Journal of The Electrochemical Society, 2021 **168** 090504

Methods—PETLION: Open-Source Software for Millisecond-Scale Porous Electrode Theory-Based Lithium-Ion Battery Simulations

Marc D. Berliner,*[*] Daniel A. Cogswell, Martin Z. Bazant,[**] and Richard D. Braatz[,z]**

Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, United States of America

This article presents PETLION, which is an open-source, high-performance computing implementation of the porous electrode theory (PET) model in Julia. A typical runtime for a dynamic simulation of full charge or discharge is 3 ms on a laptop while allocating about 1 MB of total memory, and the software is seen to be two orders of magnitude faster than comparable software for some applications. At moderate spatial resolutions, the computation times are similar to those of reduced-order and reformulated models in the literature. Multiple numerical solvers and methods for their initialization are compared in terms of numerical convergence and computational times, for a wide variety of operating conditions. PETLION is shown to quickly and robustly simulate complex battery protocols such as the Galvanostatic Intermittent Titration Technique (GITT), and to achieve high performance when incorporated into real-time PET-based nonlinear model predictive control.

Manuscript submitted May 3, 2021; revised manuscript received July 22, 2021. Published September 2, 2021.

Lithium-ion batteries have become ubiquitous in modern technology including laptops, cell phones, and automobiles. The 2019 Nobel Prize in Chemistry, awarded to Goodenough, Whittingham, and Yoshino for the development of Li-ion batteries,[1] highlights the enormous environmental and societal impact that this technology has and will continue to have on the world. Combined with renewable energies such as wind and solar, Li-ion batteries have the potential to create a society free of fossil fuel. Global adoption of electric vehicles (EV) has been rapidly rising over recent years. According to the International Energy Agency, in 2018 there were more than 14 times the amount of EVs on the road compared to just 5 years prior.[2] This trend will only increase, as government incentives to reduce emissions make EVs an attractive alternative to internal combustion engine vehicles. For wide adoption, batteries must be safe, cheap, allow for large driving ranges, and charge quickly with minimal degradation.[3]

Developing accurate, computationally efficient methods for predicting capacity fade in battery management systems (BMS) motivates this work. Batteries are commonly modeled using porous electrode theory (PET).[4,5] This model is widely used to simulate battery cycling behavior by describing electrochemical kinetics and transport in solid particles and electrolyte and modeling thermodynamics by fitting an open circuit voltage. The model consists of tightly coupled, nonlinear partial differential-algebraic equations. The common practice is to fit a half dozen effective transport and kinetic coefficients in the PET model to battery cycling data, and then to use the model to explore changes in the battery design or the operating conditions.

With the rise of EVs, quickly and accurately estimating battery parameters for fleets of EVs is crucial to assess and mitigate battery degradation. Fitting battery parameters to cycling data with the PET model quickly becomes challenging due to the large number of physical parameters, the computational cost of solving the model numerous times in series, and parameter identifiability problems when estimating even a small subset of parameters. For these reasons, researchers often employ simplified models such as the Single Particle Model (SPM), the Equivalent Circuit Model (ECM), reduced order models (ROMs), or mathematical reformulations of the modeling equations.[6,7] While directly lowering the computational cost, these methods involve replacing the original model equations with alternative equations that can have reduced accuracy when the operating conditions stray from nominal ranges, e.g., at high currents.

This article provides an open-source, high-performance computing (HPC) implementation of the PET model in Julia which is performant and robust for many operating conditions. The next section summarizes the differential-algebraic equations that define the PET model, and the finite volume method (FVM) implementation of the PET model in MATLAB known as LIONSIMBA that serves as the numerical foundation for our Julia implementation. Readers interested in a detailed description of the governing equations, structure, and parameterization of the model are referred to a previous publication.[8] Then we discuss the options implemented in PETLION, validates PETLION against LIONSIMBA, and reports the results of simulations for CC-CV charging, the Galvanostatic Intermittent Titration Technique (GITT), and model predictive control. Then the efficiency of the residuals and Jacobian functions are evaluated over a range of discretizations and varying temperature dynamics. Then different techniques are described for initializing a semi-explicit system of DAEs and its effect on the PET model stability. Then the efficiency of various stiff differential-algebraic equation (DAE) solvers available in Julia are compared for the PET model. Lastly, PETLION, LIONSIMBA, PyBaMM, DUALFOIL, and COMSOL MultiPhysics are compared.

Background

Porous electrode theory and software implementations.— Porous Electrode Theory (PET) was developed by Newman and collaborators at the University of California, Berkeley.[4,5,9,10] The main component of PET is the porous electrode, where the solid-phase active material and conductive matrix has pores filled with liquid electrolyte. Fickian diffusion and Ohmic conduction dynamically transport lithium ions between active particles in the electrolyte. The PET model is commonly referred to as a "pseudo-two-dimensional" (P2D) model, in which one dimension is the position along the length of the battery x between the two metal contact points on the opposite sides of the electrode-separator-electrode sandwich and the second dimension is the distance from the center of a solid particle r (Fig. 1). The two phases are coupled by interfacial electrochemical kinetics, such as Butler-Volmer or Marcus-Hush-Chidsey kinetics.[11] Solid-phase transport is assumed to be Fickian or approximated with a polynomial for small applied currents. The states of the PET model are the electrolyte and volume-averaged solid concentrations c_e and c_s^{avg}, the ionic flux j, the electrolyte and solid potentials Φ_e and Φ_s, the

*Electrochemical Society Student Member.
**Electrochemical Society Member.
[z]E-mail: braatz@mit.edu

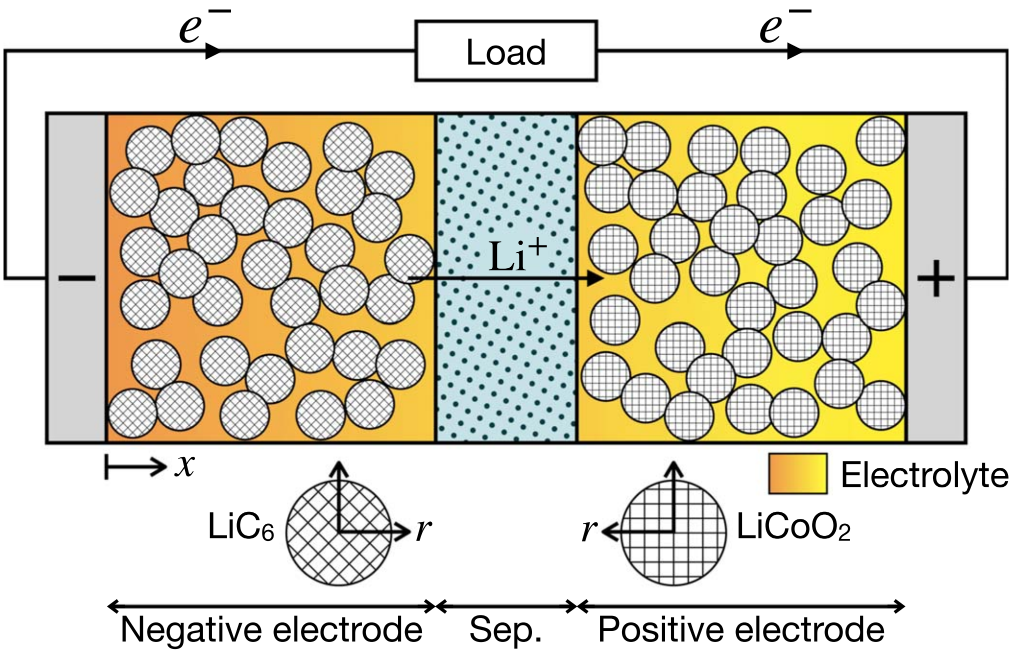


Figure 1. Schematic of the PET model for an $LiC_6/LiCoO_2$ cell during discharge.

applied current I, as well as the temperature T for non-isothermal simulations.

