I cover how I used PyBop to create a custom tailored made library for parameter identification and optimization of the Resistor Capacitor components.

10.1 Jupyter Notebook Rapid Prototyping.

Before developing the main codebase, I used a local jupyter notebook in Vs-Code to protype

- HPPC and Capacity Test data analysis
- Capacity Test polynomial fitting logic
- HPPC pulse identification and extraction
- ECM Parameter Optimization with PyBop

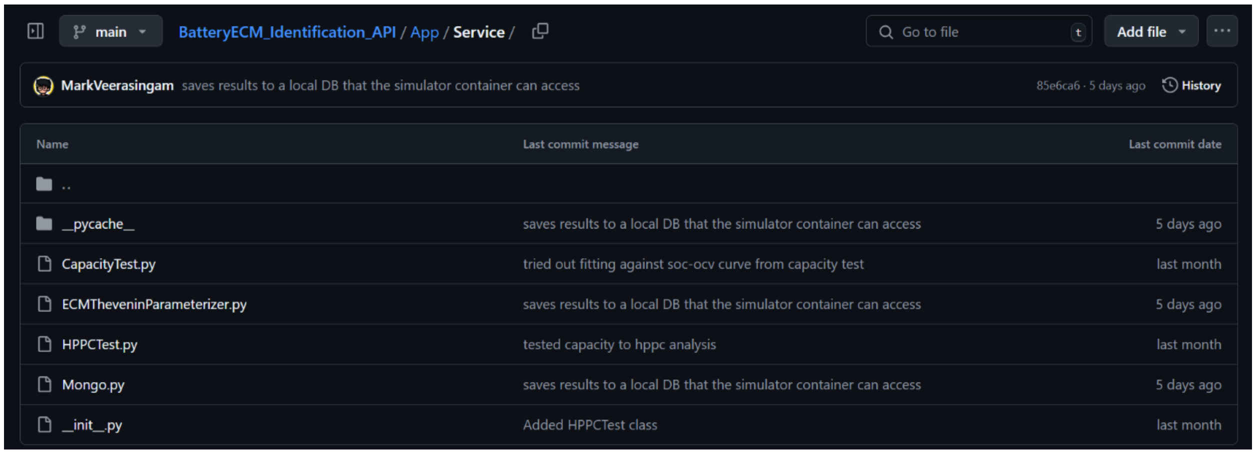
I did all this, to get a better understanding with the data and domain knowledge needed with battery modelling before adopting it into its own codebase. [7]

10.2 Custom Automated Parameterization Library

The main codebase I made, functions as a custom math library specific for ECM parameter Identification built with PyBop [9].

This is a screenshot from my public GitHub repository of the library.

Fig D.1: ECM Parameterization Codebase File structure



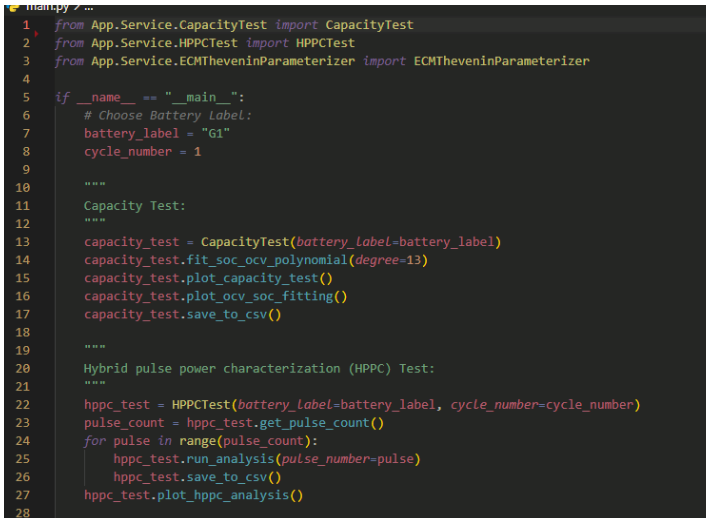
MarkVeerasingam saves results to a local DB that the simulator container can access 85e6ca6 · 5 days ago

Name	Last commit message	Last commit date
..		
__pycache__	saves results to a local DB that the simulator container can access	5 days ago
CapacityTest.py	tried out fitting against soc-ocv curve from capacity test	last month
ECMTheveninParameterizer.py	saves results to a local DB that the simulator container can access	5 days ago
HPPCTest.py	tested capacity to hppc analysis	last month
Mongo.py	saves results to a local DB that the simulator container can access	5 days ago
__init__.py	Added HPPCTest class	last month

- **CapacityTest.py**
  - handles the logic for extracting the SoC-OCV curve of each cycle for a given battery label and fit the polynomial.
- **HPPCTest.py**
  - Handles the logic for extracting all pulses for a given cycle of a battery label from its HPPC test and saving the extracted pulses locally in the repository.

At an abstracted level in Main.py, the library looks like this…

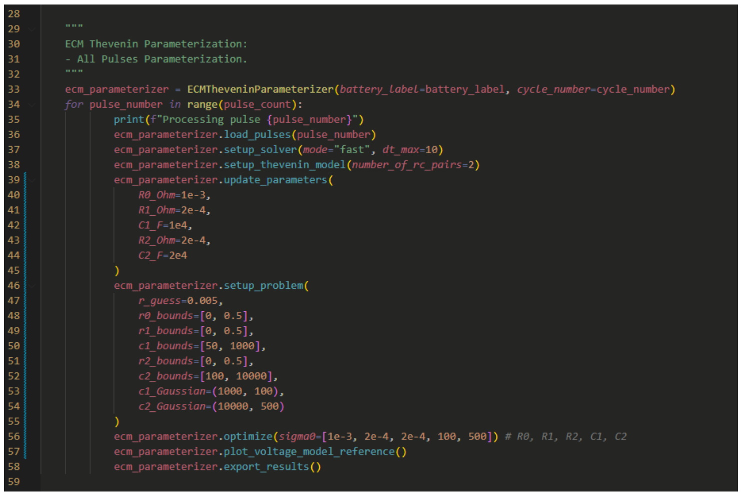
Fig D.2: ECM Parameterization codebase – Capacity Test + HPPC Test analysis



```python
from App.Service.CapacityTest import CapacityTest
from App.Service.HPPCTest import HPPCTest
from App.Service.ECMTheveninParameterizer import ECMTheveninParameterizer

if __name__ == "__main__":
    # Choose Battery Label:
    battery_label = "G1"
    cycle_number = 1

    """
    Capacity Test:
    """
    capacity_test = CapacityTest(battery_label=battery_label)
    capacity_test.fit_soc_ocv_polynomial(degree=13)
    capacity_test.plot_capacity_test()
    capacity_test.plot_ocv_soc_fitting()
    capacity_test.save_to_csv()

    """
    Hybrid pulse power characterization (HPPC) Test:
    """
    hppc_test = HPPCTest(battery_label=battery_label, cycle_number=cycle_number)
    pulse_count = hppc_test.get_pulse_count()
    for pulse in range(pulse_count):
        hppc_test.run_analysis(pulse_number=pulse)
        hppc_test.save_to_csv()
    hppc_test.plot_hppc_analysis()

```

- **ECMTheveninParameterizer.py**
    - This is the Thevenin ECM Parameter Optimization logic, with the goal to optimize the RC values of the Thevenin ECM to match the expected voltage of the experimental HPPC data.

