Python Battery Mathematical Modelling (PyBaMM)

Valentin Sulzer[a], Scott G. Marquis[a], Robert Timms[a,c], Martin Robinson[b,c], and S. Jon Chapman[a,c]

[a] *Mathematical Institute, University of Oxford, Andrew Wiles Building, Woodstock Road, Oxford, UK, OX2 6GG, UK*

[b] *Department of Computer Science, University of Oxford, 15 Parks Rd, Oxford OX1 3QD, UK*

[c] *The Faraday Institution, Quad One, Becquerel Avenue, Harwell Campus, Didcot, OX11 0RA, UK*

Abstract

As the UK battery modelling community grows, there is a clear need for software that uses modern software engineering techniques to facilitate cross-institutional collaboration and democratise research progress. The Python package PyBaMM aims to provide a flexible platform for implementation and comparison of new models and numerical methods. This is achieved by implementing models as expression trees and processing them in a modular fashion through a pipeline. Comprehensive testing provides robustness to changes and hence eases the implementation of model extensions. PyBaMM is open source and available on GitHub at `https://github.com/pybamm-team/PyBaMM`.

1 Overview

1.1 Introduction

With the battery modelling research community growing rapidly in the UK in the last few years [1], it is essential to develop tools that facilitate cross-institutional collaboration. One such tool is battery modelling software that allows new research (e.g. new physics, numerical methods) to be employed with minimal effort by the rest of the community. Modelling software should also facilitate quantitative comparison of different models and numerical methods. To be reusable and extendable, the software should be both modular, so that new models and numerical methods can easily be added, and rigorously tested, in order to be robust to changes.

1.2 Existing software

Currently, COMSOL [2] is the modelling software most commonly utilised by the the battery modelling community, providing a simple graphical user interface (GUI) to implement and solve standard battery models. However, there are several drawbacks to COMSOL. Firstly, its prohibitively expensive licence fee is a barrier to collaboration and sharing of software. Secondly, it has limited flexibility for adaptation of existing battery models, or investigation of new numerical methods. Thirdly, as the implementation is performed through a GUI, programs cannot be directly scripted without additional software, which inhibits version control, unit testing and combination with other software (for example, for parameter estimation).

Several existing open-source battery modelling software packages provide an alternative option to COMSOL. Examples include DUALFOIL [3], fastDFN [4], LIONSIMBA [5] and M-PET [6]. However, each of these packages is focused on the implementation of one specific battery model under a specific choice of operating conditions. As a result, these software packages lack the flexibility required to allow for easy implementation of reduced-order versions of a model or to include model extensions. This lack of flexibility considerably limits the reuse of such packages across different research projects. In addition, the open-source software packages currently available do not employ a test-driven development structure, which increases the difficulty of cross-institutional collaboration.

1.3 Overview of PyBaMM

PyBaMM offers improved collaboration and research impact in battery modelling by providing a modular framework through which either existing or new tools can be combined in order to solve continuum models for batteries. For example, PyBaMM can easily be adapted to incorporate new models, alternative spatial discretisations, or new time-stepping algorithms. Any such additions can then immediately be used with the existing suite of models already implemented, and comparisons can be made between different models, discretisations, or algorithms with variables such as hardware, software and implementation details held fixed. Similarly, additional physics can be incorporated into the existing models, without needing to start from scratch to study each new effect. This facilitates the simultaneous study of a range of extensions to the standard battery models, for example by coupling together several degradation mechanisms.

PyBaMM is one of the major components of the Faraday Institution's 'Common Modelling Framework', part of the Multi-Scale Modelling Fast Start project, which will act as a central repository for UK battery modelling research. PyBaMM has already been used to develop and compare reduced-order models for lithium-ion [7] and lead-acid [8, 9] batteries, and further research outcomes are anticipated from continued collaborations with other members of the Faraday modelling community.