The PET model is a set of partial differential and algebraic equations (PDAE). Many software implementations of PET have been developed.[12–15] LIONSIMBA[8] is a MATLAB implementation of the finite volume method (FVM) for the P2D lithium-ion battery model. Advantages of the FVM are (1) its exact handling of flux boundary conditions and total conservation of all conserved variables (e.g., Li atoms) throughout the control volume and (2) its relatively simple implementation compared to the finite element method (FEM), which makes the code easier to understand and modify. The software implementation converts the PDAE system into a set of ordinary differential and algebraic equations (DAEs) with time t as the independent variable.[16,17] The FVM is implemented in the x dimension, with N cells for the cathode, separator, and anode. For the r dimension for solid particle concentration, the user can choose between an 8th-order accurate finite difference method (FDM) or a spectral method. The number of equations in the DAE system resulted from the above discretizations is listed in Table A·I. In the LIONSIMBA paper,[8] the software was thoroughly validated by comparisons to DUALFOIL[18] and COMSOL MultiPhysics.[19]

Several other battery simulation tools for the PET model are available in the literature. DUALFOIL[18] is an open-source battery simulation software from the Newman group written in Fortran. DUALFOIL employs Newman's BAND(j) subroutine which first creates a local linearization of the nonlinear governing PDEs and subsequently puts the equations in finite-difference form.[20] The system of equations is then solved at each time step using Newton's method. DUALFOIL offers constant current, power, and voltage operation, cell-averaged temperature, and foil mode, which treats the anode as a non-porous Li-metal electrode. PyBaMM[21] is an flexible, open-source, multi-scale battery simulator developed in Python that is in active development at the time of publication. PyBaMM offers several options to the user, including multiple geometries, the PET and Single Particle models, FVM and FEM discretizations, and several ODE and DAE solvers. COMSOL MultiPhysics[19] is a commercial software for solving PDEs using the finite element method that offers an additional battery simulation module. COMSOL provides a graphical user interface to change simulation options, modify the geometry, and view results. COMSOL offers several battery modules in 1D, 2D, and 3D, including lithium-ion, lead-acid, nickel-metal hydride (NiMH), vanadium redox flow, and soluble lead-acid flow. The Model Comparison section includes further comparisons of the computational times for these software packages.

Differential-algebraic equation (DAEs).—In many physical systems, including for Li-ion batteries, the governing laws such as the conservation of charge are algebraic. DAEs allow for these algebraic constraints to be incorporated directly to the system of equations. DAEs can be specified in *fully implicit* form,

$$F(\dot{y}(t), y(t), t; \theta) = 0, \tag{1}$$

or in mass matrix form,

$$M(t; \theta)\dot{y}(t) = f(t, y(t); \theta), \tag{2}$$

where y is the vectors of states, $\dot{y}$ is the derivatives of y with respect to time t, θ is a vector of model parameters, F and f are vectors of algebraic functions, and M is the square mass matrix. If the mass matrix M is nonsingular, then the latter equation can be rewritten in the form of an ordinary differential equation (ODE). For many systems such as lithium-ion batteries, the mass matrix M is singular, and the more general DAE formulation is required. If the mass matrix M is diagonal with each element being a 0 or a 1, then the DAE is *semi-explicit*.

Usually the first step in the numerical solution of a DAE is the determination of a consistent set of initial conditions for the algebraic and differential variables. Typically, the initial value of the state vector, $y(0; \theta)$, must be solved for using known $\dot{y}(0; \theta)$, or a combination of both algebraic and differential variables must be determined given some subset of the variables defined by the physical operation of the system. The set of equations that define consistent initial conditions for DAEs typically include implicit equations that cannot be solved analytically, necessitating the application of nonlinear root-finding to determine the initial algebraic and/or differential states.

PETLION Package

PETLION was created to be an open-source, high-performance computing model that is simple to use and modify for even non-Julia users. While implementing the PET model in a lower-level language,

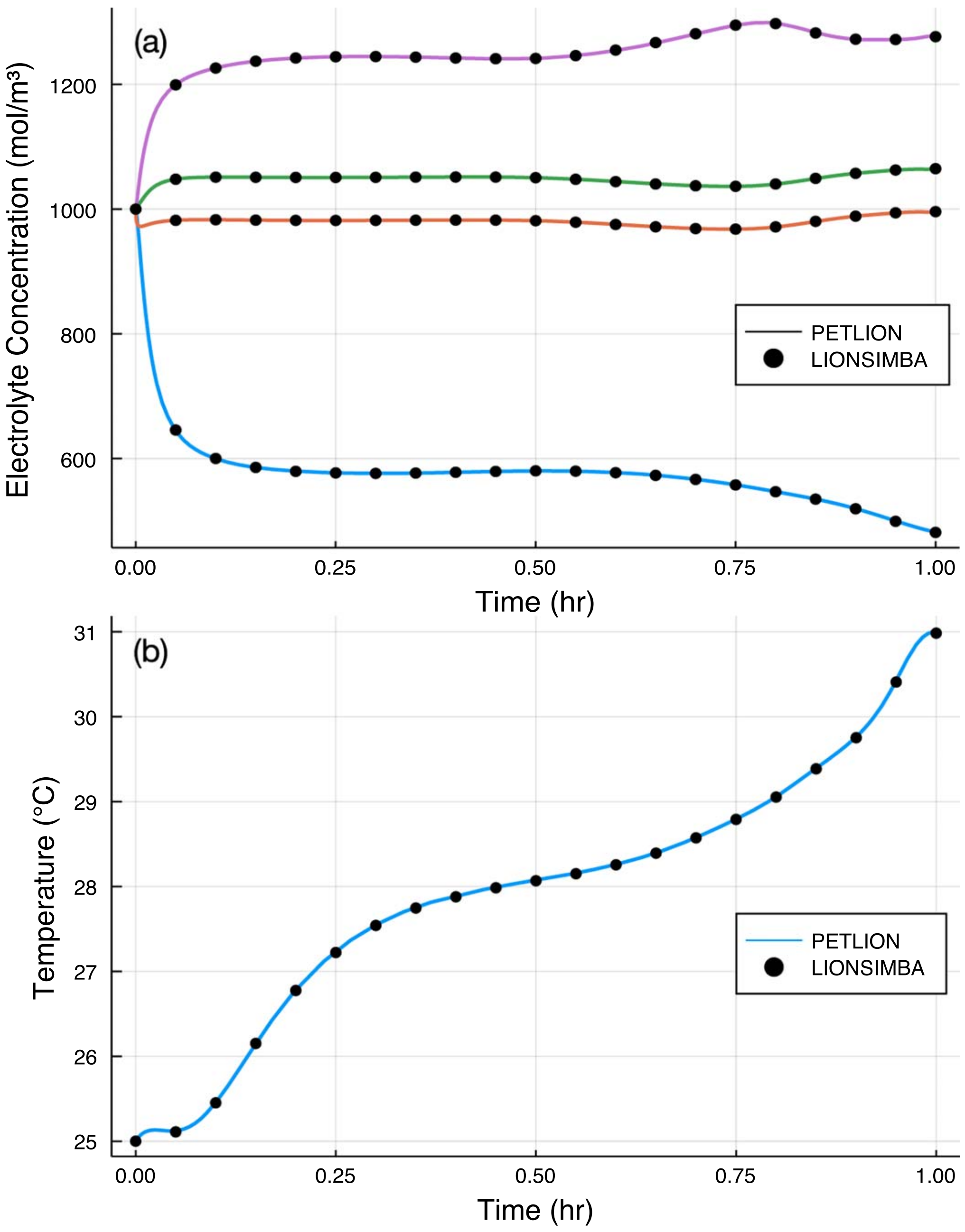
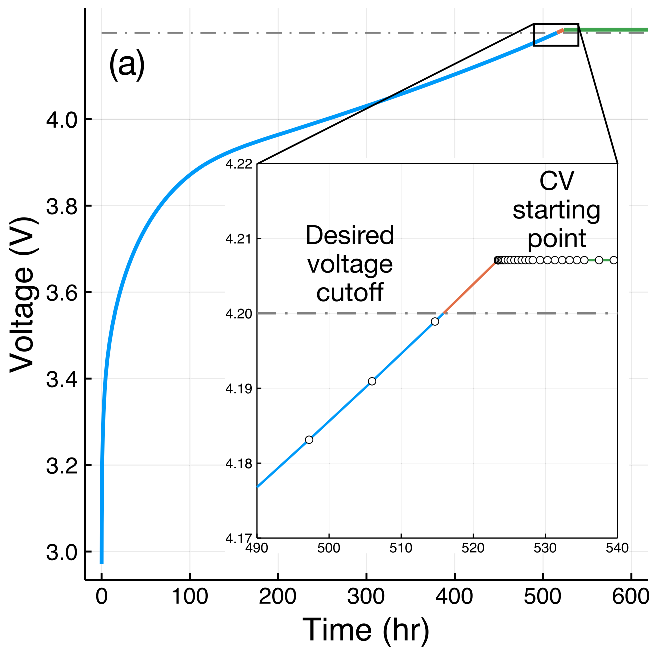


Figure 2. Comparison of the transient electrolyte concentration c_e (a) and mean temperature T (b) evaluated in PETLION and LIONSIMBA. The lines from top to bottom are interfacial c_e values at the anode-current collector, separator-anode, cathode-separator, and current collector-cathode.

such as C++ or Fortran, is likely to be more efficient than in a more user-friendly language, such as Julia, MATLAB, or Python among others, the learning curve for beginners can be steeper.[22] High performance models written in MATLAB and Python often necessitate the functions to be converted to a faster, low-level language (also known as the *"two-language" problem*[23,24]) which can hinder code modification and debugging—Julia avoids this issue since its compiled code speed is comparable to C++. Julia was ultimately chosen for its open-source license, its relative ease of use, fast speed after compilation, and the growing ecosystem for tools like neural ODEs and physics-informed neural networks. The PETLION GitHub download page[25] contains installation instructions and examples for running and modifying the parameters and equations.

