PyBaMM Documentation

Release 25.1.1

The PyBaMM Team

Jan 21, 2025

CONTENTS

CHAPTER

ONE

PYBAMM USER GUIDE

This guide is an overview and explains the important features; details are found in *API documentation*.

1.1 Installation

PyBaMM is available on GNU/Linux, MacOS and Windows. It can be installed using `pip` or `conda`, or from source.

PyBaMM can be installed via pip from PyPI.

```
pip install pybamm
```

PyBaMM is available as a `conda` package through the conda-forge channel.

The `pybamm` package on conda-forge installs PyBaMM with all the *required and optional dependencies* available on conda-forge.

```
conda install -c conda-forge pybamm

  The ``pybamm-base`` package installs PyBaMM only with its `required dependencies <
↪#install-required-dependencies>`_.
```

```
conda install -c conda-forge pybamm-base
```

1.1.1 Optional solvers

The following solvers are optionally available:

- jax -based solver, see *Optional - JaxSolver* .
- IREE (MLIR) support, see *Optional - IREE / MLIR support*.

1.1.2 Dependencies

PyBaMM requires the following dependencies:

Warning

The list of dependencies below might be outdated. Users are advised to manually check the pyproject.toml file to find out supported versions.

Required dependencies

PyBaMM requires the following dependencies.

Package	Supported version(s)
PyBaMM solvers	0.0.4
NumPy	>= 1.23.5, <2
SciPy	Whatever recent versions work. >= 1.9.3
CasADi	Whatever recent versions work. >= 3.6.7
Xarray	Whatever recent versions work. >= 2022.6.0
Anytree	Whatever recent versions work. >= 2.8.0
SymPy	Whatever recent versions work. >= 1.9.3
typing-extensions	Whatever recent versions work. >= 4.10.0
pandas	Whatever recent versions work. >= 1.5.0
pooch	Whatever recent versions work. >= 1.8.1
posthog	Whatever recent versions work. >= 3.6.5
pyyaml	
platformdirs	

Optional Dependencies

PyBaMM has a number of optional dependencies for different functionalities. If the optional dependency is not installed, PyBaMM will raise an ImportError when the method requiring that dependency is called.

If you are using `pip`, optional PyBaMM dependencies can be installed or managed in a file (e.g., setup.py, or pyproject.toml) as optional extras (e.g.,``pybamm[dev,plot]``). All optional dependencies can be installed with `pybamm[all]`, and specific sets of dependencies are listed in the sections below.

Plot dependencies

Installable with `pip install "pybamm[plot]"`

Dependency	Minimum Version	pip extra	Notes
imageio	2.3.0	plot	For generating simulation GIFs.
matplotlib	3.6.0	plot	To plot various battery models, and analyzing battery performance.

Docs dependencies

Installable with `pip install "pybamm[docs]"`

Dependency	Minimum Version	pip extra	Notes
sphinx	-	docs	Sphinx makes it easy to create intelligent and beautiful documentation.
sphinx_rtd_theme	-	docs	This Sphinx theme provides a great reader experience for documentation.
pydata-sphinx-theme	-	docs	A clean, Bootstrap-based Sphinx theme.
sphinx_design	-	docs	A sphinx extension for designing.
sphinx-copybutton	-	docs	To copy codeblocks.
myst-parser	-	docs	For technical & scientific documentation.
sphinx-inline-tabs	-	docs	Add inline tabbed content to your Sphinx documentation.
sphinxcontrib-bibtex	-	docs	For BibTeX citations.
sphinx-autobuild	-	docs	For re-building docs once triggered.
sphinx-last-updated-by-git	-	docs	To get the "last updated" time for each Sphinx page from Git.
nbsphinx	-	docs	Sphinx extension that provides a source parser for .ipynb files
ipykernel	-	docs	Provides the IPython kernel for Jupyter.
ipywidgets	-	docs	Interactive HTML widgets for Jupyter notebooks and the IPython kernel.
sphinx-gallery	-	docs	Builds an HTML gallery of examples from any set of Python scripts.
sphinx-hoverxref	-	docs	Sphinx extension to show a floating window.
sphinx-docsearch	-	docs	To replaces Sphinx's built-in search with Algolia DocSearch.

Examples dependencies

Installable with `pip install "pybamm[examples]"`

Dependency	Minimum Version	pip extra	Notes
jupyter	-	examples	For example notebooks rendering.

Dev dependencies

Installable with `pip install "pybamm[dev]"`

Dependency	Minimum Version	pip extra	Notes
pre-commit	-	dev	For managing and maintaining multi-language pre-commit hooks.
ruff	-	dev	For code formatting.
nox	-	dev	For running testing sessions in multiple environments.
pytest-subtests	-	dev	For subtests pytest fixture.
pytest-cov	-	dev	For calculating test coverage.
pytest	6.0.0	dev	For running the test suites.
pytest-doctestplus	-	dev	For running doctests.
pytest-xdist	-	dev	For running tests in parallel across distributed workers.
pytest-mock	-	dev	Provides a mocker fixture.
nbmake	-	dev	A `pytest` plugin for executing Jupyter notebooks.
importlib-metadata	-	dev	Used to read metadata from Python packages.

Cite dependencies

Installable with `pip install "pybamm[cite]"`

Dependency	Minimum Version	pip extra	Notes
pybtex	0.24.0	cite	BibTeX-compatible bibliography processor.

bpx dependencies

Installable with `pip install "pybamm[bpx]"`

Dependency	Minimum Version	pip extra	Notes
bpx	-	bpx	Battery Parameter eXchange

tqdm dependencies

Installable with `pip install "pybamm[tqdm]"`

Dependency	Minimum Version	pip extra	Notes
tqdm	-	tqdm	For logging loops.

Jax dependencies

Installable with `pip install "pybamm[jax]"`, currently supported on Python 3.9-3.11.

Dependency	Minimum Version	pip extra	Notes
JAX	0.4.20	jax	For the JAX solver
jaxlib	0.4.20	jax	Support library for JAX

IREE dependencies

Installable with `pip install "pybamm[iree]"` (requires `jax` dependencies to be installed).

Dependency	Minimum Version	pip extra	Notes
iree-compiler	20240507.886	iree	IREE compiler

1.1.3 Full installation guide

Installing a specific version? Installing from source? Check the advanced installation pages below

GNU/Linux & macOS

Contents

Prerequisites

To use PyBaMM, you must have Python 3.9, 3.10, 3.11, or 3.12 installed.

To install Python 3 on Debian-based distributions (Debian, Ubuntu), open a terminal and run

```
sudo apt-get update
sudo apt-get install python3
```

On macOS, you can use the `homebrew` package manager. First, install brew:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)
↪"
```

then follow instructions in the link on adding `brew` to path, and run

```
brew install python
```

Install PyBaMM

User install

We recommend to install PyBaMM within a virtual environment, in order not to alter any distribution Python files. First, make sure you are using Python 3.9, 3.10, 3.11, or 3.12. To create a virtual environment `env` within your current directory type:

```
virtualenv env
```

or use any of your preferred environment management tools. You can then "activate" the environment using:

```
source env/bin/activate
```

Now all the calls to pip described below will install PyBaMM and its dependencies into the environment `env`. When you are ready to exit the environment and go back to your original system, just type:

```
deactivate
```

PyBaMM can be installed via `pip` or `conda`. library beforehand.

```
pip install pybamm
```

```
conda install -c conda-forge pybamm-base
```

PyBaMM's *required dependencies*

(such as `numpy`, `casadi`, etc) will be installed automatically when you install `pybamm` using `pip` or `pybamm-base` using `conda`.

For an introduction to virtual environments, see (https://realpython.com/python-virtual-environments-a-primer/).

Optional - JaxSolver

Users can install `jax` and `jaxlib` to use the Jax solver.

```
pip install "pybamm[jax]"
```

The `pip install "pybamm[jax]"` command automatically downloads and installs `pybamm` and the compatible versions of `jax` and `jaxlib` on your system.

PyBaMM's full *conda-forge distribution* (`pybamm`) includes `jax` and `jaxlib` by default.

Optional - IREE / MLIR support

Users can install `iree` (for MLIR just-in-time compilation) to use for main expression evaluation in the IDAKLU solver. Requires `jax`.