1.4 Implementation and architecture

PyBaMM's architecture is based around two core components. The first is the expression tree, which encodes mathematical equations symbolically (see Figure 1). Each expression tree consists of a set of symbols, each of which represents either a variable, parameter, mathematical operation, matrix, or vector. Every battery model in PyBaMM is then defined as a collection of symbolic expression trees. The expression trees in each model are organised within python dictionaries representing the governing equations, boundary equations, and initial conditions of the model. An example of implementing a simple diffusion model using expression trees is provided in Appendix A.

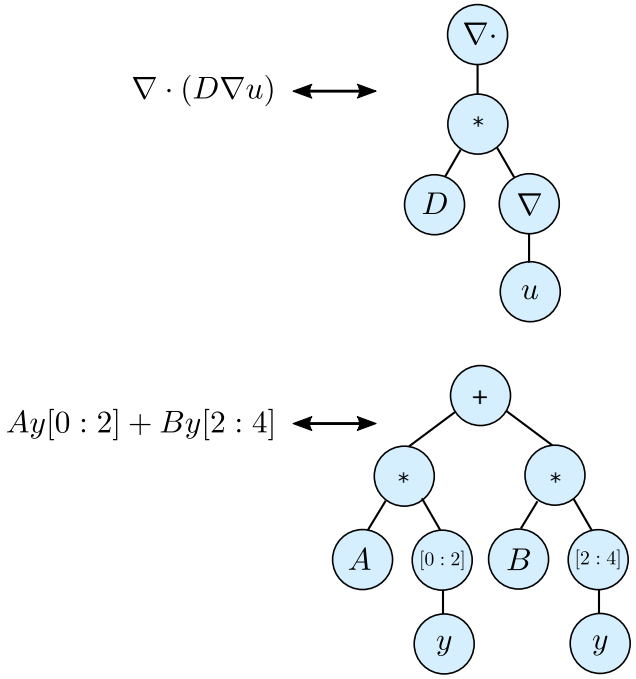

Figure 1: Models are encoded and passed down the pipeline using a symbolic expression tree data-structure. Leafs in the tree represent parameters, variables, matrices etc., while internal nodes represent either basic operators such multiplication or division, or continuous operators such as divergence or gradients.

The second core component of PyBaMM's architecture is the pipeline process (see Figure 2). In the pipeline process different modular components operate on the model in turn. The pipeline is constructed in

Python using PyBaMM classes, so that users have full control over the entire process, and can customise the pipeline or insert their own components at any stage. Figure 2 depicts a typical pipeline with the following stages:

1. Define a battery model and geometry using PyBaMM's syntax. This generates a collection of expression trees representing the model.
2. Parse the expression trees for the battery model and geometry, replacing any parameters with their provided numerical values. For convenience, parameter values may be provided in a csv file.
3. Mesh the geometry and discretise the model on this mesh with user-defined spatial methods. This process parses each expression tree converting variables into state vectors, and spatial operators (e.g. gradient and divergence) into matrices (accounting for the boundary conditions of the model).
4. Solve the model using a time-stepping algorithm. PyBaMM offers a consistent interface to a number of ordinary differential equation (ODE) and differential algebraic equation (DAE) solvers (including SciPy [10] and SUNDIALS [11, 12, 13]). One of the main benefits of PyBaMM's expression tree structure is that it provides the capability to automatically compute the Jacobian for any model, using symbolic differentiation, which significantly improves the performance of the numerical solvers.
5. Post-processes the solution. Built-in post processing utilities provided access to any user-defined output variables at any solution time or state. Additionally, PyBaMM includes a number of visualisation utilities which allow for easy plotting and comparison of any of the model variables (for example output, see Figure 3).

An example of employing the PyBaMM pipeline process to solve a model is provided in Appendix B.