Model validation.—PETLION uses the same modeling equations as LIONSIMBA, which has been validated with other codes, so PETLION can be validated by just comparing the simulation results to LIONSIMBA. The relative and absolute errors of the DAE solver were set to 10^{-3} and 10^{-6}, respectively. Simulation results are compared in Fig. 2 for a 1C discharge for the PET model with full Fickian diffusion, temperature dynamics, and 10 discretizations per section for a total of 341 DAEs. The electrolyte concentration c_e and mean temperature T profiles for the codes overlap for all time t. The relative errors for all of the PETLION output states are all within the specified tolerances (Table I).

Using PETLION.—PETLION allows for straightforward evaluation of complex battery protocols. Numerous options are available in this package, including the ability to easily change chemistries and parameters, run constant or variable current, voltage, and power simulations; user-defined functions for current, voltage, and power may be (dis)continuous and/or functions of the battery states. Optionally available are 1D temperature dynamics throughout the cell and current collectors, the ability to choose between



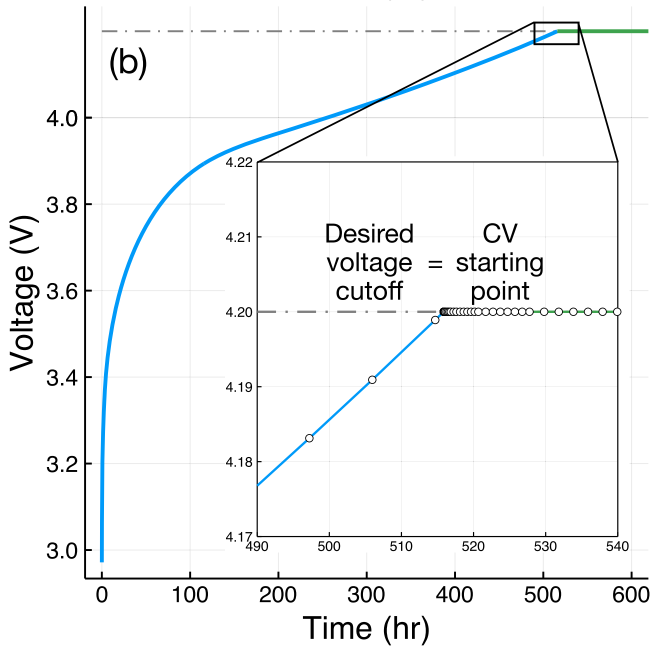


Figure 3. 4C CC-CV simulation with a voltage cutoff at 4.2 V using (a) a discrete callback mechanism and (b) a continuous callback mechanism. The discrete callback leads to an incorrect starting point for the CV simulation. The continuous callback corrects this and starts the CV step at 4.2 V. The bullets indicate model time points, and the dotted line is the desired CV cutoff value of 4.2 V.

Table I. Relative error comparison of PETLION with LIONSIMBA. All output state errors are <10^{-5}.

	V	c_e	c_s^{avg}	T	j	Φ_e	Φ_s
Relative error (10^{-6})	7.11	9.36	7.33	8.04	5.81	8.46	8.88

Butler-Volmer or Marcus-Hush-Chidsey reaction kinetics, and the inclusion of aging effects due to growth of the solid-electrolyte interface (SEI) on the anode. Running arbitrary current functions is an important feature for battery models. In real-world driving situations, the applied current is often discontinuous, quickly switching from discharge to charge due to sudden stops or regenerative breaking. Currently, PETLION does not support forward or adjoint parametric sensitivity analyses.

The rest of this section provides a series of examples for demonstrating the running of PETLION.

Example 1: stopping conditions, CC-CV charging.—Many battery protocols require a stop condition to terminate the simulation, such as reaching a maximum or minimum voltage or state-of-charge (SOC). Oftentimes models employ a *discrete callback* mechanism: after each iteration, the stop conditions are checked and the simulation is terminated if any stop condition is reached. This way is simple and efficient but does not guarantee that the model run terminates at the desired stop conditions within machine precision. To illustrate the issue with discrete callbacks, consider a 4C CC-CV simulation with a voltage cutoff at 4.2 V shown in Fig. 3a and simulated with the code below. The zoomed-in region showcases the final time points of the CC step and the beginning of CV. The discrete nature of the callback mechanism means the model will overshoot the final time point and the final states will be wrong—in this example, the result is running the CV step at an incorrect constant voltage of 4.207 V instead of the 4.2 V that was specified. In many applications, simulations must continuously run the same cell over hundreds of cycles, such as for simulating long-term degradation, which can magnify such errors and cause unpredictable and random behavior when simulating changes from one experiment to the next.

```
# Load the model and parameters with an initial SOC of 0
p = Params(LCO; SOC = 0)
# 4C CC charge until 4.2 V
model = run_model(p, I = 4, V_max = 4.2)
# CV hold for 100 s
run_model! (model, p, 100, V = :hold)
```

By default, PETLION implements a *continuous callback* mechanism that interpolates the final states with respect to the stopping condition.[a] The continuous callback mechanism quickly and accurately ensures that consecutive simulations will not propagate error, unlike a discrete callback. Figure 3b shows the CV step beginning correctly at 4.2 V with corrected final states using the continuous callback.

Example 2: Galvanostatic Intermittent Titration Technique (GITT).—GITT is a common electrochemical experimental procedure for determining diffusion coefficients and open circuit voltages (OCVs) using short current pulses followed by relaxation periods.[26] The few lines of code below simulate 20 periods of 1C pulses followed by 2 hour rests to fully charge a cell (see Figure 4). The zoomed-in region shows a quick jump in voltage from the current pulse, followed by a slow rise before the current is shut off and the voltage subsequently falls. This dynamic matches experimental behavior for GITT with lithium-ion batteries, e.g., as reported in Ref. 26.

```
# Load the model and parameters with an initial SOC of 0
p = Params(LCO; SOC = 0)
model = model_output()
# GITT: 20 1C pulses followed by 2 hour rests
for i in 1:20
    run_model!(model, p, 3600/20, I = 1)
    run_model!(model, p, 2 * 3600, I = :rest)
end
```

[a]The discrete callback mechanism is an option implemented in PETLION, so that users can compare to other codes, but is not recommended for the reasons discussed in the previous paragraph.

Example 3: PET-based nonlinear model predictive control (NMPC).—Quick and accurate optimal operation of batteries is important for advanced battery management systems (ABMS). Some researchers have investigated the direct or indirect use of the PET model within real-time NMPC algorithms, as a potential way to better incorporate physics and battery degradation (e.g., see[27,28] and citations therein). Models embedded in real-time NMPC algorithms must be able to robustly and quickly handle discontinuities. Nonlinear models may have difficulties when encountering a discontinuity if the states before and after the discontinuity are very different—PETLION is capable of handling discontinuities in current, voltage, and power functions.

A staircase NMPC algorithm was implemented to optimize 80 constant currents in which each lasts ten seconds. The C-rates are bounded between 0 and 4, and their values are determined by a numerical optimization for maximizing the cell's SOC in 800 s while keeping the temperature under 40 °C. The NMPC algorithm was evaluated under two scenarios: (1) a nominal simulation with an ambient temperature of 25 °C and (2) a disturbance simulation with an ambient temperature of 26 °C. The NMPC results are shown in Fig. 5 and the code to simulate each model evaluation is

```
# Load the model and parameters with an initial SOC of 0
# and enable temperature
p = Params(LCO; temperature = true, SOC = 0)
model = model_output()
# Set the NMPC parameters
N = 80
tf = 800
I_vector = ... # Vector of C-rates with length N
# Run the model at an ambient temperature of 25°C
p.θ[:T_amb] = 25 + 273.15
for I in I_vector
    run_model!(model, p, tf/N, I = I)
end
```

The control algorithm charges the cell at the maximum current until the temperature reaches its maximum, whereafter the temperature is carefully regulated. The disturbance simulation has a larger heat flux from the environment than the nominal simulation, leading to a slightly lower SOC after 800 s.