```
pip install "pybamm[iree,jax]"
```

The `pip install "pybamm[iree,jax]"` command automatically downloads and installs `pybamm` and the compatible versions of `jax` and `iree` onto your system.

Uninstall PyBaMM

PyBaMM can be uninstalled by running

```
pip uninstall pybamm
```

in your virtual environment.

Windows

Contents

Prerequisites

To use PyBaMM, you must have Python 3.9, 3.10, 3.11, or 3.12 installed.

To install Python 3 download the installation files from Python's website. Make sure to tick the box on `Add Python 3.X to PATH`. For more detailed instructions please see the official Python on Windows guide.

Install PyBaMM

User install

Launch the Command Prompt and go to the directory where you want to install PyBaMM. You can find a reminder of how to navigate the terminal here.

We recommend to install PyBaMM within a virtual environment, in order not to alter any distribution python files.

To install `virtualenv`, type:

```
python -m pip install virtualenv
```

To create a virtual environment `env` within your current directory type:

```
python -m virtualenv env
```

or use any of your preferred environment management tools. You can then "activate" the environment using:

```
env\Scripts\activate.bat
```

Now all the calls to pip described below will install PyBaMM and its dependencies into the environment `env`. When you are ready to exit the environment and go back to your original system, just type:

```
deactivate
```

PyBaMM can be installed via pip:

```
pip install pybamm
```

```
conda install -c conda-forge pybamm-base
```

PyBaMM's *required dependencies*

(such as `numpy`, `casadi`, etc) will be installed automatically when you install `pybamm` using `pip` or `pybamm-base` using `conda`.

For an introduction to virtual environments, see (https://realpython.com/python-virtual-environments-a-primer/).

Optional - JaxSolver

Users can install `jax` and `jaxlib` to use the Jax solver.

```
pip install "pybamm[jax]"
```

The `pip install "pybamm[jax]"` command automatically downloads and installs `pybamm` and the compatible versions of `jax` and `jaxlib` on your system.

PyBaMM's full *conda-forge distribution* (`pybamm`) includes `jax` and `jaxlib` by default.

Uninstall PyBaMM

PyBaMM can be uninstalled by running

```
pip uninstall pybamm
```

in your virtual environment.

Installation using WSL

If you want to install the optional PyBaMM solvers, you have to use the Windows Subsystem for Linux (WSL). You can find the installation instructions on the *Installation* page.

Install from source (Windows Subsystem for Linux)

To make it easier to install PyBaMM, we recommend using the Windows Subsystem for Linux (WSL) along with Visual Studio Code. This guide will walk you through the process.

Install WSL

Install Ubuntu 22.04 or 20.04 LTS as a distribution for WSL following Microsoft's guide to install WSL. For a seamless development environment, refer to this guide.

Install PyBaMM

Get PyBaMM's Source Code

1. Open a terminal in your Ubuntu distribution by selecting "Ubuntu" from the Start menu. You'll get a bash prompt in your home directory.
2. Install Git by typing the following command:

```
sudo apt install git-core
```

3. Clone the PyBaMM repository:

```
git clone https://github.com/pybamm-team/PyBaMM.git
```

4. Enter the PyBaMM Directory by running:

```
cd PyBaMM
```

5. Follow the Installation Steps

Follow the *installation instructions for PyBaMM on Linux*.

Using Visual Studio Code with the WSL

To use Visual Studio Code with the Windows Subsystem for Linux (WSL), follow these steps:

1. Open Visual Studio Code.
2. Install the "Remote - WSL" extension if not already installed.
3. Open the PyBaMM directory in Visual Studio Code.
4. In the bottom pane, select the "+" sign and choose "New WSL Window."
5. This opens a WSL terminal in the PyBaMM directory within the WSL.

Now you can develop and edit PyBaMM code using Visual Studio Code while utilizing the WSL environment.

Install from source (GNU Linux and macOS)

Contents

This page describes the build and installation of PyBaMM from the source code, available on GitHub. Note that this is **not the recommended approach for most users** and should be reserved to people wanting to participate in the development of PyBaMM, or people who really need to use bleeding-edge feature(s) not yet available in the latest released version. If you do not fall in the two previous categories, you would be better off installing PyBaMM using pip or conda.

Lastly, familiarity with the Python ecosystem is recommended (pip, virtualenvs). Here is a gentle introduction/refresher: Python Virtual Environments: A Primer.

Prerequisites

The following instructions are valid for both GNU/Linux distributions and MacOS. If you are running Windows, consider using the Windows Subsystem for Linux (WSL).

To obtain the PyBaMM source code, clone the GitHub repository

```
git clone https://github.com/pybamm-team/PyBaMM.git
```

or download the source archive on the repository's homepage.

To install PyBaMM, you will need:

- Python 3 (PyBaMM supports versions 3.9, 3.10, 3.11, and 3.12)
- The Python headers file for your current Python version.
- A BLAS library (for instance openblas).
- A C compiler (ex: `gcc`).
- A Fortran compiler (ex: `gfortran`).
- `graphviz` (optional), if you wish to build the documentation locally.
- `pandoc` (optional) to convert the example Jupyter notebooks when building the documentation.

You can install the above with

```
sudo apt install python3.X python3.X-dev libopenblas-dev gcc gfortran graphviz cmake␣
↪pandoc
```

Where `X` is the version sub-number.

```
brew install python openblas gcc gfortran graphviz libomp cmake pandoc
```

Note

If you are using some other linux distribution you can install the equivalent packages for `python3, cmake, gcc, gfortran, openblas, pandoc`.

On Windows, you can install `graphviz` using the Chocolatey package manager, or follow the instructions on the graphviz website.

Finally, we recommend using Nox. You can install it to your local user account (make sure you are not within a virtual environment) with

```
python3.X -m pip install --user nox
```

Note that running `nox` will create new virtual environments for you to use, so you do not need to create one yourself.

Depending on your operating system, you may or may not have `pip` installed along Python. If `pip` is not found, you probably want to install the `python3-pip` package.

Installing PyBaMM

You should now have everything ready to build and install PyBaMM successfully.

Using Nox (recommended)

```
# in the PyBaMM/ directory
nox -s dev
```

> **Note**
>
> It is recommended to use `--verbose` or `-v` to see outputs of all commands run.

This creates a virtual environment `venv/` inside the `PyBaMM/` directory. It comes ready with PyBaMM and some useful development tools like pre-commit and ruff.

You can now activate the environment with

```
source venv/bin/activate
```

```
venv\Scripts\activate.bat
```

and run the tests to check your installation.

Manual install

From the `PyBaMM/` directory, you can install PyBaMM using

```
pip install .
```

If you intend to contribute to the development of PyBaMM, it is convenient to install in "editable mode", along with all the optional dependencies and useful tools for development and documentation:

```
pip install -e .[all,dev,docs]
```

If you are using `zsh` or `tcsh`, you would need to use different pattern matching:

```
pip install -e '.[all,dev,docs]'
```

Before you start contributing to PyBaMM, please read the contributing guidelines.

Running the tests

Using Nox (recommended)

You can use `Nox` to run the unit tests and example notebooks in isolated virtual environments.

The default command

```
nox
```

will run pre-commit, install `Linux` and `macOS` dependencies, and run the unit tests. This can take several minutes.

To just run the unit tests, use

```
nox -s unit
```

Similarly, to run the integration tests, use

```
nox -s integration
```

Finally, to run the unit and the integration suites sequentially, use

```
nox -s tests
```

Using pytest

You can run unit tests for PyBaMM using

```
# in the PyBaMM/ directory
pytest -m unit
```

The above uses pytest (in your current Python environment) to run the unit tests. This can take a few minutes.

You can also use pytest to run the doctests:

```
pytest --doctest-plus src
```

Refer to the testing docs to find out more ways to test PyBaMM using pytest.

How to build the PyBaMM documentation

The documentation is built using