Fig D.3: ECM Parameterization codebase – running parameterization



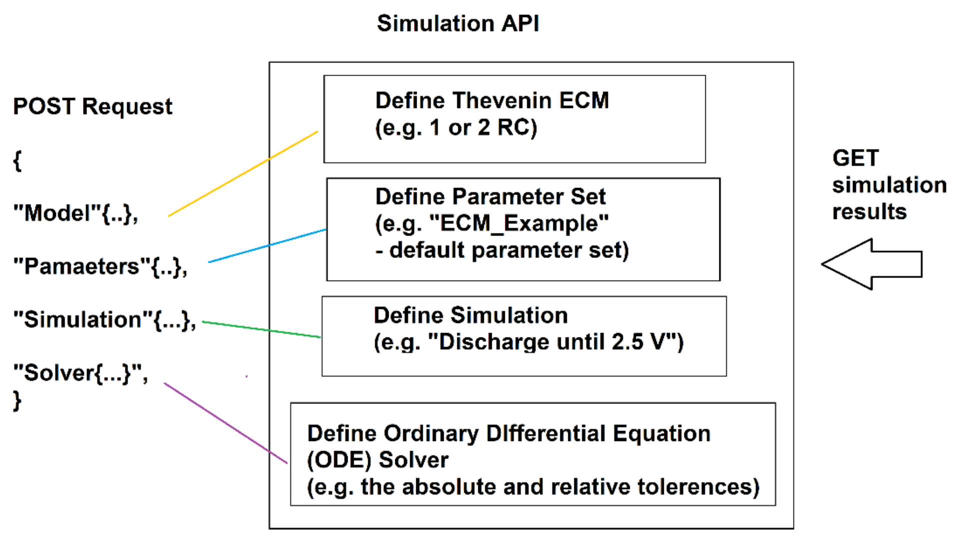
```python

    """
    ECM Thevenin Parameterization:
    - All Pulses Parameterization.
    """
    ecm_parameterizer = ECMTheveninParameterizer(battery_label=battery_label, cycle_number=cycle_number)
    for pulse_number in range(pulse_count):
            print(f"Processing pulse {pulse_number}")
            ecm_parameterizer.load_pulses(pulse_number)
            ecm_parameterizer.setup_solver(mode="fast", dt_max=10)
            ecm_parameterizer.setup_thevenin_model(number_of_rc_pairs=2)
            ecm_parameterizer.update_parameters(
                R0_Ohm=1e-3,
                R1_Ohm=2e-4,
                C1_F=1e4,
                R2_Ohm=2e-4,
                C2_F=2e4
            )
            ecm_parameterizer.setup_problem(
                r_guess=0.005,
                r0_bounds=[0, 0.5],
                r1_bounds=[0, 0.5],
                c1_bounds=[50, 1000],
                r2_bounds=[0, 0.5],
                c2_bounds=[100, 10000],
                c1_Gaussian=(1000, 100),
                c2_Gaussian=(10000, 500)
            )
            ecm_parameterizer.optimize(sigma0=[1e-3, 2e-4, 2e-4, 100, 500]) # R0, R1, R2, C1, C2
            ecm_parameterizer.plot_voltage_model_reference()
            ecm_parameterizer.export_results()

```

11 Battery Simulation API Codebase

The approach I took with PyBaMM, is that it will be the underlying simulation engine for my simulation API. Where in Object Oriented Programming (OOP) style, I have abstracted away it's 4 fundamental prerequisites to perform a simulation, being the definition of a battery model for example for my project, it will only every be an ECM model, abstracting the models parameters into its own class, abstracting the differential equation solver into its own class, and finally abstracting the user defined simulation definition into its own class.

11.1 Simulation API Structure

Fig E: Simulation API Architecture Overview to Simulate an ECM


Simulation API
POST Request
{
"Model"{..},
"Pamaeters"{..},
"Simulation"{...},
"Solver{...}",
}
Define Thevenin ECM (e.g. 1 or 2 RC)
Define Parameter Set (e.g. "ECM_Example" - default parameter set)
Define Simulation (e.g. "Discharge until 2.5 V")
Define Ordinary DIfferential Equation (ODE) Solver (e.g. the absolute and relative tolerences)
GET simulation results

The diagram above shows the core logic and structure of my Simulation API.

This section covers the main structure, logic and approach to the Simulation API.

Built on top of PyBaMM, the API follows an Object-Oriented Programming (OOP) approach, where the following features required to run a PyBaMM simulation are broken into classes…

- Modelling
- Parameters
- Simulation Conditions
- Solver

For example, the user can define an Equivalent Circuit Model of a 1 or 2 RC pair, manually update specified parameters e.g. the RC pair values, current or temperature.

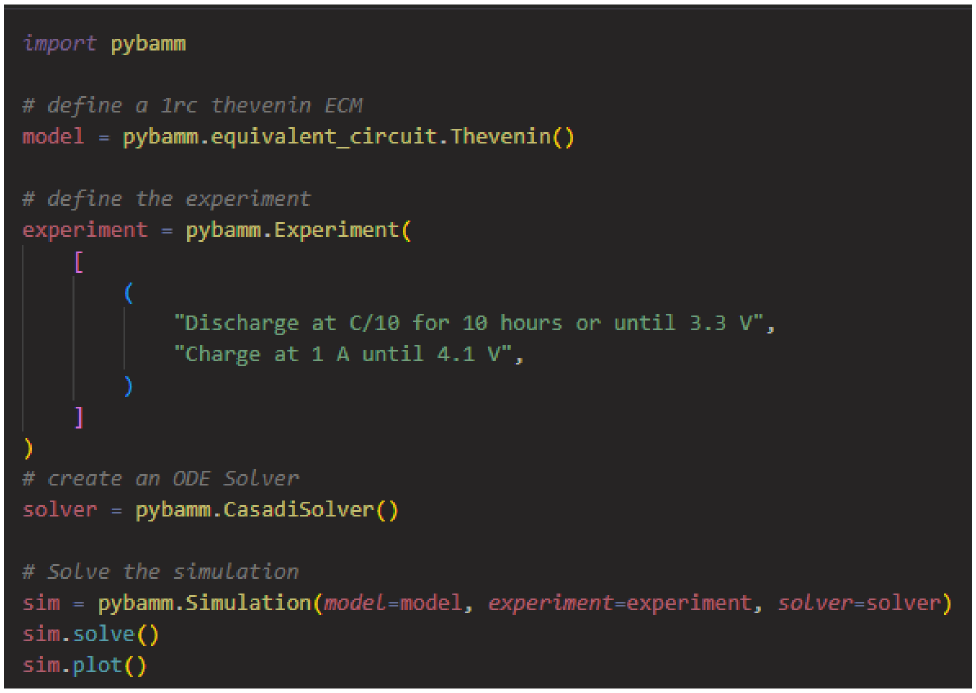
I have covered the functionality of each of the main components needed for PyBaMM and battery simulations to operate in the previous section *8.1 – 8.1.5*.

11.2 Object-Oriented Programming (OOP) approach to Simulations

The reason I chose am OOP approach for this was because of the straightforward nature of running a simulation with PyBaMM. I noticed that by structuring the API in an OOP style, a lot of the logic could become reusable and offer a more efficient design for modularity in the event of declaring a more user-definable API.

Below is an example of running a simulation in PyBaMM

Fig E.2: Generic Simulation with PyBaMM



```python
import pybamm

# define a 1rc thevenin ECM
model = pybamm.equivalent_circuit.Thevenin()

# define the experiment
experiment = pybamm.Experiment(
    [
        (
            "Discharge at C/10 for 10 hours or until 3.3 V",
            "Charge at 1 A until 4.1 V",
        )
    ]
)
# create an ODE Solver
solver = pybamm.CasadiSolver()

# Solve the simulation
sim = pybamm.Simulation(model=model, experiment=experiment, solver=solver)
sim.solve()
sim.plot()
```

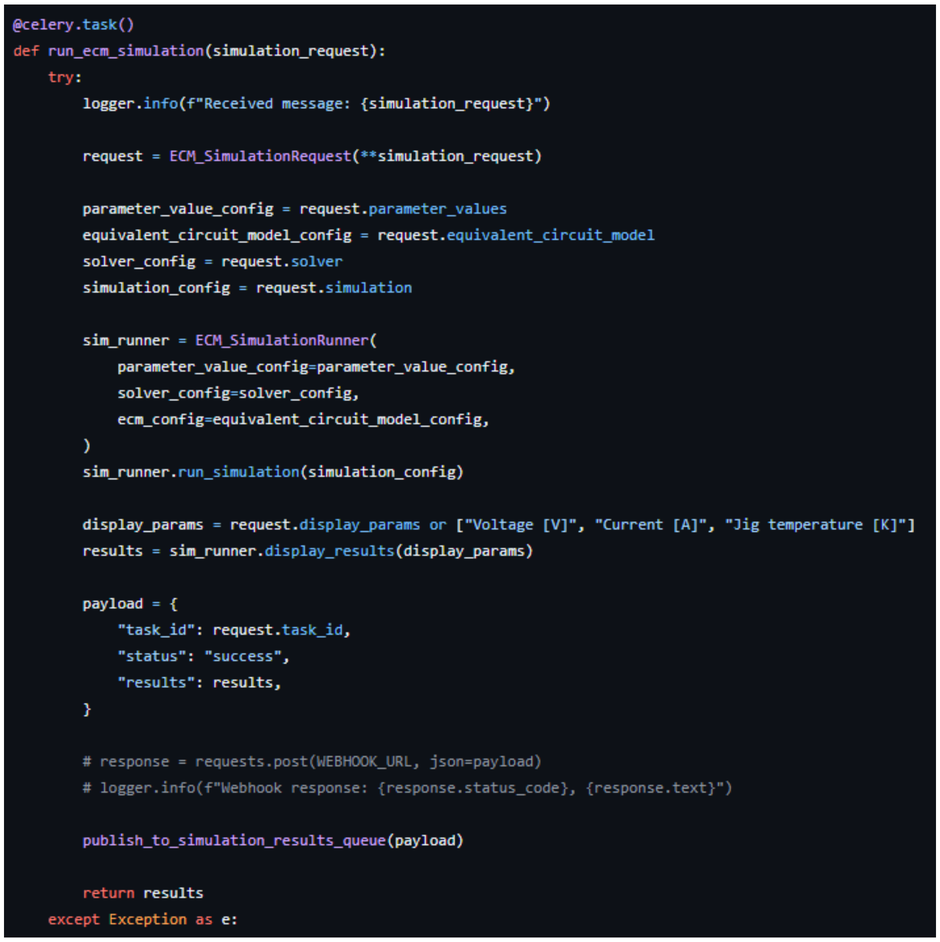
PyBaMM offers various solvers, experiment techniques such as solving for time or interpolating against a “"drive-cycle" (i.e., simulating battery behavior under realistic electric vehicle loads), and selection of other models beyond the equivalent circuit model.