Residuals and Jacobian

The computational speed of the software depends on the accuracy and speed of computing Jacobians used in the simulation code. The residuals and sparse Jacobian functions were generated and saved to disk locally via ModelingToolkit.jl, which is a high-performance symbolic framework for scientific computing[29] that supports a full computer algebra system (CAS). ModelingToolkit integrates itself seamlessly with Julia—even code for a complex model such as PETLION requires little to no modifications to work with ModelingToolkit. This is in contrast to CasADi,[30] another popular equation-based modeling tool, which has limited CAS support and that may necessitate rewriting and/or specifically writing functions to work with CasADi expressions and syntax. Debugging code with ModelingToolkit is efficient since it is written in and outputs functions in native-Julia, compared to CasADi which is written in C++ but is often accessed in high-level programming languages such as MATLAB or Python.

The Jacobian was also evaluated using sparse forward-mode automatic differentiation (AD) with a greedy distance-1 graph coloring algorithm to reduce the number of total function evaluations.[31] Since Julia is compiled at run time, the first time loading and evaluating these functions can be RAM-intensive and slow, often requiring tens of seconds to minutes. After the initial compilation, the subsequent evaluations are significantly faster. When the model code is actively being updated (e.g., when prototyping a new model) the AD Jacobian option is recommended over the symbolic Jacobian because it has shorter initial compilation times.

The memory allocations for the residuals and Jacobian functions are 0 and 48 bytes, respectively, leading to highly performant simulations. Table II reports the runtimes for calculating the residuals and Jacobian, with the corresponding number of equations per discretization being reported in Table A·I. The runtimes for both the symbolic residuals and Jacobian are on the order of one to tens of μs, with approximately linear relationship between the runtime and the number of discretization points, indicating that these main functions are very well optimized. The AD Jacobian is several time slower than the symbolic Jacobian, resulting in total simulation time increasing by about 30%–50% (see Table III for an example).

DAE Initialization

The first step in solving the DAEs is to initialize the vectors of states for the stack of algebraic terms, $y(0;\theta)$, and differential terms, $\dot{y}(0;\theta)$. While an implicit DAE initialization code solves for $y(0;\theta)$ and $\dot{y}(0;\theta)$ simultaneously, the initialization process for a DAE in semi-explicit form can be nested into two steps:[17] (1) solve only algebraic equations for $y(0;\theta)$ and (2) analytically calculate $\dot{y}(0;\theta)$ using the constant mass matrix and optimized values for $y(0;\theta)$. This two-step initialization is beneficial because the size of the Jacobian is smaller when excluding the differential terms and it will reduce error since the differential components can be solved within machine precision. The most expensive part of this procedure is the LU decomposition of the Jacobian, so reducing the number of equations in the residuals function by ~4.5× (see Table A·I) will give a performance boost. Mitigating error in the DAE initialization step is important to avoid propagating error throughout the

Table II. Runtime for each residual and sparse Jacobian iteration for (non-)isothermal simulations for a range of discretizations. The forward-mode automatic differentiation (AD) Jacobian uses a greedy distance-1 graph coloring algorithm to reduce the number of function evaluations. All tests are performed on a 2019 MacBook Pro 2.4 GHz 8-Core Intel i9 computer with 32 GB of RAM.

Function	Method/Runtime	Discretizations per Section, N					
		5	10	15	20	25	30
Residuals (Symbolic)	Isothermal (μs)	1.629	3.133	4.841	6.138	7.782	9.950
	Non-isothermal (μs)	1.751	3.217	4.965	6.478	8.085	10.08
Jacobian (Symbolic)	Isothermal (μs)	4.659	9.873	14.50	17.96	22.18	26.48
	Non-isothermal (μs)	6.390	12.48	18.69	23.35	29.34	34.77
Jacobian (AD)	Isothermal (μs)	25.13	52.02	80.16	101.1	126.7	152.1
	Non-isothermal (μs)	50.25	103.5	157.3	207.8	262.5	312.0

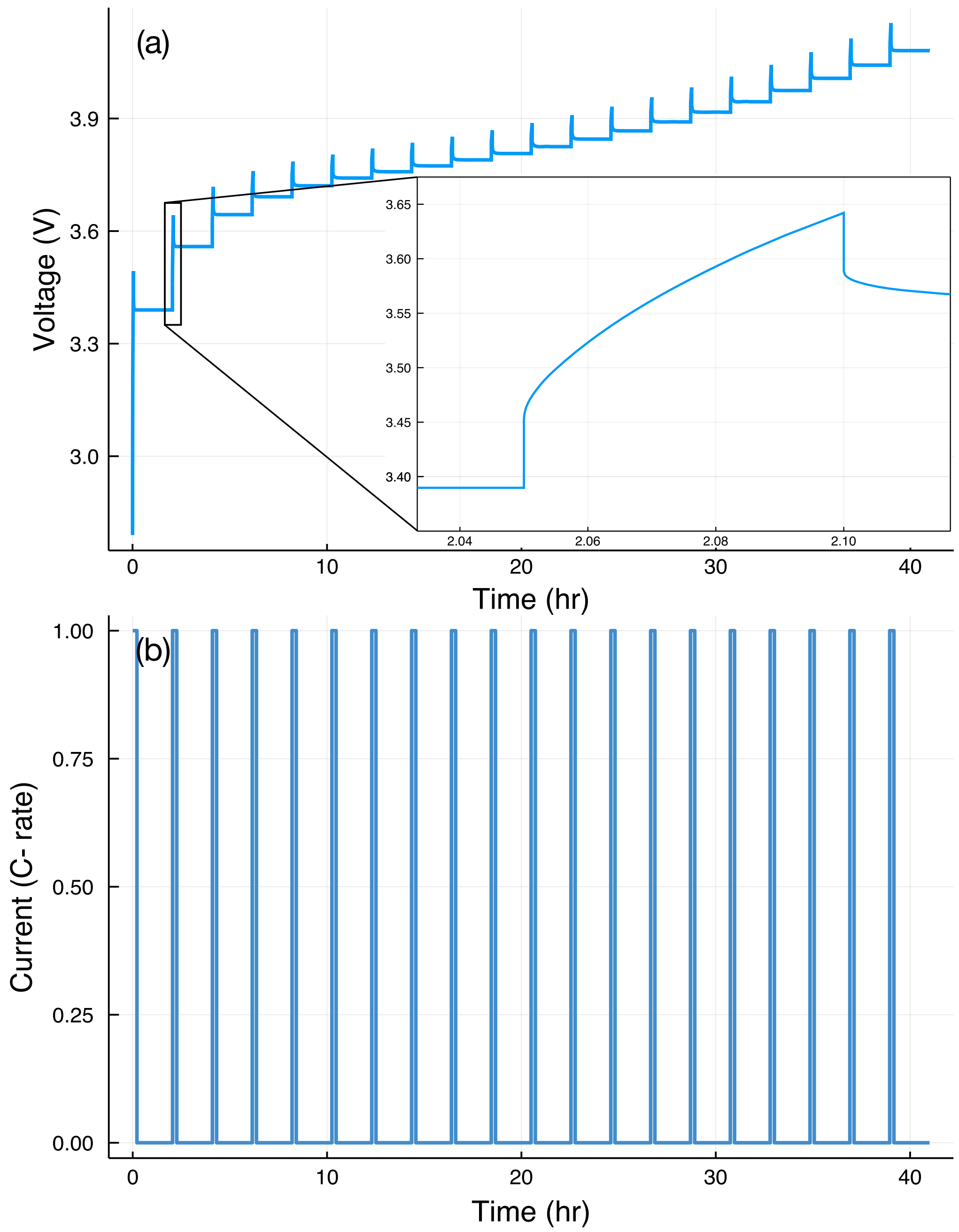


Figure 4. GITT simulation results for (a) the voltage with a zoomed-in region showing the dynamics over a single 1C pulse and (b) the applied current.

simulation and for solver instability, which is discussed in detail in the Numerical stability subsection. The initial guess for initialization is taken from a known equilibrium position: the OCV if starting a new run, or the final states of a previous run if continuing a simulation.

Boovaragavan et al.[32] proposed a technique to calculate the initial algebraic states in the electrodes using the shooting method. Three ODEs for $j(x)$, $\Phi_e(x)$, and $\Phi_s(x)$ are solved spatially between the endpoints of each electrode with an analytical solution available for $\Phi_e(x)$ in the separator (the paper includes c_s^* as a reformulated algebraic state). The pros and cons of this method compared to the initialization methods in PETLION are discussed below.