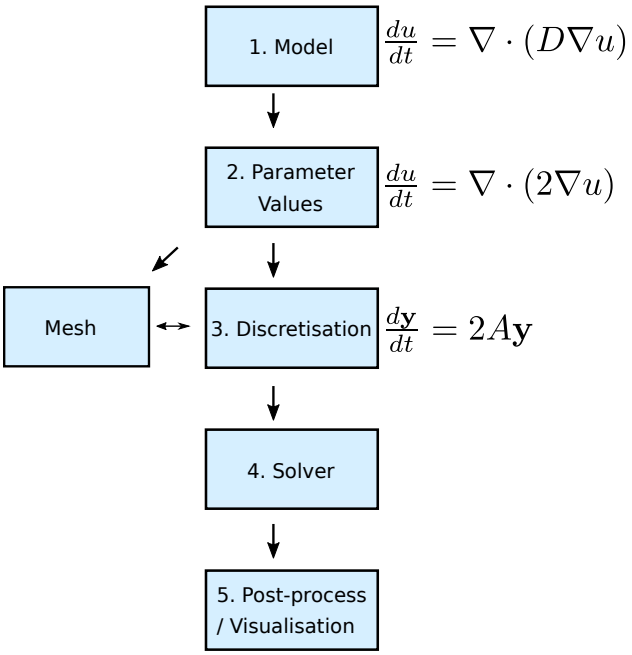

Figure 2: PyBaMM is designed around a pipeline approach. Models are initially defined using mathematical expressions encoded as expression trees. These models are then passed to a class which sets the parameters of the model, before being discretised into linear algebra expressions, and finally solved using a time-stepping class.

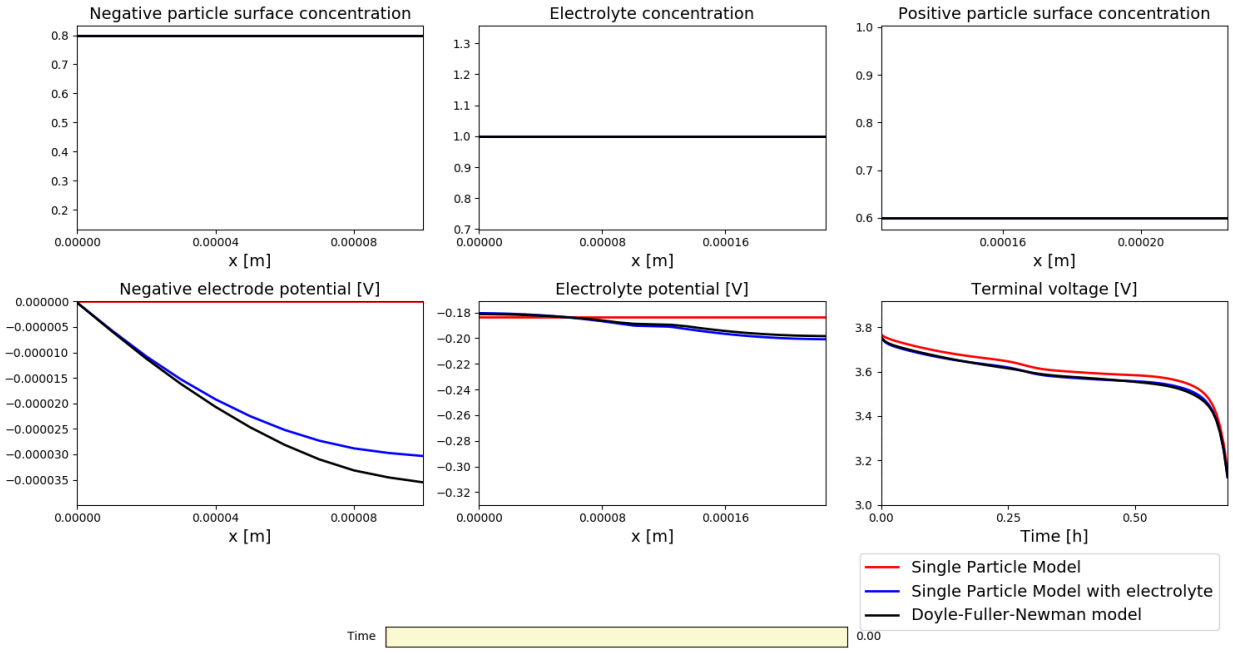

Figure 3: Interactive output of inbuilt visualisation. The user can select the time at which to view the output using the time-slider bar at the bottom. For a given list of variables and solved models, such plots are automatically generated.