Before deciding that in my project I wanted to construct battery models from experimental data, rather than simply running simulations using predefined models.

My project’s initial goal was to develop a generic Battery Simulator supporting physics-based models and ECM models. These considerations played a significant role in adopting an OOP approach.

Below is a photo of how I handle the core ECM simulation logic in my API. It takes the data from the post request and feeds it to its respective processing.

Fig E.3: Post Request task to simulate an ECM

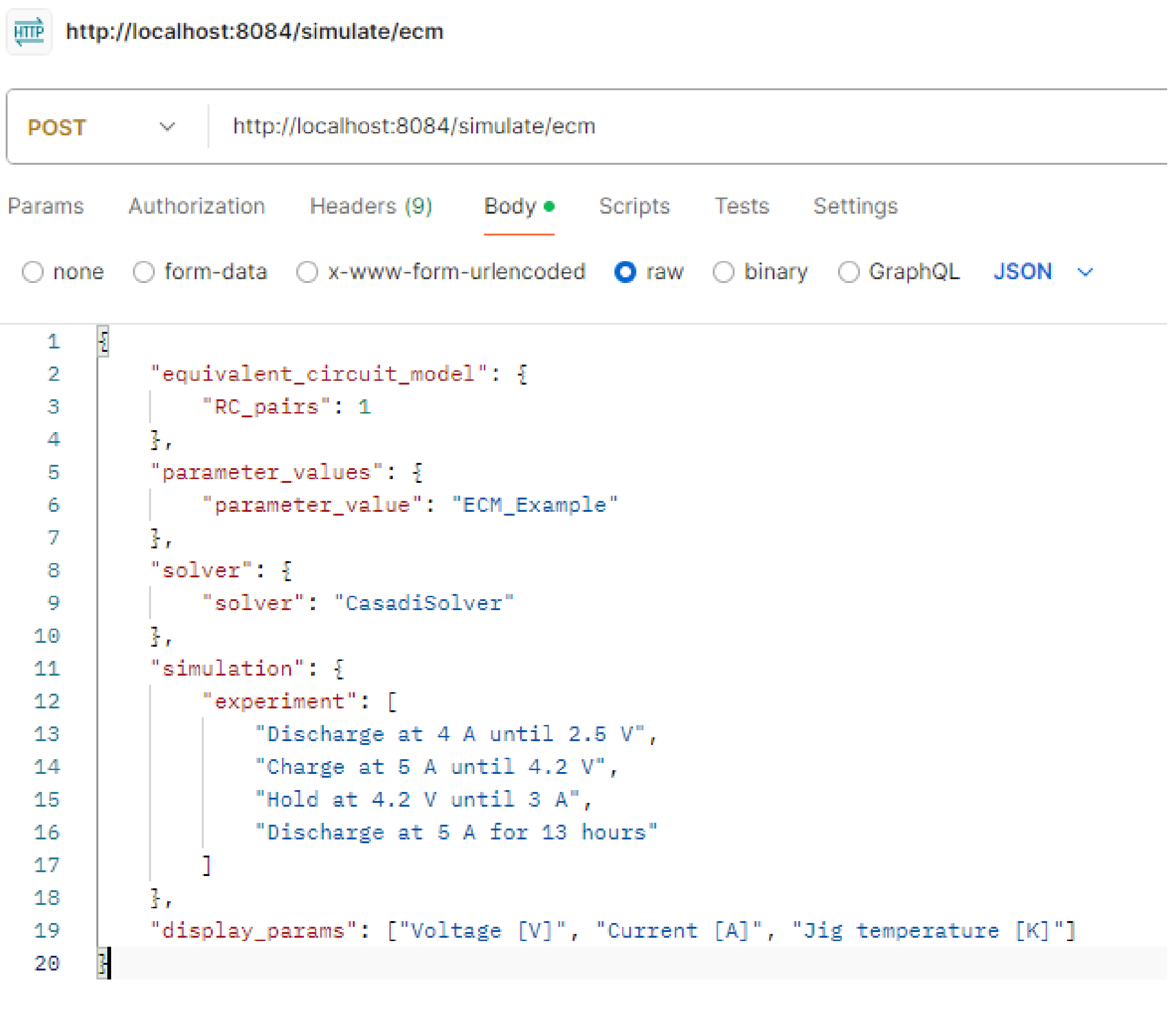


```python
@celery.task()
def run_ecm_simulation(simulation_request):
    try:
        logger.info(f"Received message: {simulation_request}")

        request = ECM_SimulationRequest(**simulation_request)

        parameter_value_config = request.parameter_values
        equivalent_circuit_model_config = request.equivalent_circuit_model
        solver_config = request.solver
        simulation_config = request.simulation

        sim_runner = ECM_SimulationRunner(
            parameter_value_config=parameter_value_config,
            solver_config=solver_config,
            ecm_config=equivalent_circuit_model_config,
        )
        sim_runner.run_simulation(simulation_config)

        display_params = request.display_params or ["Voltage [V]", "Current [A]", "Jig temperature [K]"]
        results = sim_runner.display_results(display_params)

        payload = {
            "task_id": request.task_id,
            "status": "success",
            "results": results,
        }

        # response = requests.post(WEBHOOK_URL, json=payload)
        # logger.info(f"Webhook response: {response.status_code}, {response.text}")

        publish_to_simulation_results_queue(payload)

        return results
    except Exception as e:
```

11.3 Generic ECM Simulation

Running this type of experiment would use a default parameter set to pybamm called “*ECM_Example*” this parameter contains the RC, SoC, temperature and OCV values of a battery model.

Fig E.4: Generic Battery ECM Simulation with API using pybamm default ECM parameters.



http://localhost:8084/simulate/ecm

POST http://localhost:8084/simulate/ecm

Body: [✓ raw] JSON

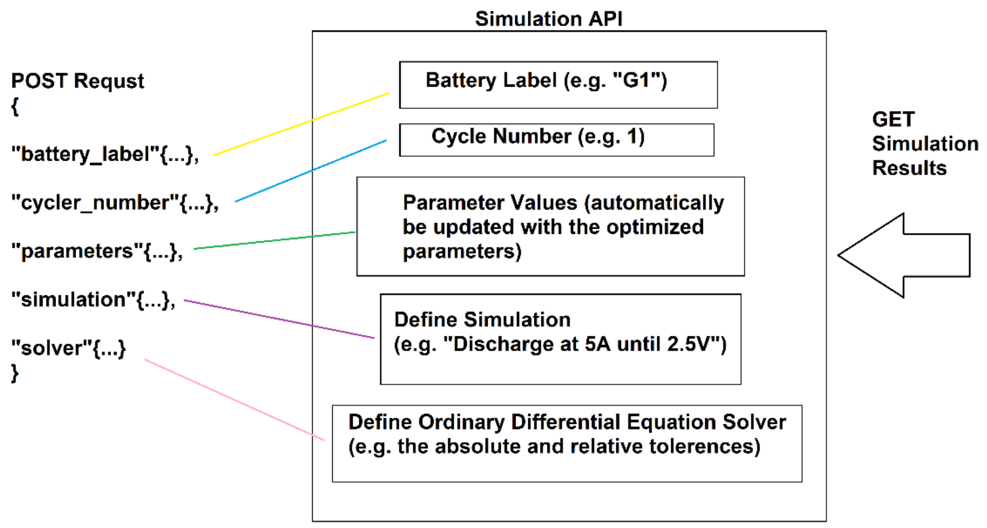
```
{
    "equivalent_circuit_model": {
        "RC_pairs": 1
    },
    "parameter_values": {
        "parameter_value": "ECM_Example"
    },
    "solver": {
        "solver": "CasadiSolver"
    },
    "simulation": {
        "experiment": [
            "Discharge at 4 A until 2.5 V",
            "Charge at 5 A until 4.2 V",
            "Hold at 4.2 V until 3 A",
            "Discharge at 5 A for 13 hours"
        ]
    },
    "display_params": ["Voltage [V]", "Current [A]", "Jig temperature [K]"]
}
```

We don’t know what battery model this parameter set takes from, this is why tailored models is as important in battery development as it is safety of battery management systems.