Initial state estimation.—Four nonlinear root-finding methods were evaluated: trust region,[33] Newton's method, Quasi-Newton's method,[34] and Anderson acceleration.[35] Of these four methods, only the trust region and Newton's method successfully found the roots for each simulation. Newton's method reached the desired absolute or relative tolerance cutoff 6.8x faster on average than the trust region method. While trust region methods are designed for high stability, that high stability comes a limit on how fast the solution can be reached, which requires more iterations to converge. Newton's method is known to be unstable for poor initial guesses, however, initializing from a known equilibrium is very stable over a wide range of C-rates (see Numerical stability subsection).

The region-wise shooting method approach of Boovaragavan et al.[32] was also investigated as an alternative to nonlinear root-finding. The ODE function and residuals were generated as described in the Residual and Jacobian ection. A single stiff ODE solve using the Rosenbrock23 algorithm takes about 300–400 μs compared to 110 μs with Newton's method with 10 discretizations/section (see Table III). The final time for initialization in the region-wise shooting method is several times this value because the shooting method requires a few evaluations of the ODE to find the

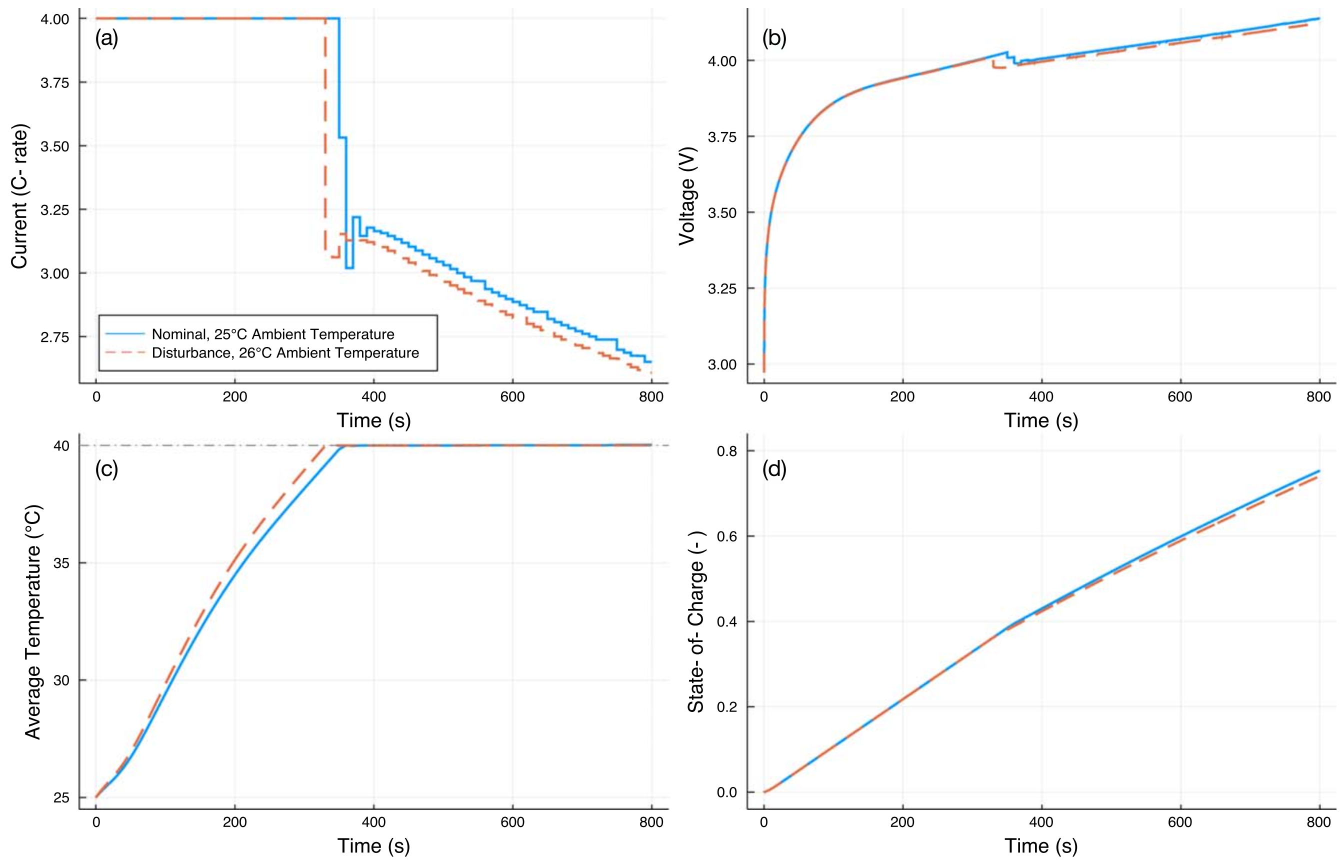


Figure 5. Model-based optimal control results for (a) the current, (b) the voltage, (c) the average temperature with maximum temperature shown by a dotted line, and (d) the state-of-charge (SOC).

correct initial states and additionally, these evaluations must be performed for each electrode. While the region-wise shooting method is slower than Newton's method for moderate discretizations, the method can be advantageous for very fine discretizations because its runtime is independent of the mesh size. As discussed in the DAE Solver section, such fine discretizations are not needed to achieve high accuracy in lithium-ion battery simulations.

Numerical stability.—Accurate consistent initialization for DAEs is an important part of maintaining the numerical stability in a simulation code, especially when running simulations under more extreme conditions, e.g., at very high charging rates. These considerations are especially important when embedding the model into optimization algorithms to solve such systems engineering problems as state-of-charge (SOC) and state-of-health (SOH) estimation and real-time model-based optimal control.

Poor initializations can result in a failure to start the simulation, or in error propagation throughout the model evaluation which may lead to failure. Oftentimes with large systems of equations, the DAE initialization can be inconsistent, i.e., the model fails to converge somewhat randomly through various starting conditions. Figure 6 showcases this issue for a fully implicit initialization with the function IDACalcIC from Sundials which uses a modified Newton's method.[36] Numerous full (dis)charge experiments are evaluated from -4 C to 4 C and their success or failure to initialize is denoted. Using a nominal set of parameters for an LCO/LiC$_6$ battery, the fully implicit initialization fails between 2.72 C to 2.87 C. Additionally, when updating the cathode porosity $\epsilon_p \leftarrow 0.9\epsilon_p$, the model fails in a different range of C-rates (2.41 C to 2.45 C). The sporadic failures become even more problematic when the parameters are ill-posed, as is often the case when the model is embedded in an optimization. Using the semi-explicit DAE

Table III. Breakdown of the model evaluation time for an example isothermal 1C discharge simulation at $N = 10$ discretizations. The details of the PETLION Jacobian are provided in the Residuals and Jacobian section. All models use default options except for printing which is disabled.

Model	Note	Evaluation Time			
		DAE Initialization	Solver	Overhead	Total
PETLION	Symbolic Jacobian	0.110 ms	2.78 ms	0.070 ms	2.96 ms
PETLION	AD Jacobian	0.112 ms	4.39 ms	0.068 ms	4.57 ms
LIONSIMBA	Jacobian provided[a)]	6.51 ms	149 ms	35.3 ms	191 ms
LIONSIMBA	Jacobian on runtime	6.51 ms	149 ms	308 ms	463 ms
PyBaMM "fast"	Direct integration	65.4 ms	31.9 ms	15.4 ms	113 ms
PyBaMM "safe"	Step-and-check integration	62.4 ms	152 ms	12 ms	226 ms

a) The LIONSIMBA Jacobian cannot be reused when the parameters or variable current/power functions are modified.

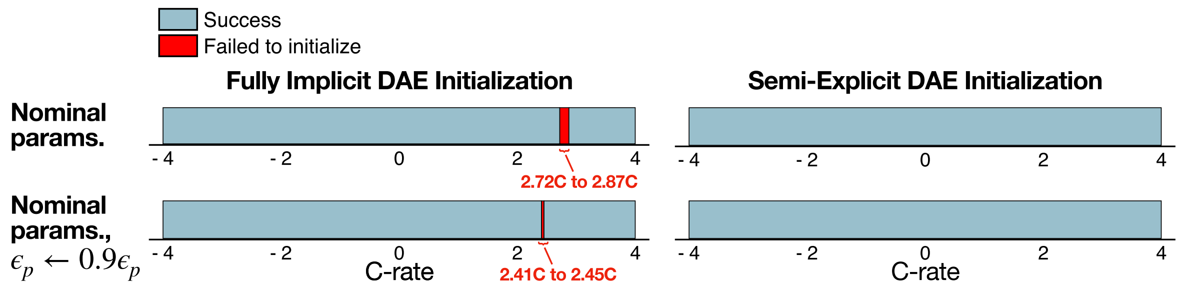


Figure 6. Numerical stability of the DAE initialization for a full (dis)charge using (left) modified Newton's method for fully implicit DAEs (right) Newton's method for semi-explicit DAEs. For some range of initial C-rates, the fully implicit method fails to initialize whereas the semi-explicit method is successful in all cases.

initialization with Newton's method successfully initializes for all conditions.