```
nox -s docs
```

This will build the documentation and serve it locally (thanks to sphinx-autobuild) for preview. The preview will be updated automatically following changes.

Doctests, examples, and coverage

`Nox` can also be used to run doctests, run examples, and generate a coverage report using:

- `nox -s examples`: Run the Jupyter notebooks in `docs/source/examples/notebooks/`.
- `nox -s examples -- <path-to-notebook-1.ipynb> <path-to_notebook-2.ipynb>`: Run specific Jupyter notebooks.
- `nox -s scripts`: Run the example scripts in `examples/scripts/`.
- `nox -s doctests`: Run doctests.
- `nox -s coverage`: Measure current test coverage and generate a coverage report.
- `nox -s quick`: Run integration tests, unit tests, and doctests sequentially.

Extra tips while using `Nox`

Here are some additional useful commands you can run with `Nox`:

- `--verbose or -v`: Enables verbose mode, providing more detailed output during the execution of Nox sessions.
- `--list or -l`: Lists all available Nox sessions and their descriptions.
- `--stop-on-first-error`: Stops the execution of Nox sessions immediately after the first error or failure occurs.

- `--envdir <path>`: Specifies the directory where Nox creates and manages the virtual environments used by the sessions. In this case, the directory is set to `<path>`.
- `--install-only`: Skips the test execution and only performs the installation step defined in the Nox sessions.
- `--nocolor`: Disables the color output in the console during the execution of Nox sessions.
- `--report output.json`: Generates a JSON report of the Nox session execution and saves it to the specified file, in this case, "output.json".
- `nox -s docs --non-interactive`: Builds the documentation without serving it locally (using `sphinx-build` instead of `sphinx-autobuild`).

Troubleshooting

Problem: I have made edits to source files in PyBaMM, but these are not being used when I run my Python script.

Solution: Make sure you have installed PyBaMM using the `-e` flag, i.e. `pip install -e .`. This sets the installed location of the source files to your current directory.

Install from source (Docker)

Contents

This page describes the build and installation of PyBaMM using a Dockerfile, available on GitHub. Note that this is **not the recommended approach for most users** and should be reserved to people wanting to participate in the development of PyBaMM, or people who really need to use bleeding-edge feature(s) not yet available in the latest released version. If you do not fall in the two previous categories, you would be better off installing PyBaMM using `pip` or `conda`.

Prerequisites

Before you begin, make sure you have Docker installed on your system. You can download and install Docker from the official Docker website. Ensure Docker installation by running:

```
docker --version
```

Pulling the Docker image

Use the following command to pull the PyBaMM Docker image from Docker Hub:

```
docker pull pybamm/pybamm
```

Running the Docker container

Once you have pulled the Docker image, you can run a Docker container with the PyBaMM environment:

1. In your terminal, use the following command to start a Docker container from the pulled image:

```
docker run -it pybamm/pybamm
```

2. You will now be inside the Docker container's shell. You can use PyBaMM and its dependencies as if you were in a virtual environment.
3. You can execute PyBaMM-related commands, run tests develop & contribute from the container.

> **Note**
>
> The default user for the container is `pybamm` with `pybamm` as password. The user belongs to `sudoers` and `root` group, so the sudo command can be issued to install additional packages to the container. After a clean install, `sudo apt-get update` should be executed to update the source list. Additional packages can be installed using `sudo apt-get install [package_name]`.

Exiting the Docker container

To exit the Docker container's shell, you can simply type:

```
exit
```

This will return you to your host machine's terminal.

Building Docker image locally from source

If you want to build the PyBaMM Docker image locally from the PyBaMM source code, follow these steps:

1. Clone the PyBaMM GitHub repository to your local machine if you haven't already:

```
git clone https://github.com/pybamm-team/PyBaMM.git
```

2. Change into the PyBaMM directory:

```
cd PyBaMM
```

3. Build the Docker image using the following command:

```
docker build -t pybamm -f scripts/Dockerfile .
```

4. Once the image is built, you can run a Docker container using:

```
docker run -it pybamm
```

5. Activate PyBaMM development virtual environment inside docker container using:

```
source /home/pybamm/venv/bin/activate
```

Using Git inside a running Docker container

> **Note**
>
> You might require re-configuring git while running the docker container for the first time. You can run `git config --list` to ensure if you have desired git configuration already.

1. Setting up git configuration

```
git config --global user.name "Your Name"

git config --global user.email your@mail.com
```

2. Setting a git remote

```
git remote set-url origin <fork_url>

git remote add upstream https://github.com/pybamm-team/PyBaMM

git fetch --all
```

Using Visual Studio Code inside a running Docker container

You can easily use Visual Studio Code inside a running Docker container by attaching it directly. This provides a seamless development environment within the container. Here's how:

1. Install the "Docker" extension from Microsoft in your local Visual Studio Code if it's not already installed.
2. Pull and run the Docker image containing PyBaMM development environment.
3. In your local Visual Studio Code, open the "Docker" extension by clicking on the Docker icon in the sidebar.
4. Under the "Containers" section, you'll see a list of running containers. Right-click the running PyBaMM container.
5. Select "Attach Visual Studio Code" from the context menu.
6. Visual Studio Code will now connect to the container, and a new VS Code window will open up, running inside the container. You can now edit, debug, and work on your code using VS Code as if you were working directly on your local machine.

1.2 Getting Started

The easiest way to use PyBaMM is to run a 1C constant-current discharge with a model of your choice with all the default settings:

```
import pybamm

model = pybamm.lithium_ion.DFN()  # Doyle-Fuller-Newman model
sim = pybamm.Simulation(model)
sim.solve([0, 3600])  # solve for 1 hour
sim.plot()
```

or simulate an experiment such as a constant-current discharge followed by a constant-current-constant-voltage charge:

```
import pybamm

experiment = pybamm.Experiment(
    [
        (
            "Discharge at C/10 for 10 hours or until 3.3 V",
            "Rest for 1 hour",
            "Charge at 1 A until 4.1 V",
            "Hold at 4.1 V until 50 mA",
            "Rest for 1 hour",
        )
    ]
    * 3,
)
model = pybamm.lithium_ion.DFN()
sim = pybamm.Simulation(model, experiment=experiment, solver=pybamm.CasadiSolver())
sim.solve()
sim.plot()
```

However, much greater customisation is available. It is possible to change the physics, parameter values, geometry, submesh type, number of submesh points, methods for spatial discretisation and solver for integration (see DFN script or notebook).

For new users we recommend the Getting Started guides. These are intended to be very simple step-by-step guides to show the basic functionality of PyBaMM, and can either be downloaded and used locally, or used online through Google Colab.

Further details can be found in a number of detailed examples, hosted on GitHub. In addition, full details of classes and methods can be found in the *API documentation*. Additional supporting material can be found here.

1.3 Fundamentals

PyBaMM (Python Battery Mathematical Modelling) is an open-source battery simulation package written in Python. Our mission is to accelerate battery modelling research by providing open-source tools for multi-institutional, interdisciplinary collaboration. Broadly, PyBaMM consists of

1. a framework for writing and solving systems of differential equations,
2. a library of battery models and parameters, and
3. specialized tools for simulating battery-specific experiments and visualizing the results.

Together, these enable flexible model definitions and fast battery simulations, allowing users to explore the effect of different battery designs and modeling assumptions under a variety of operating scenarios.

> **NOTE**: This user-guide is a work-in-progress, we hope that this brief but incomplete overview will be useful to you.

1.3.1 Core framework

The core of the framework is a custom computer algebra system to define mathematical equations, and a domain specific modeling language to combine these equations into systems of differential equations (usually partial differential equations for variables depending on space and time). The expression tree example gives an introduction to the computer algebra system, and the Getting Started tutorials walk through creating models of increasing complexity.

Once a model has been defined symbolically, PyBaMM solves it using the Method of Lines. First, the equations are discretised in the spatial dimension, using the finite volume method. Then, the resulting system is solved using third-party numerical solvers. Depending on the form of the model, the system can be ordinary differential equations (ODEs) (if only `model.rhs` is defined), or algebraic equations (if only `model.algebraic` is defined), or differential-algebraic equations (DAEs) (if both `model.rhs` and `model.algebraic` are defined). Jupyter notebooks explaining the solvers can be found here.

1.3.2 Model and Parameter Library

PyBaMM contains an extensive library of battery models and parameters. The bulk of the library consists of models for lithium-ion, but there are also some other chemistries (lead-acid, lithium metal). Models are first divided broadly into common named models of varying complexity, such as the single particle model (SPM) or Doyle-Fuller-Newman model (DFN). Most options can be applied to any model, but some are model-specific (an error will be raised if you attempt to set an option is not compatible with a model). See *Base Battery Model* for a list of options.

The parameter library is simply a collection of python files each defining a complete set of parameters for a particular battery chemistry, covering all major lithium-ion chemistries (NMC, LFP, NCA, ...). External parameter sets can be linked using entry points (see *Parameters Sets*).

1.3.3 Battery-specific tools

One of PyBaMM's unique features is the `Experiment` class, which allows users to define synthetic experiments using simple instructions in English