My project sets out to create a custom parameter set from the experimental dataset, tailored to the test data of the battery.

11.4 Parameterized ECM Simulation

Fig E.5: Simulation API Architecture Overview to simulate a parameterized-ecm


Simulation API
POST Requst
{
"battery_label"{...},
"cycler_number"{...},
"parameters"{...},
"simulation"{...},
"solver"{...}
}
Battery Label (e.g. "G1")
Cycle Number (e.g. 1)
Parameter Values (automatically be updated with the optimized parameters)
Define Simulation (e.g. "Discharge at 5A until 2.5V")
Define Ordinary Differential Equation Solver (e.g. the absolute and relative tolerences)
GET Simulation Results

11.5 Interpolating Parameters

When wanting to simulate the ECM with the optimized parameters we received from our tests, we need to interpolate them, the RC values and the SoC-OCV curve.

The ECM is dependent on the SoC meaning…

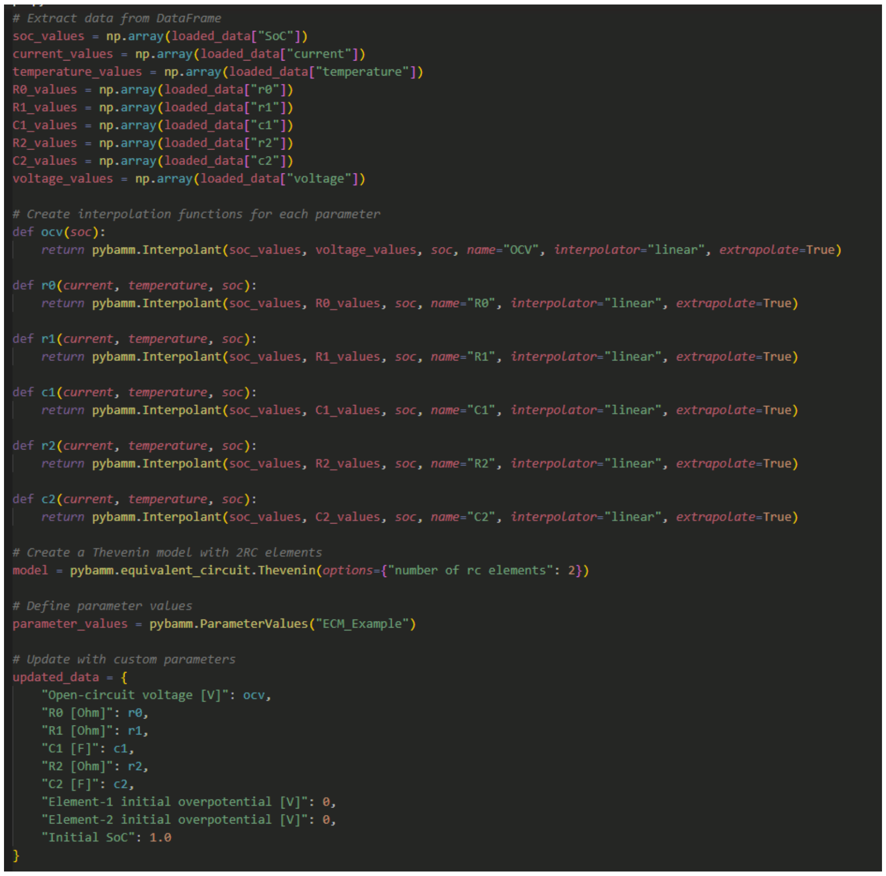
- OCV values
- RC values

Are all dependent on the batteries state of charge.

So, if the battery voltage is **3.9 V** (about **90% SoC**), we need to **interpolate** to find what the **R** and **C** values are at 90%.

The below image is an example of how I use PyBaMM to interpolate from data read in optimized parameter values saved from MongoDB

Fig E.6: Interpolating Parameters in PyBaMM



```python
# Extract data from DataFrame
soc_values = np.array(loaded_data["SoC"])
current_values = np.array(loaded_data["current"])
temperature_values = np.array(loaded_data["temperature"])
R0_values = np.array(loaded_data["r0"])
R1_values = np.array(loaded_data["r1"])
C1_values = np.array(loaded_data["c1"])
R2_values = np.array(loaded_data["r2"])
C2_values = np.array(loaded_data["c2"])
voltage_values = np.array(loaded_data["voltage"])

# Create interpolation functions for each parameter
def ocv(soc):
    return pybamm.Interpolant(soc_values, voltage_values, soc, name="OCV", interpolator="linear", extrapolate=True)

def r0(current, temperature, soc):
    return pybamm.Interpolant(soc_values, R0_values, soc, name="R0", interpolator="linear", extrapolate=True)

def r1(current, temperature, soc):
    return pybamm.Interpolant(soc_values, R1_values, soc, name="R1", interpolator="linear", extrapolate=True)

def c1(current, temperature, soc):
    return pybamm.Interpolant(soc_values, C1_values, soc, name="C1", interpolator="linear", extrapolate=True)

def r2(current, temperature, soc):
    return pybamm.Interpolant(soc_values, R2_values, soc, name="R2", interpolator="linear", extrapolate=True)

def c2(current, temperature, soc):
    return pybamm.Interpolant(soc_values, C2_values, soc, name="C2", interpolator="linear", extrapolate=True)

# Create a Thevenin model with 2RC elements
model = pybamm.equivalent_circuit.Thevenin(options={"number of rc elements": 2})

# Define parameter values
parameter_values = pybamm.ParameterValues("ECM_Example")

# Update with custom parameters
updated_data = {
    "Open-circuit voltage [V]": ocv,
    "R0 [Ohm]": r0,
    "R1 [Ohm]": r1,
    "C1 [F]": c1,
    "R2 [Ohm]": r2,
    "C2 [F]": c2,
    "Element-1 initial overpotential [V]": 0,
    "Element-2 initial overpotential [V]": 0,
    "Initial SoC": 1.0
}
```

11.6 Fast API

I used FastAPI, an API framework to access the simulation engine. This is aimed to be the primary way to simulate a battery.

11.7 Celery

Celery is an asynchronous task queue package used for running tasks asynchronously from a broker. I am using celery in my application to run multiple simulations simultaneously.

11.8 Redis

Redis is an in-memory, cache-based data store, compared to traditional disk data retrieval. I am using Redis as both a broker and result backend for Celery, allowing simulations to run asynchronously. The goal was to say if I ran multiple simulations at once, I would not be in line of a First in First out order.

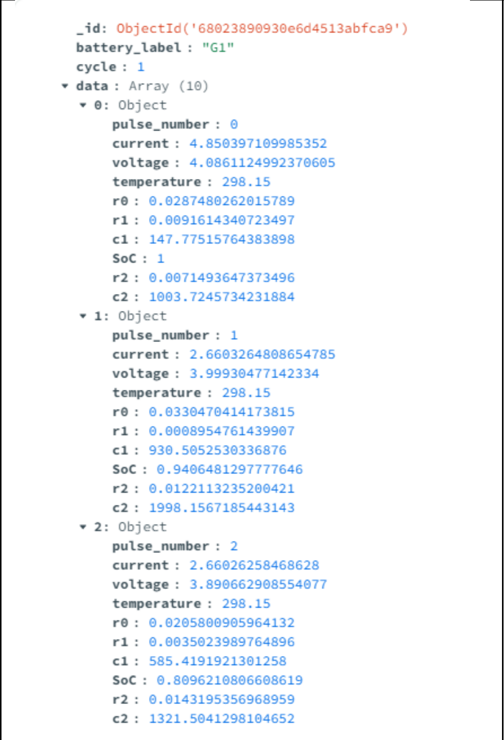
12 MongoDB

MongoDB is used as the primary database to store the ECM optimisation results. Throughout this report I have been using the battery experiment “G1” and chosen to parameterize cycle 1 as seen in *section 8.2.4* and from the math example in *8.1.4* where I expressed an ECM as a differential equation.