DAE Solver

The PETLION model equations are formulated as a semi-explicit DAE, which were fed to several stiff DAE solvers in Julia,[29] including the Sundials IDA with KLU linear solver ("Sundials"),[36] Rodas5, Rosenbrock23, and Radau5. The input to Sundials is a fully implicit DAE, which needs to be fed a sparse Jacobian with the KLU linear solver. Rodas, Rosenbrock23, and Radau5 use the mass matrix form. The current implementations of Rodas and Rosenbrock23 in Julia can be fed sparse Jacobians, whereas Radau5 uses a dense Jacobian (its Fortran code is incompatible with sparse Julia types). The model runs were benchmarked using 10 linearly spaced C-rates ranging from -2 C to 2 C with full Fickian diffusion (see Fig. 7). The time required for computing a consistent DAE initialization was not included in the runtime comparison since the same initialization was fed to all four solvers.

The most efficient solver by far is Sundials, which is more than an order of magnitude faster than the others, followed by Rodas5, Rosenbrock23, and Radau5. The slowness of Rosenbrock23 and Rodas5 for larger numbers of DAEs is consistent with past reports that these methods perform best with $<$100 DAEs.[29] Radau5 is slow because it cannot work with the sparse matrices from Julia, which leads to poor scaling with increased discretizations. Given that the Jacobian in this model is quite sparse, significant performance penalties occur when evaluating Jacobian steps in the Radau5 solver.

Solver Tolerances. The speed and accuracy of the simulations depend on the specified absolute and relative tolerances. The states span many orders of magnitude, ranging from 1,000–10,000 for concentrations, 1 for potentials, and 10^{-7}–10^{-5} for ionic fluxes. Due to these different scales, the relative tolerance is a better descriptor of model accuracy than the absolute tolerance.

There exists no analytical solution to the PET modeling equations, so a high-resolution simulation with a fine mesh ($N = 30$ discretizations/section, relative and absolute tolerances of 10^{-13}) is treated as the "exact" solution. This "exact" solution is used as a baseline for comparison against a coarser mesh and lower tolerances which would be relevant in applications. Figure 8 shows the error for a coarse simulation with $N = 10$ and absolute tolerance of 10^{-6} undergoing a full 1C charge. The high-resolution simulation agrees very well with the coarse mesh simulation. The error with a relative tolerance of 10^{-3} begins quite oscillatory, but these oscillations become less pronounced over time. With a tighter relative tolerance of 10^{-6}, the oscillations occur but are much smaller in amplitude. Tightening the relative tolerance even further continues to slightly dampen these oscillations, but these differences are visually indistinguishable from a relative tolerance of 10^{-6}. Despite its jagged appearance, the 10^{-3} relative tolerance has a similar average error compared to 10^{-6}, 8.93×10^{-5} and 7.87×10^{-5} respectively. For many use scenarios, a relative

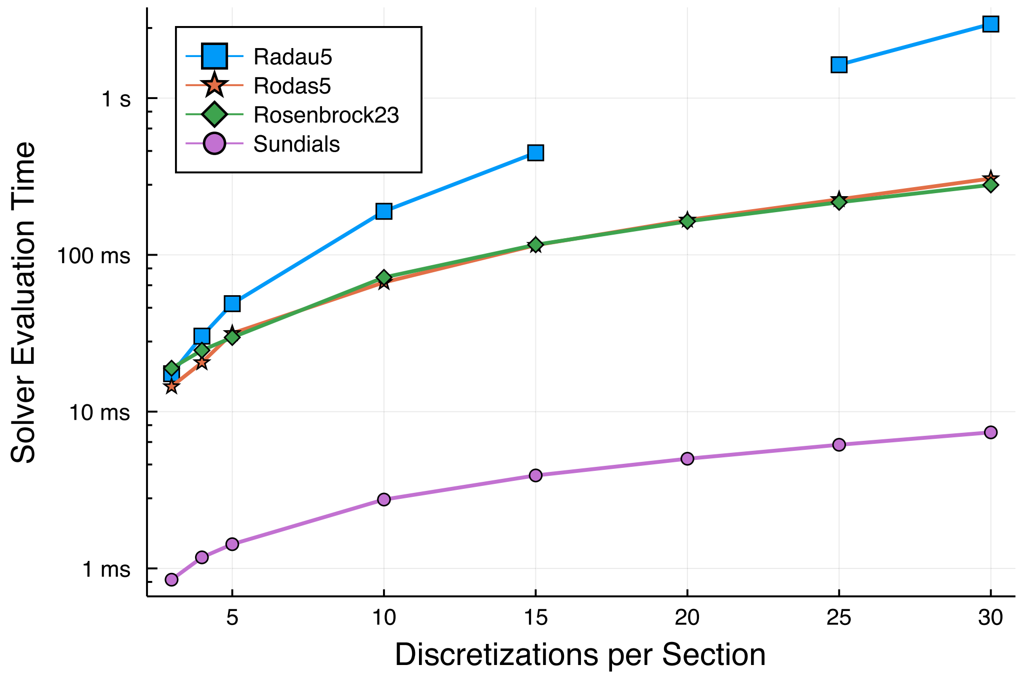


Figure 7. Runtime comparison of various DAE solvers available in Julia. The simulations are isothermal with Fickian diffusion and $N = 10$. The break in the Radau5 line is due to solver failure.

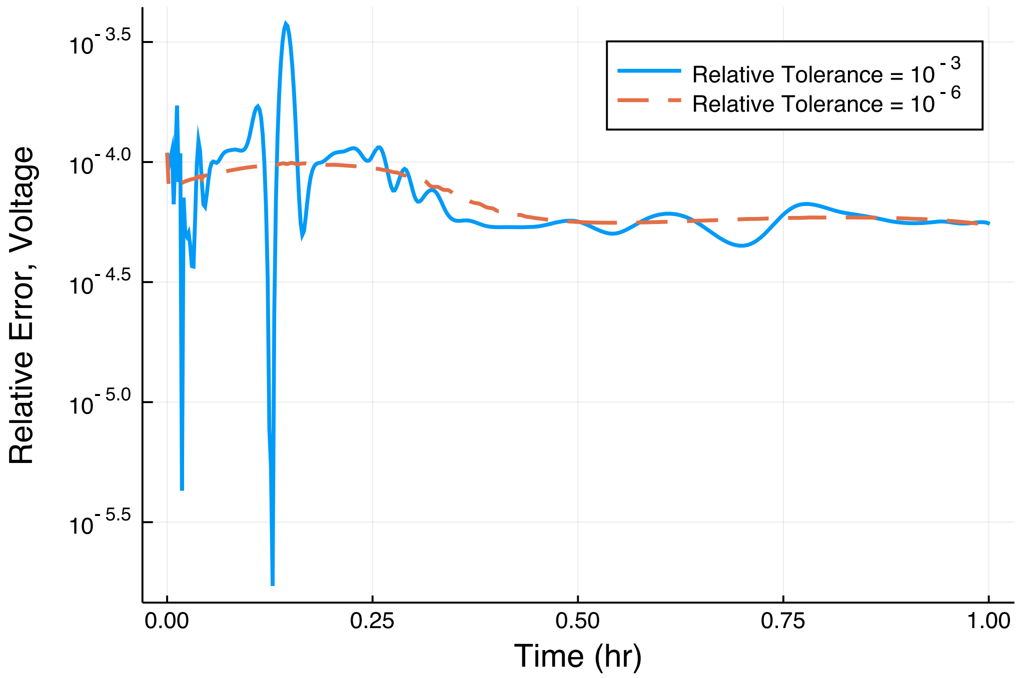


Figure 8. Comparison of the relative errors in the voltage for coarse simulations of a full 1C charge. The coarse simulations have $N = 10$ discretizations/section and an absolute tolerance of 10^{-6}, for two values for the relative tolerance. The high-resolution simulation used as the baseline for computing the errors has $N = 30$ discretizations/section and absolute and relative tolerances of 10^{-13}.

tolerance of 10^{-3} is acceptable especially when considering the large speedup compared to tighter tolerances.

Model Comparison

PETLION is designed for fast serialized model evaluations. The above sections detail the performant residuals and Jacobian functions, DAE initialization, and DAE solvers. Additionally, overhead from operations such as reading/writing data from the disk and assigning values to variables contributes to the total evaluation time of a model. PETLION reduces the overhead to a minimum with additional memory-saving options to prevent any unneeded outputs from the model.