```
pybamm.Experiment(
    [
        (
            "Discharge at C/10 for 10 hours or until 3.3 V",
            "Rest for 1 hour",
            "Charge at 1 A until 4.1 V",
            "Hold at 4.1 V until 50 mA",
            "Rest for 1 hour",
        )
    ]
    * 3,
)
```

The above instruction will conduct a standard discharge / rest / charge / rest cycle three times, with a 10 hour discharge and 1 hour rest at the end of each cycle.

The `Simulation` class handles simulating an `Experiment`, as well as calculating additional outputs such as capacity as a function of cycle number. For example, the following code will simulate the experiment above and plot the standard output variables:

```
import pybamm
import matplotlib.pyplot as plt

# load model and parameter values
model = pybamm.lithium_ion.DFN()
sim = pybamm.Simulation(model, experiment=experiment)
solution = sim.solve()
solution.plot()
```

Finally, PyBaMM provides custom visualization tools:

- *Quick Plot*: for easily plotting simulation outputs in a grid, including comparing multiple simulations

- *pybamm.plot_voltage_components*: for plotting the component overpotentials that make up a voltage curve

Users are not limited to these tools and can plot the output of a simulation solution by accessing the underlying numpy array for the solution variables as

```
solution["variable name"].data
```

and using the plotting library of their choice.

1.4 Battery Models

References for the battery models used in PyBaMM simulations can be found calling

```
pybamm.print_citations()
```

However, a few papers are provided in this section for anyone interested in reading the theory behind the models before doing the tutorials.

1.4.1 Review Articles

Review of physics-based lithium-ion battery models

Review of parameterisation and a novel database for Li-ion battery models

1.4.2 Model References

Lithium-Ion Batteries

Doyle-Fuller-Newman model

Single particle model

Lead-Acid Batteries

Isothermal porous-electrode model

Leading-Order Quasi-Static model

1.5 Public API

PyBaMM is a Python package for mathematical modelling and simulation of battery systems. The main classes and functions that are intended to be used by the user are described in this document. For a more detailed description of the classes and methods, see the *API reference*.

1.5.1 Available PyBaMM models

PyBaMM includes a number of pre-implemented models, which can be used as they are or modified to suit your needs. The main models are:

- *lithium_ion.SPM*: Single Particle Model
- *lithium_ion.SPMe*: Single Particle Model with Electrolyte
- *lithium_ion.DFN*: Doyle-Fuller-Newman

The behaviour of the models can be modified by passing in an *BatteryModelOptions* object when creating the model.

1.5.2 Simulations

Simulation is a class that automates the process of setting up a model and solving it, and acts as the highest-level API to PyBaMM. Pass at least a *BaseModel* object, and optionally the experiment, solver, parameter values, and geometry objects described below to the *Simulation* object. Any of these optional arguments not provided will be supplied by the defaults specified in the model.

1.5.3 Parameters

PyBaMM models are parameterised by a set of parameters, which are stored in a *ParameterValues* object. This object acts like a Python dictionary with a few extra PyBaMM specific features and methods. Parameters in a model are represented as either *Parameter* objects or *FunctionParameter* objects, and the values in the *ParameterValues* object replace these objects in the model before it is solved. The values in the *ParameterValues* object can be scalars, Python functions or expressions of type *Symbol*.

1.5.4 Experiments

An *Experiment* object represents an experimental protocol that can be used to simulate the behaviour of a battery. The particular protocol can be provided as a Python string, or as a sequences of *step.BaseStep* objects.

1.5.5 Solvers

The two main solvers in PyBaMM are the *CasadiSolver* and the *IDAKLUSolver*. Both are wrappers around the Sundials suite of solvers, but the *CasadiSolver* uses the CasADi library whereas the *IDAKLUSolver* is PyBaMM specific. Both solvers have many options that can be set to control the solver behaviour, see the documentation for each solver for more details.

When a model is solved, the solution is returned as a *Solution* object.

1.5.6 Plotting

A solution object can be plotted using the *Solution.plot()* or *Simulation.plot()* methods, which returns a *QuickPlot* object. Note that the arguments to the plotting methods of both classes are the same as *QuickPlot*.

Other plotting functions are the *plot_voltage_components()* and *plot_summary_variables()* functions, which correspond to the similarly named methods of the *Solution* and *Simulation* classes.

1.5.7 Writing PyBaMM models

Each PyBaMM model, and the custom models written by users, are written as a set of expressions that describe the model. Each of the expressions is a subclass of the *Symbol* class, which represents a mathematical expression.

If you wish to create a custom model, you can use the *BaseModel* class as a starting point.

1.5.8 Discretisation

Each PyBaMM model contains continuous operators that must be discretised before they can be solved. This is done using a *Discretisation* object, which takes a *Mesh* object and a dictionary of *SpatialMethod* objects.

1.5.9 Logging

PyBaMM uses the Python logging module to log messages at different levels of severity. Use the *pybamm.set_logging_level()* function to set the logging level for PyBaMM.

1.6 Contributing to PyBaMM

If you'd like to contribute to PyBaMM (thanks!), please have a look at the *guidelines below*.

If you're already familiar with our workflow, maybe have a quick look at the *pre-commit checks* directly below.

1.6.1 Pre-commit checks

Before you commit any code, please perform the following checks:

- *All tests pass*: `$ nox -s unit`
- *The documentation builds*: `$ nox -s docs`

Installing and using pre-commit

`PyBaMM` uses a set of `pre-commit` hooks and the `pre-commit` bot to format and prettify the codebase. The hooks can be installed locally using -

```
pip install pre-commit
pre-commit install
```

This would run the checks every time a commit is created locally. The checks will only run on the files modified by that commit, but the checks can be triggered for all the files using -

```
pre-commit run --all-files
```

If you would like to skip the failing checks and push the code for further discussion, use the `--no-verify` option with `git commit`.

1.6.2 Workflow

We use GIT and GitHub to coordinate our work. When making any kind of update, we try to follow the procedure below.

A. Before you begin

1. Create an issue where new proposals can be discussed before any coding is done.
2. Create a branch of this repo (ideally on your own fork), where all changes will be made
3. Download the source code onto your local system, by cloning the repository (or your fork of the repository).
4. Install PyBaMM with the developer options.
5. *Test* if your installation worked, using pytest: `$ pytest -m unit`.

You now have everything you need to start making changes!

B. Writing your code

6. PyBaMM is developed in Python, and makes heavy use of NumPy.
7. Make sure to follow our *coding style guidelines*.
8. Commit your changes to your branch with useful, descriptive commit messages: Remember these are publicly visible and should still make sense a few months ahead in time. While developing, you can keep using the GitHub issue you're working on as a place for discussion.
9. If you want to add a dependency on another library, or re-use code you found somewhere else, have a look at *these guidelines*.

C. Merging your changes with PyBaMM

10. *Test your code!*
11. PyBaMM has online documentation at http://docs.pybamm.org/. To make sure any new methods or classes you added show up there, please read the *documentation* section.
12. If you added a major new feature, perhaps it should be showcased in an *example notebook*.
13. When you feel your code is finished, or at least warrants serious discussion, run the *pre-commit checks* and then create a pull request (PR) on PyBaMM's GitHub page.
14. Once a PR has been created, it will be reviewed by any member of the community. Changes might be suggested which you can make by simply adding new commits to the branch. When everything's finished, someone with the right GitHub permissions will merge your changes into PyBaMM main repository.

Finally, if you really, really, *really* love developing PyBaMM, have a look at the current *project infrastructure*.

1.6.3 Coding style guidelines

PyBaMM follows the PEP8 recommendations for coding style. These are very common guidelines, and community tools have been developed to check how well projects implement them. We recommend using pre-commit hooks to check your code before committing it. See *installing and using pre-commit* section for more details.

Ruff

We use ruff to check our PEP8 adherence. To try this on your system, navigate to the PyBaMM directory in a console and type