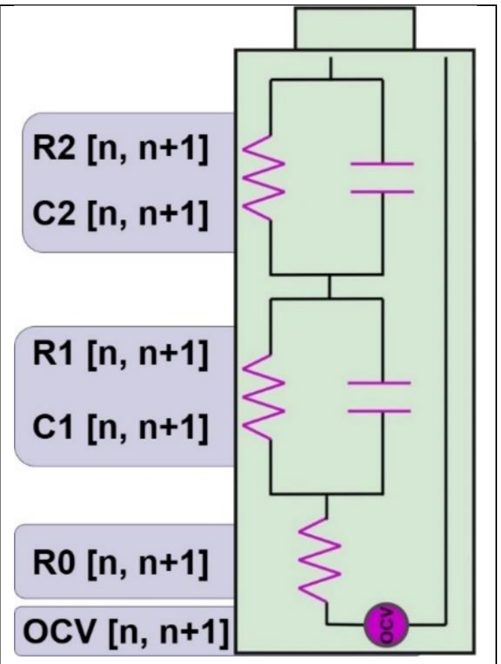
G1, Cycle 1 had 10 pulses, in MongoDB we save these 10 pulses as noted in the nested “data” column of size 10.

It was for this reason that I chose MongoDB, a noSQL database to hold the amount of complex structure of data I would be required to save.

Fig E.7: G1, Cycle-1, 10 parameterized pulse results in MongoDB with visual aid



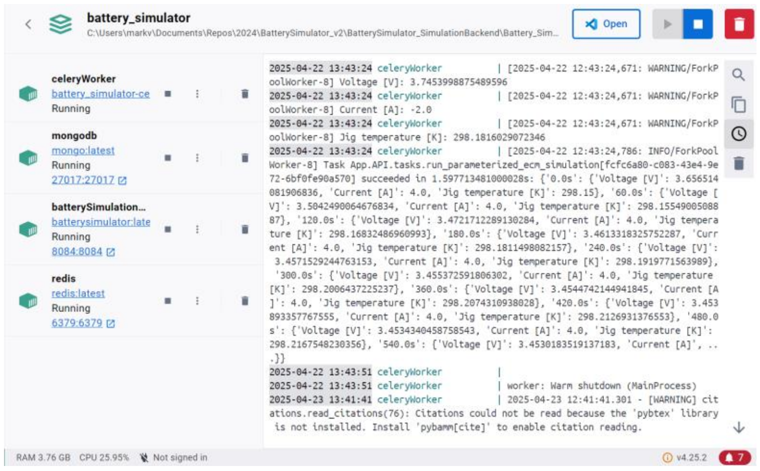
```
    _id: ObjectId('68023890930e6d4513abfca9')
    battery_label : "G1"
    cycle : 1
  ▾ data : Array (10)
    ▾ 0: Object
        pulse_number : 0
        current : 4.850397109985352
        voltage : 4.0861124992370605
        temperature : 298.15
        r0 : 0.0287480262015789
        r1 : 0.0091614340723497
        c1 : 147.77515764383898
        SoC : 1
        r2 : 0.0071493647373496
        c2 : 1003.7245734231884
    ▾ 1: Object
        pulse_number : 1
        current : 2.6603264808654785
        voltage : 3.99930477142334
        temperature : 298.15
        r0 : 0.0330470414173815
        r1 : 0.0008954761439907
        c1 : 930.5052530336876
        SoC : 0.9406481297777646
        r2 : 0.0122113235200421
        c2 : 1998.1567185443143
    ▾ 2: Object
        pulse_number : 2
        current : 2.66026258468628
        voltage : 3.890662908554077
        temperature : 298.15
        r0 : 0.0205800905964132
        r1 : 0.0035023989764896
        c1 : 585.4191921301258
        SoC : 0.8096210806608619
        r2 : 0.0143195356968959
        c2 : 1321.5041298104652
```



13 Docker

My entire project has been containerized with Docker. Below is a screenshot of my docker-compose file and docker container with all my images running.

Fig E.8: Docker container of the Battery Simulator



```
2025-04-22 13:43:24 celeryWorker        | [2025-04-22 12:43:24,671: WARNING/ForkP
oolWorker-8] Voltage [V]: 3.7453998875489596
2025-04-22 13:43:24 celeryWorker        | [2025-04-22 12:43:24,671: WARNING/ForkP
oolWorker-8] Current [A]: -2.0
2025-04-22 13:43:24 celeryWorker        | [2025-04-22 12:43:24,671: WARNING/ForkP
oolWorker-8] Jig temperature [K]: 298.1816029072346
2025-04-22 13:43:24 celeryWorker        | [2025-04-22 12:43:24,786: INFO/ForkPool
Worker-8] Task App.API.tasks.run_parameterized_ecm_simulation[fcfc6a80-c083-43e4-9e
72-6bf0fe90a570] succeeded in 1.597713481000028s: {'0.0s': {'Voltage [V]': 3.656514
081906836, 'Current [A]': 4.0, 'Jig temperature [K]': 298.15}, '60.0s': {'Voltage [
V]': 3.5042490064676834, 'Current [A]': 4.0, 'Jig temperature [K]': 298.15549005088
87}, '120.0s': {'Voltage [V]': 3.4721712289130284, 'Current [A]': 4.0, 'Jig tempera
ture [K]': 298.16832486960993}, '180.0s': {'Voltage [V]': 3.4613318325752287, 'Curr
ent [A]': 4.0, 'Jig temperature [K]': 298.1811498082157}, '240.0s': {'Voltage [V]':
 3.4571529244763153, 'Current [A]': 4.0, 'Jig temperature [K]': 298.1919771563989},
 '300.0s': {'Voltage [V]': 3.455372591806302, 'Current [A]': 4.0, 'Jig temperature
[K]': 298.2006437225237}, '360.0s': {'Voltage [V]': 3.4544742144941845, 'Current [A
]': 4.0, 'Jig temperature [K]': 298.2074310938028}, '420.0s': {'Voltage [V]': 3.453
893357767555, 'Current [A]': 4.0, 'Jig temperature [K]': 298.2126931376553}, '480.0
s': {'Voltage [V]': 3.4534340458758543, 'Current [A]': 4.0, 'Jig temperature [K]':
298.2167548230356}, '540.0s': {'Voltage [V]': 3.4530183519137183, 'Current [A]', ..
.}}
2025-04-22 13:43:51 celeryWorker        |
2025-04-22 13:43:51 celeryWorker        | worker: Warm shutdown (MainProcess)
2025-04-23 13:41:41 celeryWorker        | 2025-04-23 12:41:41.301 - [WARNING] cit
ations.read_citations(76): Citations could not be read because the 'pybtex' library
 is not installed. Install 'pybamm[cite]' to enable citation reading.
```

14 Simulation Results + Summary

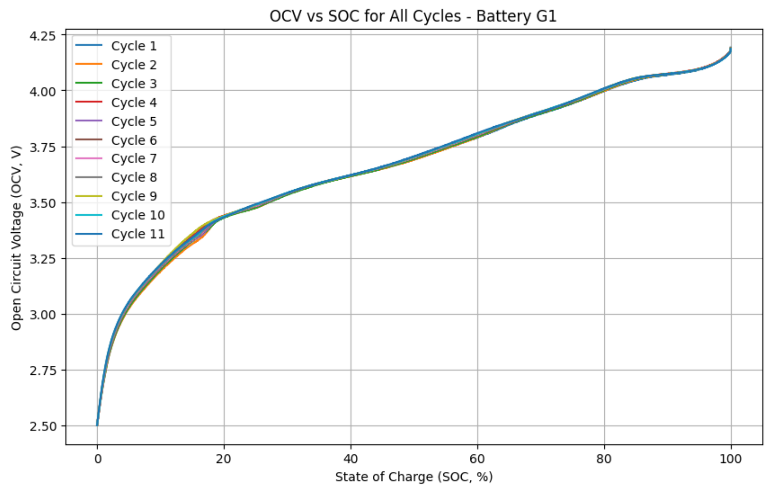
In this section, I will present the results from the G1 battery experiment, Cycle 1, as part of a broader set of experiments, from the Stanford University Dataset.

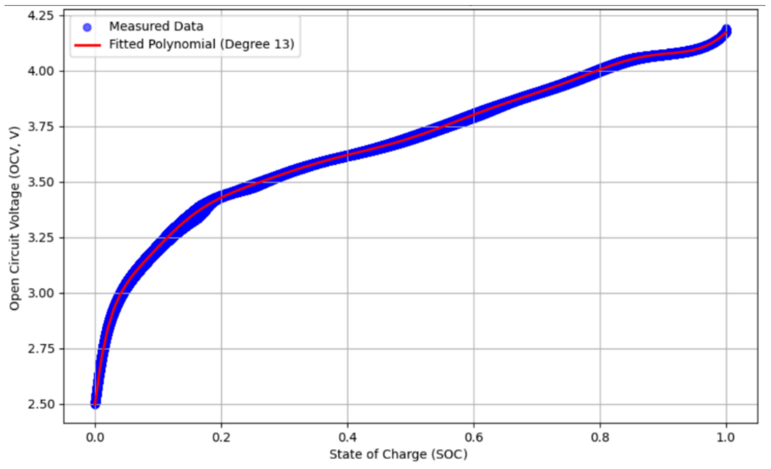
This will demonstrate how the Battery Simulator can construct full LGM50 battery model, from test data. Showing how a user can access the simulator via API endpoints highlighting an end-end workflow.