DUALFOIL and COMSOL are robust P2D solvers but have slow runtime which makes them impractical for parameter estimation, MPC, or real-time control purposes. A major reason DUALFOIL is slower than other tools is because of the time-stepping algorithm used to integrate the system of equations. The PET model is particularly stiff during the first few seconds after an applied current or voltage and then becomes much less stiff over time, resulting in much faster runtime when using an adaptive time stepper whose Δt spans many orders of magnitude. The DUALFOIL time-stepping algorithm does not update Δt very frequently—often appearing as pseudo-constant for most of the simulation—which increases the simulation time. DUALFOIL also requires finer spatial discretizations due to the method of formulating the system of equations. The BAND(j) subroutine linearizes the nonlinear governing PDEs and then discretizes them with a finite difference method,[20] requiring a larger number of spatial discretization points to satisfy the local linearity approximation. The linearized system of equations is solved at each time step using Newton's method, which is a highly accurate way of solving the equations, but computationally expensive due to the frequent LU decompositions of the Jacobian matrix. COMSOL numerically solves differential equations using the finite element method (FEM) which divides the domain into multiple small, simple elements. FEM is particularly well-suited to solve structural analysis problems, but computational fluid dynamics typically employ finite difference or volume methods which are better suited to handle high fluxes. In battery simulations, finite elements need finer discretizations to match the accuracy of finite difference or volume at much coarser discretizations. In addition, FEM can be unstable in time for larger Δt which increases the number of steps to integrate the system of equations. Torchio et al.[8] evaluated DUALFOIL and COMSOL speeds for isothermal conditions and found each simulation took tens of seconds to minutes even for moderate discretizations. The current versions of PETLION, LIONSIMBA, and PyBaMM are several orders of magnitude faster than both DUALFOIL and COMSOL.

PETLION, LIONSIMBA, and PyBaMM are tools with speeds that make them suitable for parameter estimation or control purposes. Table III shows the breakdown for evaluation time for an example 1C discharge at $N = 10$ discretizations. In PETLION, solving the DAE contributes to >90% of the total evaluation time while DAE initialization and overhead are about 5% and 1% respectively. The AD Jacobian causes the solver time to increase by nearly 60% compared to the symbolic Jacobian, but does not significantly impact the initialization time. When the model equations are changing rapidly (e.g., when prototyping model equations), an AD Jacobian may be preferable because creating new symbolic functions is more time-consuming. When performance is important, the symbolic Jacobian is preferable. In LIONSIMBA, the Jacobian function must be recreated upon any parameter or variable current/voltage/power function updates which contributes to significant overhead time. When LIONSIMBA is embedded in parameter estimation or MPC experiments, the total evaluation time is significantly burdened by the overhead. Even when providing the Jacobian, PETLION is still >60× faster than LIONSIMBA for this example. In PyBaMM, there are two solver modes: "safe" which checks for battery stop conditions on every iteration of the DAE solve ($V(t) < V_{\min}$, SOC(t) < $\text{SOC}_{\min}$, etc..) and is suitable for drive cycles which do not encounter events, and "fast" which does not make these checks and is suitable for full charge or discharge cycles which typically encounter events. The "safe" mode performs step-and-check integration which leads to additional slowness from Python, while the "fast" performs direct integration that avoids

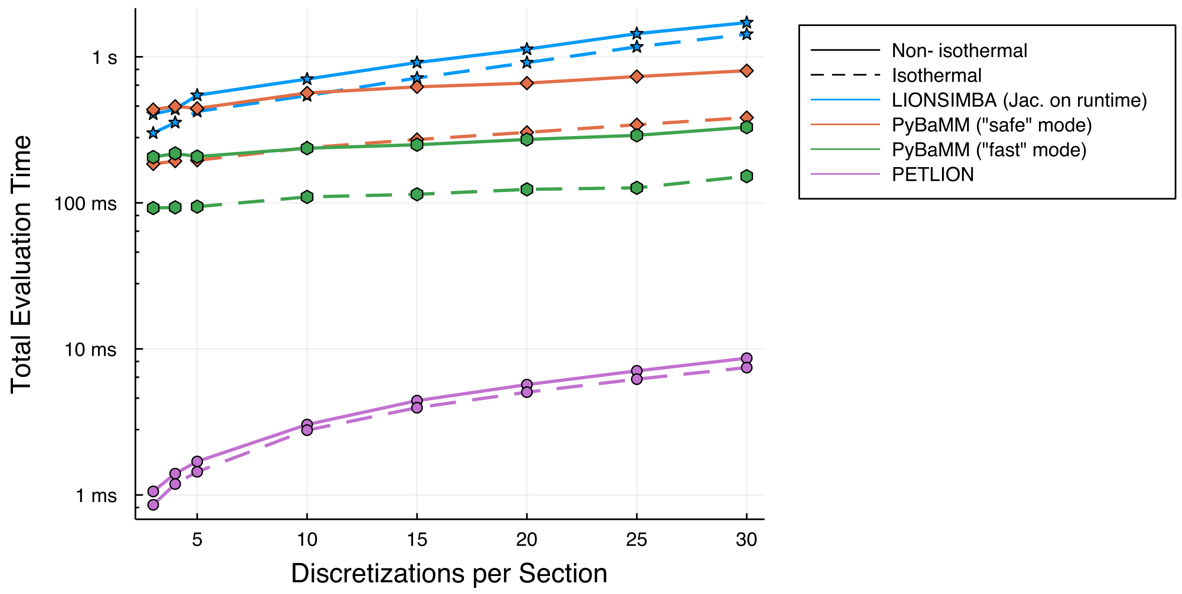


Figure 9. Model runtime of PETLION, LIONSIMBA, and PyBaMM averaged over 10 linearly spaced C-rates between −2C to 2C. Fickian diffusion, Sundials, and Newton's method for semi-explicit DAE initialization (PETLION only) are implemented. Absolute and relative tolerances are 10^{-6} and 10^{-3} respectively.

this slowdown. PETLION and LIONSIMBA also use an integration scheme comparable to "safe" by default (which also contributes to slowness in LIONSIMBA). PETLION has an option to disable checks, however since Julia is compiled, disabling or enabling stop condition checks does not have a noticeable effect on PETLION's runtime. For this example, PETLION is 38× and 76× faster than PyBaMM's "fast" and "safe" modes respectively.

The total evaluation time for PETLION, LIONSIMBA, and PyBaMM in a parameter estimation setting were evaluated for numerous discretizations (see Fig. 9). At its coarsest discretization with 91 DAEs, PETLION is capable of evaluating a full-order model isothermally in under 1 ms. Even the non-isothermal fine-mesh discretization at $N = 30$ (1,011 DAEs) is comparable to the runtimes of reformulated models in the literature.[6,37,38] Polynomial approximations to the solid surface concentration (which are optionally available) reduce this runtime even further to a few hundred μs, although with reduced accuracy at higher C-rates. [39]

On average for isothermal/non-isothermal conditions, PETLION is 206×/229× faster than LIONSIMBA, 35x/68x faster for PyBaMM's "fast" mode, and 80×/159× faster for PyBaMM's "safe" mode. In PETLION and LIONSIMBA, the non-isothermal speeds are roughly 1.1–1.3× slower than the isothermal speeds at the same discretization, but PyBaMM modes show >2× slowdowns for the same comparison. The full model runs of PETLION only allocate memory on the order of single megabytes in large part due to the extremely efficient residuals and Jacobian functions. This substantial increase in the computational efficiency would directly translate into similar increases when the PET model is incorporated into parameter estimation, SOC and SOH estimation, model predictive control, and other algorithms.

One potential source of the speed differences of LIONSIMBA and PyBaMM is the language. MATLAB and Python are primarily interpreted languages compared to Julia which is compiled. Interpreted languages can be more flexible and simple to use than compiled languages but have performance penalties. MATLAB can have efficient performance, particularly with its A\b linear solver and when using vector and/or matrix operations, but certain aspects of the language can act as bottlenecks. MATLAB shares arrays between functions using *pass-by-value*, meaning copies of arrays are created and stored separately in memory. Julia and Python use the more memory-efficient *pass-by-reference* which shares the pointer to these arrays in memory and is less expensive. Additionally, "for" loops in interpreted languages can cause significant overhead since the contents of the "for" loop must be reinterpreted upon each iteration. While a common strategy to speed up a MATLAB code is to use matrix algebra in place of "for" loops, iteratively solving stiff DAEs with an unknown number of steps requires at least one "for" loop so "for" loops cannot be completely avoided in MATLAB and Python. As Julia is a compiled language, "for" loops do not incur the same performance penalty seen in MATLAB and Python.