1.5 Quality control

Tests in PyBaMM are performed within the `unittest` framework. We follow a test-driven development process, and unit tests are implemented for every class with unit test code coverage consistently above 98%. In addition, a smaller set of integration tests are implemented to ensure the end-to-end reliability of the code. The integration tests consist of tests that check every model in PyBaMM can be processed and solved for a set of default inputs, convergence tests between reduced-order and full-order models, converge tests for each spatial method, and tests for each solver type.

PyBaMM is developed using git version control, with all unit and integration tests being run on Travis CI for Linux (Ubuntu 16.04) and macOS High Sierra systems, and Appveyor for Windows systems every time a pull request is made. The only exception to this is the (optional) DAE solver in PyBaMM, which has only been developed and tested for Linux, due to its third-party dependencies of scikits-odes [13] and SUNDIALS 3.1.1 [11, 12]. At the time of writing, the PyBaMM CI tests run on Ubuntu and Windows systems with Python 3.6-3.7, and macOS with Python 3.6.

The main PyBaMM repository contains a selection of Jupyter Notebooks that serve as a "getting started" documentation for PyBaMM, and a useful set of examples on how to use PyBaMM for different tasks such as creating a new battery model, running the existing models, or changing the default parameters. These are tested along with the main PyBaMM code to ensure they are up to date.

Please consult the `CONTRIBUTING.md` file in the PyBaMM repository for more detailed and up-to-date information on our development workflow, testing and CI infrastructure, and coding style guidelines.

2 Availability

2.1 Operating system

The core PyBaMM code can be run on any Linux, MacOS and Windows sytems that has Python 3.6 or higher installed, along with the dependencies listed below. In order to solve any of the the DAE battery

models an optional dependency, scikits-odes and SUNDIALS, must be installed and a Linux OS is currently recommended for this. For Windows users, we therefore recommend using Windows Subsystems for Linux (WSL); detailed instructions are available on GitHub.

2.2 Programming language

Python 3.6 or higher

2.3 Additional system requirements

PyBaMM has no special requirements and can be run on a standard laptop or desktop machine.

2.4 Dependencies

Required:

- numpy≥1.16
- scipy≥1.0
- pandas≥0.23
- anytree≥2.4.3
- autograd≥1.2
- scikit-fem≥0.2.0
- matplotlib≥2.0

Optional:

- scikits.odes≥2.4.0 (for solving DAE models, requires SUNDIALS 3.1.1)

2.5 List of contributors

Valentin Sulzer, Scott Marquis, Robert Timms, Martin Robinson, Fergus Cooper, Ferran Brosa-Planella, Tom Tranter

2.6 Software location:

2.6.1 Release

Name: GitHub (release v0.1.0)
Persistent identifier: `https://github.com/pybamm-team/PyBaMM/releases/tag/v0.1.0`
Licence: BSD 3-clause
Publisher: The PyBaMM team
Version published: v0.1.0
Date published: 03/10/19

2.6.2 Code repository

Name: GitHub
Persistent identifier: `https://github.com/pybamm-team/PyBaMM`
Licence: BSD-3-Clause
Date published: 04/11/2018

2.7 Language

English

3 Reuse potential

We anticipate that the main use case will be the implementation, extension, and comparison of new models and parameter sets. For example, this will allow researchers to implement models that couple several degradation mechanisms together. Further, although PyBaMM has been written with battery models in mind, the expression tree and pipeline architecture could be potentially be used to solve different sets of continuum models numerically.