This summary will encapsulate the key findings and methods discussed throughout the report, providing a cohesive overview of the Battery Simulator.

- **Analysing the Capacity Test.**
  - The goal was to capture the SoC-OCV curve for the G1 battery. This is to capture the generalist behaviour of how the voltage response of the battery would be as the battery ages.

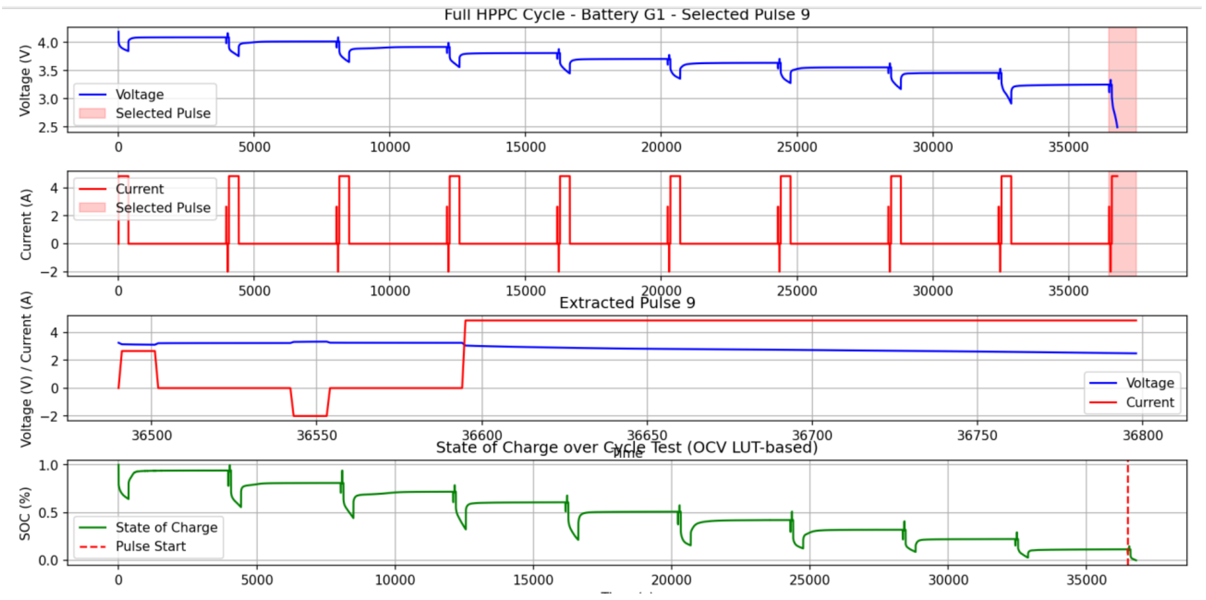
        - This involved fitting a polynomial across all the cycles to smoothen the voltage response to change to capture a generalist SoC-OCV curve.
    - *Section 9.2 - "Custom Automated Parameterization Library"* shows the code process in how I achieve this.
    - The results can be found in section *7.1.2 - "Capacity Test"* and *7.1.3 - "Capacity Test – Polynomial fitting"*. Below are the same results from that section.





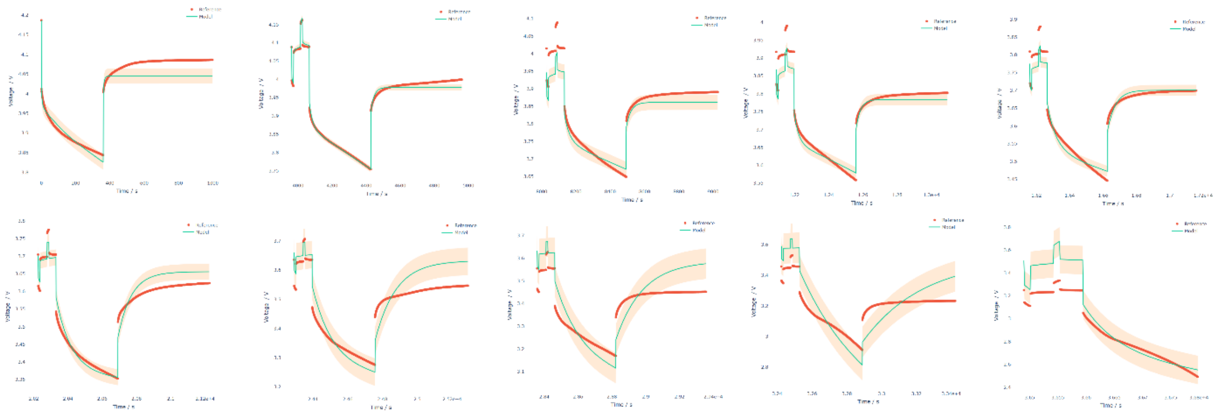
- **Hybrid Pulse Power Characterization (HPPC) Test Analysis.**
    - Covered in *section 7.1.4* the goal was to isolate the pulse that captured the non-linear voltage characteristics of the battery at different charge intervals, when a current stimulus is applied.
    - We use HPPC data for parametrization to gain insight into how the battery model ECM, RC pair values should be at given charge intervals.

- *Section 9.2 - "Custom Automated Parameterization Library"* shows the code process in how I achieve this.
- Below are the results from the same section



  - Each extracted pulse is saved locally to the repository to *"Data/Output/HPPC_Test/G1"*

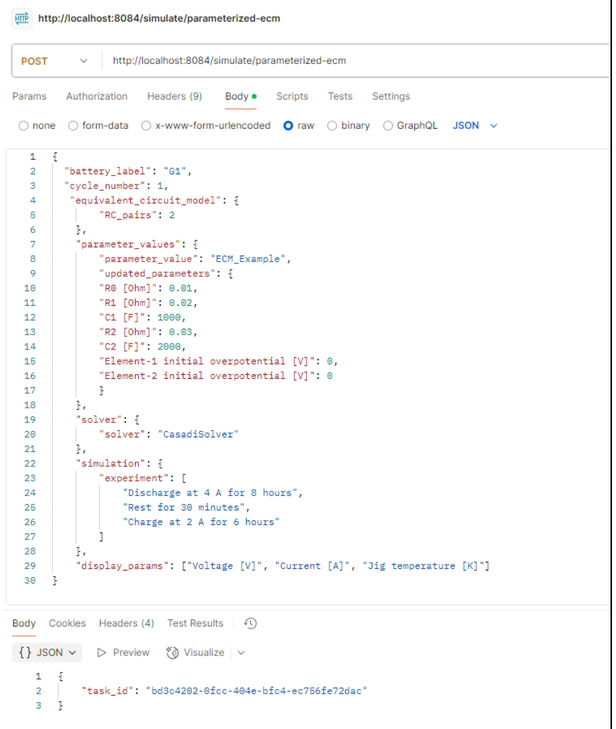
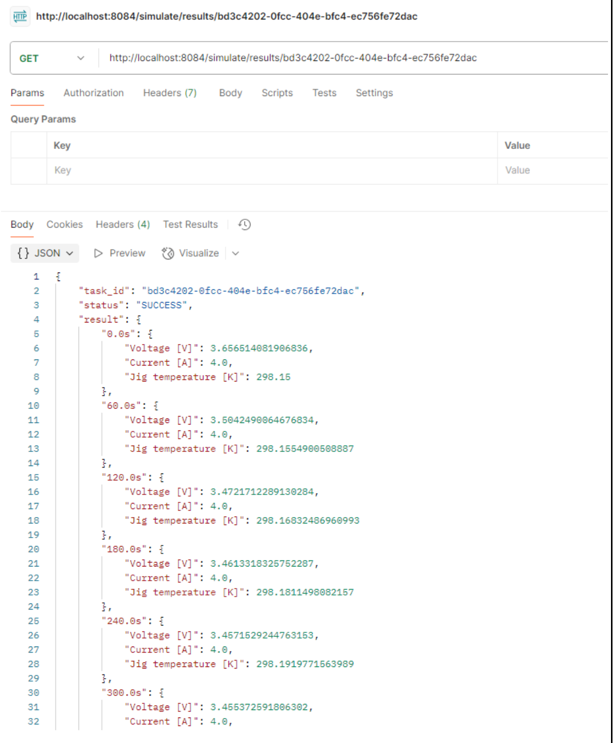
- **Parameter Identification and Optimization.**
  - Next is to find the RC pair values of the ECM, explained in *section 8.2.1 – "Parameter Identification" and section 8.2.2 – "Parameter Optimization".*
    - It uses the data saved locally from the repository i.e. it expects there to be in *"Data/Output/HPPC_Test/G1"*
  - Executing the code covered in Section *9.2 - "Custom Automated Parameterization Library"*.
  - Results of the 10 pulse optimized parameters are shown in *section 8.2.4 – "optimization results".*