```
python -m pip install pre-commit
pre-commit run ruff
```

ruff is configured inside the file `pre-commit-config.yaml`, allowing us to ignore some errors. If you think this should be added or removed, please submit an issue

When you commit your changes they will be checked against ruff automatically (see *Pre-commit checks*).

Naming

Naming is hard. In general, we aim for descriptive class, method, and argument names. Avoid abbreviations when possible without making names overly long, so `mean` is better than `mu`, but a class name like `MyClass` is fine.

Class names are CamelCase, and start with an upper case letter, for example `MyOtherClass`. Method and variable names are lower case, and use underscores for word separation, for example `x` or `iteration_count`.

1.6.4 Dependencies and reusing code

While it's a bad idea for developers to "reinvent the wheel", it's important for users to get a *reasonably sized download and an easy install*. In addition, external libraries can sometimes cease to be supported, and when they contain bugs it might take a while before fixes become available as automatic downloads to PyBaMM users. For these reasons, all dependencies in PyBaMM should be thought about carefully, and discussed on GitHub.

Direct inclusion of code from other packages is possible, as long as their license permits it and is compatible with ours, but again should be considered carefully and discussed in the group. Snippets from blogs and stackoverflow can often be included without attribution, but if they solve a particularly nasty problem (or are very hard to read) it's often a good idea to attribute (and document) them, by making a comment with a link in the source code.

Separating dependencies

On the other hand… We *do* want to compare several tools, to generate documentation, and to speed up development. For this reason, the dependency structure is split into 4 parts:

1. Core PyBaMM: A minimal set, including things like NumPy, SciPy, etc. All infrastructure should run against this set of dependencies, as well as any numerical methods we implement ourselves.
2. Extras: Other inference packages and their dependencies. Methods we don't want to implement ourselves, but do want to provide an interface to can have their dependencies added here.
3. Documentation generating code: Everything you need to generate and work on the docs.
4. Development code: Everything you need to do PyBaMM development (so all of the above packages, plus ruff and other testing tools).

Only 'core pybamm' is installed by default. The others have to be specified explicitly when running the installation command.

Managing Optional Dependencies and Their Imports

PyBaMM utilizes optional dependencies to allow users to choose which additional libraries they want to use. Managing these optional dependencies and their imports is essential to provide flexibility to PyBaMM users.

PyBaMM provides a utility function `import_optional_dependency`, to check for the availability of optional dependencies within methods. This function can be used to conditionally import optional dependencies only if they are available. Here's how to use it:

Optional dependencies should never be imported at the module level, but always inside methods. For example:

```
def use_pybtex(x, y, z):
    pybtex = import_optional_dependency("pybtex")
    ...
```

While importing a specific module instead of an entire package/library:

```
def use_parse_file(x, y, z):
    parse_file = import_optional_dependency("pybtex.database", "parse_file")
    ...
```

This allows people to (1) use PyBaMM without importing optional dependencies by default and (2) configure module-dependent functionalities in their scripts, which *must* be done before e.g. `print_citations` method is first imported.

Writing Tests for Optional Dependencies

Below, we list the currently available test functions to provide an overview. If you find it useful to add new test cases please do so within `tests/unit/test_util.py`.

Currently, there are three functions to test what concerns optional dependencies:

- `test_import_optional_dependency`
- `test_pybamm_import`
- `test_optional_dependencies`

The `test_import_optional_dependency` function extracts the optional dependencies installed in the setup environment, makes them unimportable (by setting them to `None` among the `sys.modules`), and tests that the `pybamm.util.import_optional_dependency` function throws a `ModuleNotFoundError` exception when their import is attempted.

The `test_pybamm_import` function extracts the optional dependencies installed in the setup environment and makes them unimportable (by setting them to `None` among the `sys.modules`), unloads `pybamm` and its sub-modules, and

finally tests that `pybamm` can be imported successfully. In fact, it is essential that the `pybamm` package is importable with only the mandatory dependencies.

The `test_optional_dependencies` function extracts `pybamm` mandatory distribution packages and verifies that they are not present in the optional distribution packages list in `pyproject.toml`. This test is crucial for ensuring the consistency of the released package information and potential updates to dependencies during development.

1.6.5 Testing

All code requires testing. We use the pytest package for our tests. (These tests typically just check that the code runs without error, and so, are more *debugging* than *testing* in a strict sense. Nevertheless, they are very useful to have!)

We use following plugins for various needs:

nbmake : plugins to test the example notebooks.

pytest-xdist : plugins to run tests in parallel.

If you have `nox` installed, to run unit tests, type

```
nox -s unit
```

else, type

```
pytest -m unit
```

Writing tests

Every new feature should have its own test. To create ones, have a look at the `test` directory and see if there's a test for a similar method. Copy-pasting this is a good way to start.

Next, add some simple (and speedy!) tests of your main features. If these run without exceptions that's a good start! Next, check the output of your methods using assert statements.

Running more tests

The tests are divided into `unit` tests, whose aim is to check individual bits of code (e.g. discretising a gradient operator, or solving a simple ODE), and `integration` tests, which check how parts of the program interact as a whole (e.g. solving a full model). If you want to check integration tests as well as unit tests, type

```
nox -s tests
```

or, alternatively, you can use posargs to pass the path to the test to `nox`. For example:

```
nox -s tests -- tests/unit/test_plotting/test_quick_plot.py::TestQuickPlot::test_simple_
↪ode_model
```

When you commit anything to PyBaMM, these checks will also be run automatically (see *infrastructure*).

Testing the example notebooks

To test all the example notebooks in the `docs/source/examples/` folder with `pytest` and `nbmake`, type

```
nox -s examples
```

Alternatively, you may use `pytest` directly with the `--nbmake` flag:

```
pytest --nbmake docs/source/examples/
```

which runs all the notebooks in the `docs/source/examples/notebooks/` folder in parallel by default, using the `pytest-xdist` plugin.

Sometimes, debugging a notebook can be a hassle. To run a single notebook, pass the path to it to `pytest`:

```
pytest --nbmake docs/source/examples/notebooks/notebook-name.ipynb
```

or, alternatively, you can use posargs to pass the path to the notebook to `nox`. For example:

```
nox -s examples -- docs/source/examples/notebooks/notebook-name.ipynb
```

You may also test multiple notebooks this way. Passing the path to a folder will run all the notebooks in that folder:

```
nox -s examples -- docs/source/examples/notebooks/models/
```

You may also use an appropriate glob pattern to run all notebooks matching a particular folder or name pattern.

To edit the structure and how the Jupyter notebooks get rendered in the Sphinx documentation (using `nbsphinx`), install Pandoc on your system, either using `conda` (through the `conda-forge` channel)

```
conda install -c conda-forge pandoc
```

or refer to the Pandoc installation instructions specific to your platform.

Testing the example scripts

To test all the example scripts in the `examples/` folder, type

```
nox -s scripts
```

Debugging

Often, the code you write won't pass the tests straight away, at which stage it will become necessary to debug. The key to successful debugging is to isolate the problem by finding the smallest possible example that causes the bug. In practice, there are a few tricks to help you to do this, which we give below. Once you've isolated the issue, it's a good idea to add a unit test that replicates this issue, so that you can easily check whether it's been fixed, and make sure that it's easily picked up if it crops up again. This also means that, if you can't fix the bug yourself, it will be much easier to ask for help (by opening a bug-report issue).

1. Run individual test scripts instead of the whole test suite:

   ```
   pytest tests/unit/path/to/test
   ```

   You can also run an individual test from a particular script, e.g.

   ```
   pytest tests/unit/test_plotting/test_quick_plot.py::TestQuickPlot::test_simple_ode_
   ↪model
   ```

   If you want to run several, but not all, the tests from a script, you can restrict which tests are run from a particular script by using the skipping decorator:

   ```
   @pytest.mark.skip("")
   def test_bit_of_code(self):
       ...
   ```

   or by just commenting out all the tests you don't want to run.

2. Set break points, either in your IDE or using the Python debugging module. To use the latter, add the following line where you want to set the break point

```
import ipdb

ipdb.set_trace()
```

This will start the Python interactive debugger. If you want to be able to use magic commands from `ipython`, such as `%timeit`, then set

```
from IPython import embed

embed()
import ipdb

ipdb.set_trace()
```

at the break point instead. Figuring out where to start the debugger is the real challenge. Some good ways to set debugging break points are:

1. Try-except blocks. Suppose the line `do_something_complicated()` is raising a `ValueError`. Then you can put a try-except block around that line as:

```
try:
    do_something_complicated()
except ValueError:
    import ipdb

    ipdb.set_trace()
```

This will start the debugger at the point where the `ValueError` was raised, and allow you to investigate further. Sometimes, it is more informative to put the try-except block further up the call stack than exactly where the error is raised.

2. Warnings. If functions are raising warnings instead of errors, it can be hard to pinpoint where this is coming from. Here, you can use the `warnings` module to convert warnings to errors:

```
import warnings

warnings.simplefilter("error")
```

Then you can use a try-except block, as in a., but with, for example, `RuntimeWarning` instead of `ValueError`.

3. Stepping through the expression tree. Most calls in PyBaMM are operations on expression trees. To view an expression tree in ipython, you can use the `render` command:

```
expression_tree.render()
```

You can then step through the expression tree, using the `children` attribute, to pinpoint exactly where a bug is coming from. For example, if `expression_tree.jac(y)` is failing, you can check `expression_tree.children[0].jac(y)`, then `expression_tree.children[0].children[0].jac(y)`, etc.

3. To isolate whether a bug is in a model, its Jacobian or its simplified version, you can set the `use_jacobian` and/or `use_simplify` attributes of the model to `False` (they are both `True` by default for most models).

4. If a model isn't giving the answer you expect, you can try comparing it to other models. For example, you can investigate parameter limits in which two models should give the same answer by setting some parameters to be small or zero. The `StandardOutputComparison` class can be used to compare some standard outputs from battery models.
5. To get more information about what is going on under the hood, and hence understand what is causing the bug, you can set the logging level to `DEBUG` by adding the following line to your test or script:

```
pybamm.set_logging_level("DEBUG")
```

6. In models that inherit from `pybamm.BaseBatteryModel` (i.e. any battery model), you can use `self.process_parameters_and_discretise` to process a symbol and see what it will look like.

Profiling

Sometimes, a bit of code will take much longer than you expect to run. In this case, you can set

```
from IPython import embed

embed()
import ipdb

ipdb.set_trace()
```

as above, and then use some of the profiling tools. In order of increasing detail:

1. Simple timer. In ipython, the command

```
%time command_to_time()
```

   tells you how long the line `command_to_time()` takes. You can use `%timeit` instead to run the command several times and obtain more accurate timings.

2. Simple profiler. Using `%prun` instead of `%time` will give a brief profiling report 3. Detailed profiler. You can install the detailed profiler `snakeviz` through pip:

```
pip install snakeviz
```

   and then, in ipython, run

```
%load_ext snakeviz
%snakeviz command_to_time()
```

   This will open a window in your browser with detailed profiling information.

1.6.6 Documentation

PyBaMM is documented in several ways.

First and foremost, every method and every class should have a docstring that describes in plain terms what it does, and what the expected input and output is.

These docstrings can be fairly simple, but can also make use of reStructuredText, a markup language designed specifically for writing technical documentation. For example, you can link to other classes and methods by writing ``:class:`pybamm.Model`` `` and ``:meth:`run()` ``.

In addition, we write a (very) small bit of documentation in separate reStructuredText files in the `docs` directory. Most of what these files do is simply import docstrings from the source code. But they also do things like add tables and

indexes. If you've added a new class to a module, search the `docs` directory for that module's `.rst` file and add your class (in alphabetical order) to its index. If you've added a whole new module, copy-paste another module's file and add a link to your new file in the appropriate `index.rst` file.

Using Sphinx the documentation in `docs` can be converted to HTML, PDF, and other formats. In particular, we use it to generate the documentation on http://docs.pybamm.org/

Building the documentation

To test and debug the documentation, it's best to build it locally. To do this, navigate to your PyBaMM directory in a console, and then type (on GNU/Linux, macOS, and Windows):

```
nox -s docs
```

And then visit the webpage served at `http://127.0.0.1:8000`. Each time a change to the documentation source is detected, the HTML is rebuilt and the browser automatically reloaded. In CI, the docs are built and tested using the `docs` session in the `noxfile.py` file with warnings turned into errors, to fail the build. The warnings can be removed or ignored by adding the appropriate warning identifier to the `suppress_warnings` list in `docs/conf.py`.

Example notebooks

Major PyBaMM features are showcased in Jupyter notebooks stored in the docs/source/examples directory. Which features are "major" is of course wholly subjective, so please discuss on GitHub first!

All example notebooks should be listed in docs/source/examples/index.rst. Please follow the (naming and writing) style of existing notebooks where possible.

All the notebooks are tested daily.

1.6.7 Citations

We aim to recognize all contributions by automatically generating citations to the relevant papers on which different parts of the code are built. These will change depending on what models and solvers you use. Adding the command

```
pybamm.print_citations()
```

to the end of a script will print all citations that were used by that script. This will print BibTeX information to the terminal; passing a filename to `print_citations` will print the BibTeX information to the specified file instead.

When you contribute code to PyBaMM, you can add your own papers that you would like to be cited if that code is used. First, add the BibTeX for your paper to CITATIONS.bib. Then, add the line

```
pybamm.citations.register("your_paper_bibtex_identifier")
```

wherever code is called that uses that citation (for example, in functions or in the `__init__` method of a class such as a model or solver).

1.6.8 Infrastructure

Installation

Installation of PyBaMM and its dependencies is handled via pip

Configuration files:

```
pyproject.toml
```

Continuous Integration using GitHub Actions

Each change pushed to the PyBaMM GitHub repository will trigger the test and benchmark suites to be run, using GitHub Actions.

Tests are run for different operating systems, and for all Python versions officially supported by PyBaMM. If you opened a Pull Request, feedback is directly available on the corresponding page. If all tests pass, a green tick will be displayed next to the corresponding test run. If one or more test(s) fail, a red cross will be displayed instead.

Similarly, the benchmark suite is automatically run for the most recently pushed commit. Benchmark results are compared to the results available for the latest commit on the `develop` branch. Should any significant performance regression be found, a red cross will be displayed next to the benchmark run.

In all cases, more details can be obtained by clicking on a specific run.

Configuration files for various GitHub actions workflow can be found in `.github/worklfows`.

Codecov

Code coverage (how much of our code is actually seen by the (linux) unit tests) is tested using Codecov, a report is visible on https://codecov.io/gh/pybamm-team/PyBaMM.

Configuration files:

```
.coveragerc
```

Read the Docs

Documentation is built using https://readthedocs.org/ and published on http://docs.pybamm.org/.

Google Colab

Editable notebooks are made available using Google Colab here.

GitHub

GitHub does some magic with particular filenames. In particular:

- The first page people see when they go to our GitHub page displays the contents of README.md, which is written in the Markdown format. Some guidelines can be found here.
- The license for using PyBaMM is stored in LICENSE, and automatically linked to by GitHub.
- This file, CONTRIBUTING.md is recognised as the contribution guidelines and a link is automatically displayed when new issues or pull requests are created.

1.6.9 Acknowledgements

This CONTRIBUTING.md file, along with large sections of the code infrastructure, was copied from the excellent Pints GitHub repo

CHAPTER

TWO

EXAMPLE NOTEBOOKS

PyBaMM ships with example notebooks that demonstrate how to use it and reveal some of its functionalities and its inner workings. For more examples, see the Examples section.

The notebooks are not included in PDF formats of the documentation. You may access them on PyBaMM's hosted documentation available at https://docs.pybamm.org/en/latest/source/examples/index.html

CHAPTER

THREE

TELEMETRY

PyBaMM optionally collects anonymous usage data to help improve the library. This telemetry is opt-in and can be easily disabled. Here's what you need to know:

- **What is collected**: Basic usage information like PyBaMM version, Python version, and which functions are run.
- **Why**: To understand how PyBaMM is used and prioritize development efforts.
- **Opt-out**: To disable telemetry, set the environment variable `PYBAMM_DISABLE_TELEMETRY=true` (or any value other than `false`) or use `pybamm.telemetry.disable()` in your code.
- **Privacy**: No personal information (name, email, etc) or sensitive information (parameter values, simulation results, etc) is ever collected.

CHAPTER

FOUR

API DOCUMENTATION

Release
25.1.1

Date
Jan 21, 2025

This reference manual details the classes, functions, modules, and objects included in PyBaMM, describing what they are and what they do. For a high-level introduction to PyBaMM, see the *user guide* and the examples.

4.1 Expression Tree

4.1.1 Symbol

pybamm.**simplify_if_constant**(*symbol:* Symbol)

Utility function to simplify an expression tree if it evalutes to a constant scalar, vector or matrix

class pybamm.**Symbol**(*name: str*, *children: Sequence[*Symbol*] | None = None*, *domain: DomainType = None*, *auxiliary_domains: AuxiliaryDomainType = None*, *domains: DomainsType = None*)

Base node class for the expression tree.

Parameters

- **name** (*str*) – name for the node
- **children** (iterable *Symbol*, optional) – children to attach to this node, default to an empty list
- **domain** (*iterable of str, or str*) – list of domains over which the node is valid (empty list indicates the symbol is valid over all domains)
- **auxiliary_domains** (*dict of str*) – dictionary of auxiliary domains over which the node is valid (empty dictionary indicates no auxiliary domains). Keys can be "secondary", "tertiary" or "quaternary". The symbol is broadcast over its auxiliary domains. For example, a symbol might have domain "negative particle", secondary domain "separator" and tertiary domain "current collector" (*domain="negative particle", auxiliary_domains={"secondary": "separator", "tertiary": "current collector"}*).
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.

__abs__() → *AbsoluteValue*

return an *AbsoluteValue* object, or a smooth approximation.

__add__(*other: ChildSymbol*) → *pybamm.Addition*

return an *Addition* object.

__array_ufunc__(*ufunc*, *method*, **inputs*, ***kwargs*)

If a numpy ufunc is applied to a symbol, call the corresponding pybamm function instead.

__eq__(*other*)

Return self==value.

__ge__(*other:* Symbol) → *EqualHeaviside*

return a *EqualHeaviside* object, or a smooth approximation.

__gt__(*other:* Symbol) → *NotEqualHeaviside*

return a *NotEqualHeaviside* object, or a smooth approximation.

__hash__()

Return hash(self).

__init__(*name: str*, *children: Sequence[*Symbol*]* | *None* = *None*, *domain: DomainType* = *None*, *auxiliary_domains: AuxiliaryDomainType* = *None*, *domains: DomainsType* = *None*)

__le__(*other:* Symbol) → *EqualHeaviside*

return a *EqualHeaviside* object, or a smooth approximation.

__lt__(*other:* Symbol | *float*) → *NotEqualHeaviside*

return a *NotEqualHeaviside* object, or a smooth approximation.

__matmul__(*other: ChildSymbol*) → *pybamm.MatrixMultiplication*

return a *MatrixMultiplication* object.

__mod__(*other:* Symbol) → *Modulo*

return an *Modulo* object.

__mul__(*other: ChildSymbol*) → *pybamm.Multiplication*

return a *Multiplication* object.

__neg__() → *Negate*

return a *Negate* object.

__pow__(*other: ChildSymbol*) → *pybamm.Power*

return a *Power* object.

__radd__(*other: ChildSymbol*) → *pybamm.Addition*

return an *Addition* object.

__repr__()

returns the string *__class__(id, name, children, domain)*

__rmatmul__(*other: ChildSymbol*) → *pybamm.MatrixMultiplication*

return a *MatrixMultiplication* object.

__rmul__(*other: ChildSymbol*) → *pybamm.Multiplication*

return a *Multiplication* object.

__rpow__(*other:* Symbol) → *Power*

return a *Power* object.

__rsub__(*other: ChildSymbol*) → *pybamm.Subtraction*

return a *Subtraction* object.

__rtruediv__(*other: ChildSymbol*) → *pybamm.Division*

return a *Division* object.

__str__()

return a string representation of the node and its children.

__sub__(*other: ChildSymbol*) → *pybamm.Subtraction*

return a *Subtraction* object.

__truediv__(*other: ChildSymbol*) → *pybamm.Division*

return a *Division* object.

__weakref__

list of weak references to the object

property auxiliary_domains

Returns auxiliary domains.

property children

returns the cached children of this node.

Note: it is assumed that children of a node are not modified after initial creation

clear_domains()

Clear domains, bypassing checks.

copy_domains(*symbol:* Symbol)

Copy the domains from a given symbol, bypassing checks.

create_copy(*new_children: list[*Symbol*]* | *None* = *None*, *perform_simplifications: bool* = *True*)

Make a new copy of a symbol, to avoid Tree corruption errors while bypassing copy.deepcopy(), which is slow.

If new_children are provided, they are used instead of the existing children.

If *perform_simplifications* = True, some classes (e.g. *BinaryOperator*, *UnaryOperator*, *Concatenation*) will perform simplifications and error checks based on the new children before copying the symbol. This may result in a different symbol being returned than the one copied.

Turning off this behaviour to ensure the symbol remains unchanged is discouraged.

diff(*variable:* Symbol)

Differentiate a symbol with respect to a variable. For any symbol that can be differentiated, return *1* if differentiating with respect to yourself, *self._diff(variable)* if *variable* is in the expression tree of the symbol, and zero otherwise.

Parameters
variable (*pybamm.Symbol*) – The variable with respect to which to differentiate

property domain

list of applicable domains.

Return type
iterable of str

evaluate(*t: float | None = None*, *y: np.ndarray | None = None*, *y_dot: np.ndarray | None = None*, *inputs: dict | str | None = None*) → ChildValue

Evaluate expression tree (wrapper to allow using dict of known values).

Parameters

- **t** (*float or numeric type, optional*) – time at which to evaluate (default None)
- **y** (*numpy.array, optional*) – array with state values to evaluate when solving (default None)
- **y_dot** (*numpy.array, optional*) – array with time derivatives of state values to evaluate when solving (default None)
- **inputs** (*dict, optional*) – dictionary of inputs to use when solving (default None)

Returns
the node evaluated at (t,y)

Return type
number or array

evaluate_for_shape()

Evaluate expression tree to find its shape.

For symbols that cannot be evaluated directly (e.g. *Variable* or *Parameter*), a vector of the appropriate shape is returned instead, using the symbol's domain. See *pybamm.Symbol.evaluate()*

evaluate_ignoring_errors(*t: float | None = 0*)

Evaluates the expression. If a node exists in the tree that cannot be evaluated as a scalar or vector (e.g. Time, Parameter, Variable, StateVector), then None is returned. If there is an InputParameter in the tree then a 1 is returned. Otherwise the result of the evaluation is given.

See also

evaluate
evaluate the expression

evaluates_on_edges(*dimension: str*) → bool

Returns True if a symbol evaluates on an edge, i.e. symbol contains a gradient operator, but not a divergence operator, and is not an IndefiniteIntegral. Caches the solution for faster results.

Parameters
dimension (*str*) – The dimension (primary, secondary, etc) in which to query evaluation on edges

Returns
Whether the symbol evaluates on edges (in the finite volume discretisation sense)

Return type
bool

evaluates_to_number()

Returns True if evaluating the expression returns a number. Returns False otherwise, including if NotImplementedError or TyperError is raised. !Not to be confused with isinstance(self, pybamm.Scalar)!

> **See also**
>
> *evaluate*
> evaluate the expression

get_children_domains(*children: Sequence[Symbol]*)

Combine domains from children, at all levels.

has_symbol_of_classes(*symbol_classes: tuple[type[Symbol], ...] | type[Symbol]*)

Returns True if equation has a term of the class(es) *symbol_class*.

Parameters
symbol_classes (*pybamm class or iterable of classes*) – The classes to test the symbol against

is_constant()

returns true if evaluating the expression is not dependent on *t* or *y* or *inputs*

> **See also**
>
> *evaluate*
> evaluate the expression

jac(*variable: Symbol*, *known_jacs: dict[Symbol, Symbol] | None = None*, *clear_domain=True*)

Differentiate a symbol with respect to a (slice of) a StateVector or StateVectorDot. See *pybamm.Jacobian*.

property name

name of the node.

property ndim_for_testing

Number of dimensions of an object, found by evaluating it with appropriate t and y

property orphans

Returning new copies of the children, with parents removed to avoid corrupting the expression tree internal data

post_order(*filter=None*)

returns an iterable that steps through the tree in post-order fashion.

pre_order()

returns an iterable that steps through the tree in pre-order fashion.

Examples