In addition to new models and parameter sets, the modular framework described in Section 1 allows researchers to add new numerical algorithms in the form of spatial discretisations or new ODE/DAE solvers. Any such extensions can then be immediately tested with the existing set of models and parameters. This allows researchers to quickly assess the accuracy and speed of their numerical algorithms for a range of models and relevant parameter values.

Information on how to extend the software in these ways is available both through tutorials in the API docs and example notebooks. All of the development is done through GitHub issues and pull requests, using the workflow explained in the `CONTRIBUTING.md` file. Users can request support by raising an issue on GitHub.

Acknowledgements

The authors are grateful to all contributors, workshop attendees, and the following people for useful discussions, feedback and support on PyBaMM: Jacqueline Edge, Jamie Foster, David Howey, Ivan Korotkin, Charles Monroe, Greg Offer, Colin Please, and Giles Richardson.

Funding statement

VS acknowledges funding from the EPSRC Center for Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in collaboration with BBOXX. SM acknowledges funding from the EPSRC Center for Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in collaboration with Siemens. RT and JC acknowledge support from the Faraday Institution (EP/S003053/1). MR acknowledges funding from the EPSRC Impact Acceleration Account - University of Oxford (D4D00010).

Competing interests

The authors declare that they have no competing interests.

References

[1] EPSRC Press Office. Greg Clark announces Faraday Institution, October 2017. `https://epsrc.ukri.org/newsevents/news/faradayinstitution/`. [Online; accessed 11-September-2019].

[2] COMSOL Inc. COMSOL multiphysics reference manual, version 5.4. `https://www.comsol.com`.

[3] J Newman. FORTRAN programs for the simulation of electrochemical systems. `http://www.cchem.berkeley.edu/jsngrp/fortran.html`.

[4] SJ Moura. Fast doyle-fuller-newman (DFN) electrochemical-thermal battery model simulator. `https://github.com/scott-moura/fastDFN`.

[5] M Torchio, L Magni, RB Gopaluni, RD Braatz, and DM Raimondo. LIONSIMBA: A matlab framework based on a finite volume model suitable for li-ion battery design, simulation, and control. *Journal of The Electrochemical Society*, 163(7):A1192–A1205, 2016. doi: 10.1149/2.0291607jes.

[6] RB Smith and MZ Bazant. Multiphase porous electrode theory. *Journal of The Electrochemical Society*, 164(11):E3291–E3310, 2017. doi: 10.1149/2.0171711jes.

[7] SG Marquis, V Sulzer, R Timms, CP Please, and SJ Chapman. An asymptotic derivation of a single particle model with electrolyte. *arXiv preprint arXiv:1905.12553*, 2019.

[8] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part I. physical model. *Journal of The Electrochemical Society*, 166(12):A2363–A2371, 2019. doi: 10.1149/2.0301910jes.

[9] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part II. asymptotic analysis. *Journal of The Electrochemical Society*, 166(12):A2372–A2382, 2019. doi: 10.1149/2.0441908jes.

[10] E Jones, T Oliphant, P Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. `http://www.scipy.org/`.

[11] AC Hindmarsh. The PVODE and IDA algorithms. Technical report, Lawrence Livermore National Lab., CA (US), 2000. doi:10.2172/802599.

[12] AC Hindmarsh, PN Brown, KE Grant, SL Lee, R Serban, DE Shumaker, and CS Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005. doi: 10.1145/1089014.1089020.

[13] B Malengier, P Kišon, J Tocknell, C Abert, F Bruckner, and M-A Bisotti. ODES: a high level interface to ODE and DAE solvers. *The Journal of Open Source Software*, 3(22):165, feb 2018. doi: 10.21105/joss.00165.

[14] M Doyle, TF Fuller, and J Newman. Modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal of the Electrochemical society*, 140(6):1526–1533, 1993. doi: 10.1149/1.2221597.