- In my code, I have the optimization problem hard coded. This means that the RC bounds defined in the problem were the same for each pulse applied, leaving the results to vary drastically per pulse.
- The optimized RC, SoC and OCV values for pulse is saved to a collection called "*ECM_LUT*" in MongoDB, covered in *section 11 – "MongoDB".*

- **Simulating the Model.**
  - Having all the parameters needed to simulate the LGM50 battery constructed from the G1 experiment, cycle 1. We simulate the following experiment via API

```
▪   "experiment": [
▪               "Discharge at 4 A for 8 hours",
▪               "Rest for 30 minutes",
▪               "Charge at 2A for 6 hours"
▪           ]
```

Post localhost:8084/simulate/paramaterized-ecm	GET localhost:8084/simulate/results/{task_id}
	

http://localhost:8084/simulate/parameterized-ecm

POST http://localhost:8084/simulate/parameterized-ecm

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
  "battery_label": "G1",
  "cycle_number": 1,
   "equivalent_circuit_model": {
       "RC_pairs": 2
    },
    "parameter_values": {
        "parameter_value": "ECM_Example",
        "updated_parameters": {
        "R0 [Ohm]": 0.01,
        "R1 [Ohm]": 0.02,
        "C1 [F]": 1000,
        "R2 [Ohm]": 0.03,
        "C2 [F]": 2000,
        "Element-1 initial overpotential [V]": 0,
        "Element-2 initial overpotential [V]": 0
        }
    },
    "solver": {
        "solver": "CasadiSolver"
    },
    "simulation": {
        "experiment": [
            "Discharge at 4 A for 8 hours",
            "Rest for 30 minutes",
            "Charge at 2 A for 6 hours"
        ]
    },
    "display_params": ["Voltage [V]", "Current [A]", "Jig temperature [K]"]
}
```

Body Cookies Headers (4) Test Results

{} JSON Preview Visualize

```
{
    "task_id": "bd3c4202-0fcc-404e-bfc4-ec756fe72dac"
}
```

http://localhost:8084/simulate/results/bd3c4202-0fcc-404e-bfc4-ec756fe72dac

GET http://localhost:8084/simulate/results/bd3c4202-0fcc-404e-bfc4-ec756fe72dac

Params Authorization Headers (7) Body Scripts Tests Settings

Query Params

	Key	Value
	Key	Value

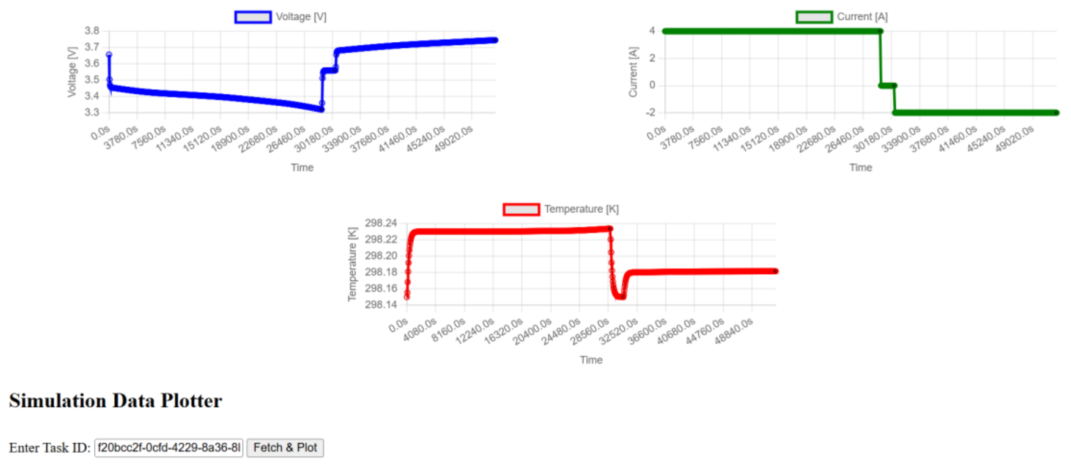
Body Cookies Headers (4) Test Results

{} JSON Preview Visualize

```
{
    "task_id": "bd3c4202-0fcc-404e-bfc4-ec756fe72dac",
    "status": "SUCCESS",
    "result": {
        "0.0s": {
            "Voltage [V]": 3.656514081906836,
            "Current [A]": 4.0,
            "Jig temperature [K]": 298.15
        },
        "60.0s": {
            "Voltage [V]": 3.5042490064676834,
            "Current [A]": 4.0,
            "Jig temperature [K]": 298.1554900508887
        },
        "120.0s": {
            "Voltage [V]": 3.4721712289130284,
            "Current [A]": 4.0,
            "Jig temperature [K]": 298.16832486960993
        },
        "180.0s": {
            "Voltage [V]": 3.4613318325752287,
            "Current [A]": 4.0,
            "Jig temperature [K]": 298.1811498082157
        },
        "240.0s": {
            "Voltage [V]": 3.4571529244763153,
            "Current [A]": 4.0,
            "Jig temperature [K]": 298.1919771563989
        },
        "300.0s": {
            "Voltage [V]": 3.455372591806302,
            "Current [A]": 4.0,
```

- **Graphed Results.**
    - Below are the graphed results plotting the voltage, current and temperature of the battery in a simple UI I wrote for demonstration purposes to compare to the PyBaMM results to note the expected graph.

Fig D: Demonstrative web UI Graphing of simulation results in PyBaMM

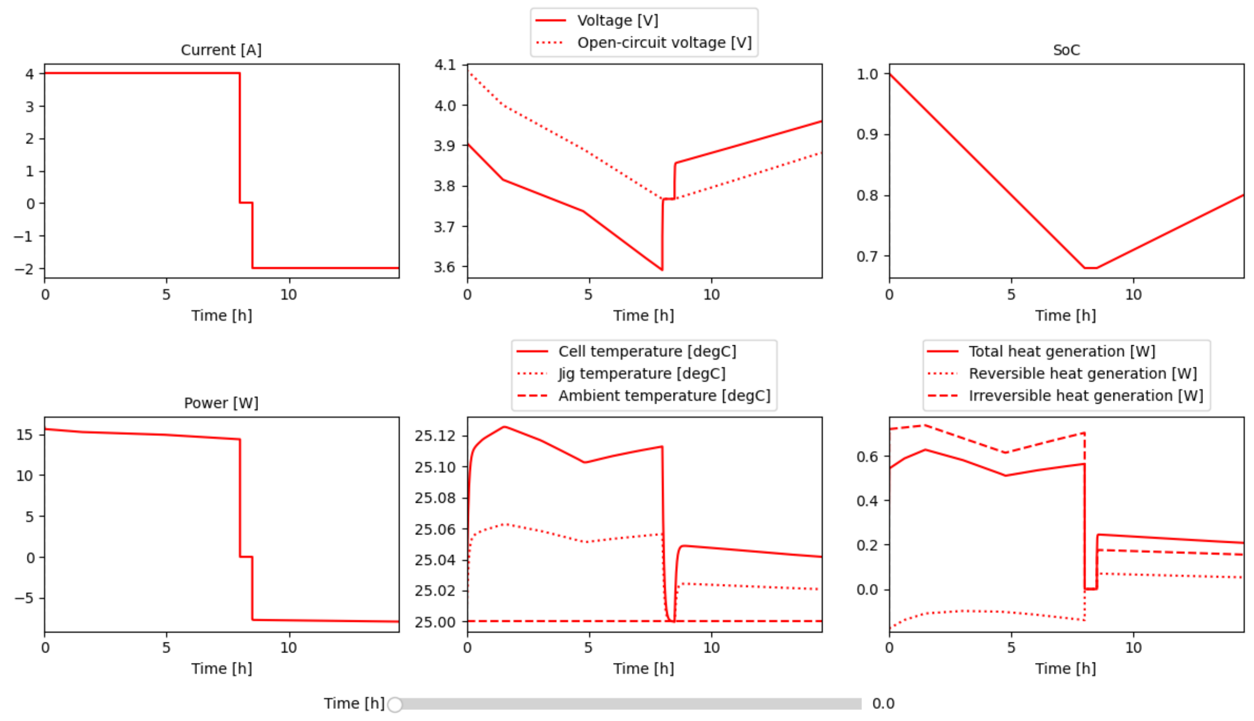


Simulation Data Plotter

Enter Task ID: f20bcc2f-0cfd-4229-8a36-8l

Comparing the above graph to what would be expected if I ran the simulation with PyBaMM in its own script unrelated to the API. We can see that the data we wanted to receive, being Voltage, Current and Temperature all align.

Fig D.1: Graphing of simulation results in PyBaMM



14.1 Future Considerations

14.2 Charge Profiles

If you notice from the above graph, there is a sharp return to rest when the battery charges. My simulation scenario included the instruction to “Charge at 2A for 6 hours”, yet the charge profile does not adhere like the discharge profile.

While trying to debug this with the data, I have come the conclusion that the SoC-OCV curve only capture the discharge dynamics. The Capacity test only had discharge cycles and no charge cycles, thus preventing me from capture greater charging characteristics. In further developing I would like to experiment with other datasets, that capture discharging elements.

14.3 JSON Truncation of Time

If I simulate a scenario that takes a considerable amount of time. E.g.

"experiment": [

"Discharge at 4 A until 2.5 V",

"Charge at 5 A until 4.2 V",

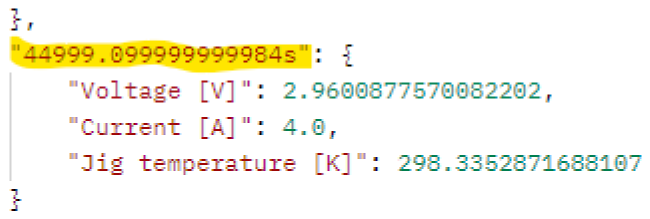
"Hold at 4.2 V until 3 A",

"Discharge at 5 A for 13 hours"

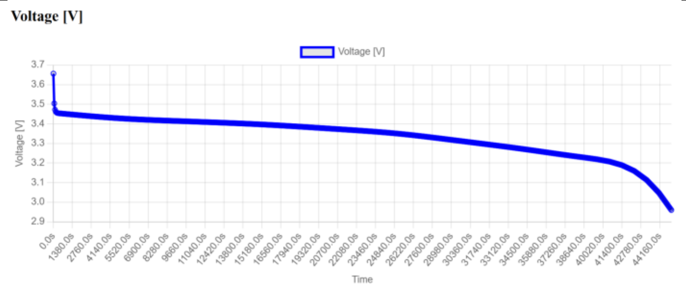
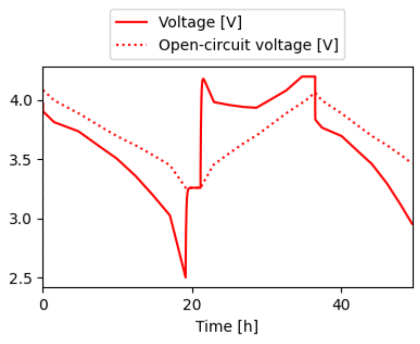
]

the API results won't produce higher than 44999 seconds.

Fig D.2: Simulation Json response unable to succeed 44999 seconds