Conclusions

PETLION is highly computationally efficient battery modeling software developed in the Julia programming language for high-performance computing capabilities. The code in PETLION was validated against LIONSIMBA with relative errors $<10^{-5}$ and can run a constant current simulation of a full-order model with 301 DAEs in 3 ms on a laptop computer with about 1 MB of memory allocation. The software is illustrated for several examples, including CC-CV, GITT, and nonlinear model predictive control. The residuals and Jacobian generated in this model are highly performant functions which both contribute to negligible memory allocations per iteration. Several stiff DAE solvers available in Julia were compared over many discretizations of the PET model, and by far the best for this problem is the Sundials IDA with KLU linear solver. DAE initialization was investigated for speed and robustness, and Newton's method was found to be the fastest method while ensuring convergence of the model for all conditions.

PETLION is open-source and freely available to download on GitHub.[25] With such high increases in model runtime efficiency, PETLION reduces the costs of performing PET-based systems engineering calculations, which can be used for developing both faster and more sophisticated algorithms. Further improvements, such as forward and/or adjoint sensitivity analyses, will only enhance its performance for complex calculations.

Acknowledgments

This work was supported by the Toyota Research Institute through the D3BATT Center on Data-Driven-Design of Rechargeable Batteries. The authors thank Richard B. Canty for designing the PETLION logo.

Appendix

Table A·I. Number of differential, algebraic, and total equations in the PET model with Fickian diffusion.

Mode	Equations	General Number of Discretization Points	Simplified Formula[a)]
Differential	$N_p + N_s + N_n + N_pN_{r,p} + N_nN_{r,n}$	$23N$	
Isothermal	Algebraic	$3N_p + N_s + 3N_n + 1$	$7N + 1$
Total	$4N_p + 2N_s + 4N_n + N_pN_{r,p} + N_nN_{r,n} + 1$	$30N + 1$	
Differential	$2N_p + 2N_s + 2N_n + N_a + N_z + N_pN_{r,p} + N_nN_{r,n}$	$26N + 20$	
Non-isothermal	Algebraic	$3N_p + N_s + 3N_n + 1$	$7N + 1$
Total	$5N_p + 3N_s + 5N_n + N_a + N_z + N_pN_{r,p} + N_nN_{r,n} + 1$	$33N + 21$	

a) This formula is the number of discretization points when the discretizations satisfy A·1.

DAE Discretization.—There are up to 7 sections of the battery which are independently discretized in the *x*- and *r*-directions. For isothermal simulations, the number of discretization points in the *x*-direction are N_p for the cathode, N_s for the separator, and N_n for the anode. For non-isothermal simulations, N_a discretization points are needed for the positive current collector and N_z for the negative current collector. When Fickian diffusion is enabled, the number of discretizations in the *r*-direction are $N_{r,p}$ for the cathode solid particles and $N_{r,n}$ for the anode solid particles. The software allows any values over 2 for these numbers. In the simulations in this article, the number of discretization points were set to:

$$N_p = N_s = N_n = N,$$
$$N_a = N_z = N_{r,p} = N_{r,n} = 10. \qquad [A·1]$$

ORCID

Marc D. Berliner https://orcid.org/0000-0002-2511-1853
Daniel A. Cogswell https://orcid.org/0000-0001-8027-9635
Martin Z. Bazant https://orcid.org/0000-0002-8200-4501
Richard D. Braatz https://orcid.org/0000-0003-4304-3484

References

1. P. V. Kamat, *ACS Energy Lett.*, **4**, 2757 (2019).
2. IEA, *Global EV Outlook 2019—Analysis*. [Online]. Available: https://www.iea.org/reports/global-ev-outlook-2019..
3. M. A. Hannan, M. M. Hoque, A. Mohamed, and A. Ayob, *Renew. Sustain. Energy Rev.*, **69**, 771 (2017).
4. M. Doyle, T. F. Fuller, and J. Newman, *J. Electrochem. Soc.*, **140**, 1526 (1993).
5. J. Newman and W. Tiedemann, *AIChE J.*, **21**, 25 (1975).
6. P. W. C. Northrop, B. Suthar, V. Ramadesigan, S. Santhanagopalan, R. D. Braatz, and V. R. Subramanian, *J. Electrochem. Soc.*, **161**, E3149 (2014).
7. V. R. Subramanian, V. Boovaragavan, V. Ramadesigan, and M. Arabandi, *J. Electrochem. Soc.*, **156**, A260 (2009).
8. M. Torchio, L. Magni, R. B. Gopaluni, R. D. Braatz, and D. M. Raimondo, *J. Electrochem. Soc.*, **163**, A1192 (2016).
9. T. F. Fuller, M. Doyle, and J. Newman, *J. Electrochem. Soc.*, **141**, 982 (1994).
10. T. F. Fuller, M. Doyle, and J. Newman, *J. Electrochem. Soc.*, **141**, 1 (1994).
11. M. C. Henstridge, E. Laborda, N. V. Rees, and R. G. Compton, *Electrochimica Acta*, **84**, 12 (2012).
12. K. W. Baek, E. S. Hong, and S. W. Cha, *International Journal of Automotive Technology*, **16**, 309 (2015).
13. C. M. Doyle, "Design and Simulation of Lithium Rechargeable Batteries." *Ph.D. dissertation*, University of California, Berkeley (1995).
14. W. Fang, O. J. Kwon, and C.-Y. Wang, *International Journal of Energy Research*, **34**, 107 (2010).
15. W. B. Gu and C. Y. Wang, *J. Electrochem. Soc.*, **147**, 2910 (2000).
16. J. Unger, A. Kröner, and W. Marquardt, *Computers & Chemical Engineering*, **19**, 867 (1995).
17. J. E. Cuthrell and L. T. Biegler, *AIChE J.*, **33**, 1257 (1987).
18. Newman, J., *FORTRAN Programs for Simulation of Electrochemical Systems: Dualfoil* (1998), http://www.cchem.berkeley.edu/jsngrp/fortran.html.
19. L. Cai and R. E. White, *Journal of Power Sources*, **196**, 5985 (2011).
20. J. Newman, *Industrial & Engineering Chemistry Fundamentals*, **7**, 514 (1968).
21. V. Sulzer, S. G. Marquis, R. Timms, M. Robinson, and S. J. Chapman, *Journal of Open Research Software*, **9**, 14 (2021).
22. K. Shah, A. Subramaniam, L. Mishra, T. Jang, M. Z. Bazant, R. D. Braatz, and V. R. Subramanian, *J. Electrochem. Soc.*, **167**, 133501 (2020).
23. W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython* (O'Reilly Media, Newton, Massachusetts) (2012).
24. S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture* (Addison-Wesley, Boston, Massachusetts) (2002).
25. M. D. Berliner and R. D. Braatz, *PETLION*, 2021. [Online]. Available: https://github.com/MarcBerliner/PETLION.jl..
26. D. W. Dees, S. Kawauchi, D. P. Abraham, and J. Prakash, *Journal of Power Sources*, **189**, 263 (2009).
27. S. Kolluri, S. V. Aduru, M. Pathak, R. D. Braatz, and V. R. Subramanian, *J. Electrochem. Soc.*, **167**, 063505 (2020).
28. A. Pozzi, M. Torchio, R. D. Braatz, and D. M. Raimondo, *Journal of Power Sources*, **461**, 228133 (2020).
29. C. Rackauckas and Q. Nie, *Journal of Open Research Software*, **5**, 15 (2017).
30. J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, *Mathematical Programming Computation*, **11**, 1 (2019).
31. J. Revels, M. Lubin, and T. Papamarkou, (2016), arXiv:1607.07892.
32. V. Boovaragavan and V. R. Subramanian, *Electrochemistry Communications*, **9**, 1772 (2007).
33. A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust Region Methods* (SIAM, New York, NY) (2000).
34. C. T. Kelley, *Solving Nonlinear Equations with Newton's Method* (SIAM, New York, NY) (2003).
35. H. F. Walker and P. Ni, *SIAM Journal on Numerical Analysis*, **49**, 1715 (2011).
36. A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, *ACM Transactions on Mathematical Software (TOMS)*, **31**, 363 (2005).
37. V. Boovaragavan, S. Harinipriya, and V. R. Subramanian, *Journal of Power Sources*, **183**, 361 (2008).
38. P. W. C. Northrop, V. Ramadesigan, S. De, and V. R. Subramanian, *J. Electrochem. Soc.*, **158**, A1461 (2011).
39. V. R. Subramanian, V. D. Diwakar, and D. Tapriyal, *J. Electrochem. Soc.*, **152**, A2002 (2005).