A Creating a model

In this section, we present an example of how to enter a simple diffusion model in PyBaMM. This model serves as a good representation of the types of models that arise in battery modelling because it contains most of the key components: spatial operators, parameters, Dirichlet and Neumann boundary conditions, and initial conditions.

We consider the concentration of some species c, on a spatial domain $x \in [0, 1]$, and at some time $t \in [0, \infty)$. The concentration of the species is taken to evolve according to a nonlinear diffusion process with the concentration being fixed at $x = 0$ and a constant inward flux of species imposed at $x = 1$. Mathematically, the model is stated as

$$\frac{\partial c}{\partial t} = \nabla \cdot (D(c)\nabla c) \qquad \text{in } 0 < x < 1, \quad t > 0, \tag{1a}$$

$$c = 1 \qquad \text{at } x = 0, \tag{1b}$$

$$D(c)\frac{\partial c}{\partial x} = 1 \qquad \text{at } x = 1, \tag{1c}$$

$$c = x + 1 \qquad \text{at } t = 0, \tag{1d}$$

where $D(c) = k(1 + c)$ is the diffusion coefficient and k is a parameter, which we will refer to as the diffusion parameter.

In Listing 1, we provide the PyBaMM code implementing (1). Note that operator overloading of $*$ and $+$ allows symbols to be intuitively combined to produce expression trees. A more detailed and up-to-date introduction to the syntax is provided in the online examples available on GitHub.

The model is now represented by a collection of expression trees and can therefore be solved by passing it through the pipeline just like any other model in PyBaMM. Additionally, extending the model to include additional physics is simple and intuitive due to the simple symbolic representation of the underlying mathematical equations. For example, we can add a source term to the governing equation (1a) by only modifying one line of code (line 10 of Listing 1) and still obtain useful properties of the model such as the analytical Jacobian.

Listing 1: Defining a model in PyBaMM

```
# 1. Initialise model
model = pybamm.BaseModel()

# 2. Define parameters and variables
c = pybamm.Variable("c", domain="unit line")
k = pybamm.Parameter("Diffusion parameter")

# 3. State governing equations
D = k * (1 + c)
dcdt = pybamm.div(D * pybamm.grad(c))
model.rhs = {c: dcdt}

# 4. State boundary conditions
D_right = pybamm.BoundaryValue(D, "right")
model.boundary_conditions = {
  c: {
    "left": (1, "Dirichlet"),
    "right": (1/D_right, "Neumann")
  }
}

# 5. State initial conditions
x = pybamm.SpatialVariable("x", domain="unit line")
model.initial_conditions = {c: x + 1}
```

B Solving a model

We now consider an example of solving a PyBaMM model by passing it through the pipeline process. Here, we solve the Doyle-Fuller-Newman (DFN) model, which is the standard model of a lithium-ion battery [14]. The code is presented in Listing 2. Please see the PyBaMM documentation for more detailed and up-to-date examples.

The common interface of all PyBaMM models makes it easy to perform the pipeline process as illustrated here upon multiple models or the same model with different options activated. Therefore, comparing the results of different models, mesh types, discretisations, and solvers then becomes straightforward within the PyBaMM framework.

Listing 2: Solving a model in PyBaMM

```
# 1. Load model and geometry
model = pybamm.lithium_ion.DFN()
geometry = model.default_geometry

# 2. Process parameters
param = model.default_parameter_values
param.process_model(model)
param.process_geometry(geometry)

# 3. Set mesh
mesh = pybamm.Mesh(
          geometry,
          model.default_submesh_types,
          model.default_var_pts
        )

# 4. Discretise model
disc = pybamm.Discretisation(
          mesh, model.default_spatial_methods
        )
disc.process_model(model)

# 5. Solve model
t_eval = numpy.linspace(0, 0.2, 100)
solution = model.default_solver.solve(model, t_eval)
```