```
},
"44999.099999999984s": {
    "Voltage [V]": 2.9600877570082202,
    "Current [A]": 4.0,
    "Jig temperature [K]": 298.3352871688107
}
```

API Response of the above simulation. Results graphed in JavaScript for comparative purposes	Native PyBaMM plotting of the same simulation results.
 Time ends: 44999 seconds	 Time does not end at 44999 seconds

15 Ethics

15.1 Environmental Concerns of Lithium Mining

This project focuses on the simulation and modelling of lithium-ion batteries, the beating heart and solution to today's energy storage systems. However, the ethical implications surrounding lithium mining and the exploitation of developing countries raise ethical concerns. Countries like the Democratic Republic of Congo, has raised significant environmental and human rights concerns.

Inspiration for this reflection came from Professor Siddharth Kara's *"Cobalt Red",* a book that documents the authors firsthand account witnessing the exploitative labour conditions and severe health risks faced by local communities, many of whom are involved in artisanal lithium and cobalt mining operations.

The ethical implications of lithium-mining highlight a darker side to clean energy, one that takes advantage of the health disproportionately by vulnerable populations.

While this simulator is not involved with battery manufacturing or lithium material sourcing, the ethical weight of these realities informs the motivation behind the project and cannot be dismissed.

By developing effective battery simulation tools, engineers and researchers can reduce reliance on physical prototyping, potentially lowering the demand for raw materials through improved design, extended battery life, and better battery recycling strategies.

16 Conclusion

To conclude my report, I want to break it down into 2 parts Outcome and Further Developments

The goal of my project was to develop an API based platform capable of simulation the behaviour of lithium-ion batteries, as well as creating models based on real-world battery test data.

16.1 Outcome

I was successful in achieving this goal. I was able to create an end-end workflow allowing me to input battery test data from a public dataset, obtain the information I needed to construct a model of the battery that I can simulate with by accessing an API.

This was a very academically and technically difficult project but very rewarding. While the main goal was achieved, I want to highlight some further developments that could emphasize a more thorough and complete platform.

16.2 Further Development

Further development of this project would include a

- **Interactive UI**:
    - Offering visual aid to simulation and parameterizing experimental data
- **Supporting other datasets**:
    - The hardest difficulty of datasets is aligning the type of data up. While all battery tests achieve the same outcome, not all data is the same label, structure or content wise. Being able to support a variety of datasets would be a great further development.
- **Deployment on the web/cloud**.
    - I have achieved an API endpoint exposure of the simulator but granting an endpoint to the parameterization codebase and exposing all of it online, would solidify the goal of a new workflow for battery development.
- **Automated Parameterization problem**.
    - While my code allows for auto-parameterization, it is at the levy that the fitting problem includes the same bounds, so results like in *section 8.2.4* can occur. Using a python package like Ray that can help identify the best bounds to use for a more thorough fitting would be an interest avenue to explore in future development.

17 References

[1] G. Pozzato, A. Allam, and S. Onori, “Lithium-ion battery aging dataset based on electric vehicle real-driving profiles,” *Data in Brief*, vol. 41, p. 107995, Apr. 2022, doi: https://doi.org/10.1016/j.dib.2022.107995.

[2] Battery University, “How to Measure Capacity” *Battery University*, Jan. 13, 2011. https://batteryuniversity.com/article/bu-904-how-to-measure-capacity (accessed Apr. 20, 2025).

[3] *Mathworks.com*, " Characterize Battery Cell for Electric Vehicles", 2015. https://uk.mathworks.com/help/simscape-battery/ug/battery-cell-characterization-for-ev.html (accessed Apr. 20, 2025).

[4] V. Sulzer, S. G. Marquis, R. Timms, M. Robinson, and S. J. Chapman, “Python Battery Mathematical Modelling (PyBaMM),” *Journal of Open Research Software*, vol. 9, no. 1, p. 14, Jun. 2021, doi: https://doi.org/10.5334/jors.309.

[5] Nigel, “SoC Estimation by Coulomb Counting,” *Battery Design*, May 27, 2022. https://www.batterydesign.net/soc-estimation-by-coulomb-counting/

[6] G. Plett, “ECE4710/5710: Modeling, Simulation, and Identification of Battery Dynamics Equivalent-Circuit Cell Models.” Available: http://mocha-java.uccs.edu/ECE5710/ECE5710-Notes02.pdf

[7] MarkVeerasingam, “GitHub - MarkVeerasingam/BatteryECM_Identification_API,” *GitHub*, 2025. https://github.com/MarkVeerasingam/BatteryECM_Identification_API (accessed Apr. 22, 2025).

[8] “What Is Particle Swarm Optimization? - MATLAB & Simulink - MathWorks United Kingdom,” *uk.mathworks.com*. https://uk.mathworks.com/help/gads/what-is-particle-swarm-optimization.html

[9] MarkVeerasingam, “GitHub - MarkVeerasingam/LGM50t_cell_analysis at main,” *GitHub*, 2025. https://github.com/MarkVeerasingam/LGM50t_cell_analysis/tree/main (accessed Apr. 22, 2025).

[10] MarkVeerasingam, “GitHub - MarkVeerasingam/Battery-Simulator,” *GitHub*, 2024. https://github.com/MarkVeerasingam/Battery-Simulator (accessed Apr. 28, 2025).