```
>>> a = pybamm.Symbol('a')
>>> b = pybamm.Symbol('b')
>>> for node in (a*b).pre_order():
...     print(node.name)
*
a
b
```

property quaternary_domain

Helper function to get the quaternary domain of a symbol.

relabel_tree(*symbol:* Symbol, *counter: int*)

Finds all children of a symbol and assigns them a new id so that they can be visualised properly using the graphviz output

render()

Print out a visual representation of the tree (this node and its children)

property secondary_domain

Helper function to get the secondary domain of a symbol.

set_id()

Set the immutable "identity" of a variable (e.g. for identifying y_slices).

Hashing can be slow, so we set the id when we create the node, and hence only need to hash once.

property shape

Shape of an object, found by evaluating it with appropriate t and y.

property shape_for_testing

Shape of an object for cases where it cannot be evaluated directly. If a symbol cannot be evaluated directly (e.g. it is a *Variable* or *Parameter*), it is instead given an arbitrary domain-dependent shape.

property size

Size of an object, found by evaluating it with appropriate t and y

property size_for_testing

Size of an object, based on shape for testing.

property tertiary_domain

Helper function to get the tertiary domain of a symbol.

test_shape()

Check that the discretised self has a pybamm *shape*, i.e. can be evaluated.

Raises
pybamm.ShapeError – If the shape of the object cannot be found

to_casadi(*t: casadi.MX | None = None*, *y: casadi.MX | None = None*, *y_dot: casadi.MX | None = None*, *inputs: dict | None = None*, *casadi_symbols:* Symbol | *None = None*)

Convert the expression tree to a CasADi expression tree. See *pybamm.CasadiConverter*.

to_json()

Method to serialise a Symbol object into JSON.

visualise(*filename: str*)

Produces a .png file of the tree (this node and its children) with the name filename

Parameters
filename (*str*) – filename to output, must end in ".png"

4.1.2 Parameter

class pybamm.**Parameter**(*name: str*)

A node in the expression tree representing a parameter.

This node will be replaced by a *pybamm.Scalar* node

Parameters
name (*str*) – name of the node

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications=True*) → *Parameter*

See pybamm.Symbol.new_copy().

is_constant() → Literal[False]

See *pybamm.Symbol.is_constant()*.

to_equation() → Symbol

Convert the node and its subtree into a SymPy equation.

to_json()

Method to serialise a Symbol object into JSON.

class pybamm.**FunctionParameter**(*name: str*, *inputs: dict[str,* Symbol*]*, *diff_variable:* Symbol | *None* = *None*, *print_name='calculate'*)

A node in the expression tree representing a function parameter.

This node will be replaced by a *pybamm.Function* node if a callable function is passed to the parameter values, and otherwise (in some rarer cases, such as constant current) a *pybamm.Scalar* node.

Parameters

- **name** (*str*) – name of the node
- **inputs** (*dict*) – A dictionary with string keys and *pybamm.Symbol* values representing the function inputs. The string keys should provide a reasonable description of what the input to the function is (e.g. "Electrolyte concentration [mol.m-3]")
- **diff_variable** (*pybamm.Symbol*, optional) – if diff_variable is specified, the FunctionParameter node will be replaced by a *pybamm.Function* and then differentiated with respect to diff_variable. Default is None.
- **print_name** (*str, optional*) – The name to show when printing. Default is 'calculate', in which case the name is calculated using sys._getframe().

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

diff(*variable:* Symbol) → *FunctionParameter*

See *pybamm.Symbol.diff()*.

set_id()

See *pybamm.Symbol.set_id()*

to_equation() → Symbol

Convert the node and its subtree into a SymPy equation.

to_json()

Method to serialise a Symbol object into JSON.

4.1.3 Variable

class pybamm.**Variable**(*name: str*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *bounds: tuple[*Symbol*] | None = None*, *print_name: str | None = None*, *scale: float | Symbol | None = 1*, *reference: float | Symbol | None = 0*)

A node in the expression tree represending a dependent variable.

This node will be discretised by *Discretisation* and converted to a *pybamm.StateVector* node.

Parameters

- **name** (*str*) – name of the node domain : iterable of str, optional list of domains that this variable is valid over
- **auxiliary_domains** (*dict, optional*) – dictionary of auxiliary domains ({'secondary': ..., 'tertiary': ..., 'quaternary': ...}). For example, for the single particle model, the particle concentration would be a Variable with domain 'negative particle' and secondary auxiliary domain 'current collector'. For the DFN, the particle concentration would be a Variable with domain 'negative particle', secondary domain 'negative electrode' and tertiary domain 'current collector'
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.
- **bounds** (*tuple, optional*) – Physical bounds on the variable
- **print_name** (*str, optional*) – The name to use for printing. Default is None, in which case self.name is used.
- **scale** (float or *pybamm.Symbol*, optional) – The scale of the variable, used for scaling the model when solving. The state vector representing this variable will be multiplied by this scale. Default is 1.
- **reference** (float or *pybamm.Symbol*, optional) – The reference value of the variable, used for scaling the model when solving. This value will be added to the state vector representing this variable. Default is 0.

Extends: pybamm.expression_tree.variable.VariableBase

diff(*variable:* Symbol)

Differentiate a symbol with respect to a variable. For any symbol that can be differentiated, return *1* if differentiating with respect to yourself, *self._diff(variable)* if *variable* is in the expression tree of the symbol, and zero otherwise.

Parameters

variable (*pybamm.Symbol*) – The variable with respect to which to differentiate

class pybamm.**VariableDot**(*name: str*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *bounds: tuple[*Symbol*] | None = None*, *print_name: str | None = None*, *scale: float | Symbol | None = 1*, *reference: float | Symbol | None = 0*)

A node in the expression tree represending the time derviative of a dependent variable

This node will be discretised by *Discretisation* and converted to a *pybamm.StateVectorDot* node.

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str*) – list of domains that this variable is valid over

- **auxiliary_domains** (*dict*) – dictionary of auxiliary domains ({'secondary': ..., 'tertiary': ..., 'quaternary': ...}). For example, for the single particle model, the particle concentration would be a Variable with domain 'negative particle' and secondary auxiliary domain 'current collector'. For the DFN, the particle concentration would be a Variable with domain 'negative particle', secondary domain 'negative electrode' and tertiary domain 'current collector'
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.
- **bounds** (*tuple, optional*) – Physical bounds on the variable. Included for compatibility with *VariableBase*, but ignored.
- **print_name** (*str, optional*) – The name to use for printing. Default is None, in which case self.name is used.
- **scale** (float or *pybamm.Symbol*, optional) – The scale of the variable, used for scaling the model when solving. The state vector representing this variable will be multiplied by this scale. Default is 1.
- **reference** (float or *pybamm.Symbol*, optional) – The reference value of the variable, used for scaling the model when solving. This value will be added to the state vector representing this variable. Default is 0.

Extends: `pybamm.expression_tree.variable.VariableBase`

diff(*variable:* Symbol) → *Scalar*

Differentiate a symbol with respect to a variable. For any symbol that can be differentiated, return *1* if differentiating with respect to yourself, *self._diff(variable)* if *variable* is in the expression tree of the symbol, and zero otherwise.

Parameters

variable (*pybamm.Symbol*) – The variable with respect to which to differentiate

get_variable() → *Variable*

return a *Variable* corresponding to this VariableDot

Note: Variable._jac adds a dash to the name of the corresponding VariableDot, so we remove this here

4.1.4 Independent Variable

class pybamm.**IndependentVariable**(*name: str*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*)

A node in the expression tree representing an independent variable.

Used for expressing functions depending on a spatial variable or time

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str*) – list of domains that this variable is valid over
- **auxiliary_domains** (*dict, optional*) – dictionary of auxiliary domains, defaults to empty dict
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

to_equation() → Symbol

Convert the node and its subtree into a SymPy equation.

class pybamm.**Time**

A node in the expression tree representing time.

Extends: *pybamm.expression_tree.independent_variable.IndependentVariable*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

to_equation()

Convert the node and its subtree into a SymPy equation.

class pybamm.**SpatialVariable**(*name: str*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *coord_sys=None*)

A node in the expression tree representing a spatial variable.

Parameters

- **name** (*str*) – name of the node (e.g. "x", "y", "z", "r", "x_n", "x_s", "x_p", "r_n", "r_p")
- **domain** (*iterable of str*) – list of domains that this variable is valid over (e.g. "cartesian", "spherical polar")
- **auxiliary_domains** (*dict, optional*) – dictionary of auxiliary domains, defaults to empty dict
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.

Extends: *pybamm.expression_tree.independent_variable.IndependentVariable*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

pybamm.**t = the independent variable time**

A node in the expression tree representing time.

4.1.5 Scalar

class pybamm.**Scalar**(*value: int | float | number*, *name: str | None = None*)

A node in the expression tree representing a scalar value.

Parameters

- **value** (*numeric*) – the value returned by the node when evaluated
- **name** (*str, optional*) – the name of the node. Defaulted to str(value) if not provided

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

is_constant() → Literal[True]

See *pybamm.Symbol.is_constant()*.

set_id()

See *pybamm.Symbol.set_id()*.

to_equation()

Returns the value returned by the node when evaluated.

to_json()

Method to serialise a Symbol object into JSON.

property value

The value returned by the node when evaluated.

4.1.6 Array

class pybamm.**Array**(*entries: ndarray | list[float] | csr_matrix*, *name: str | None = None*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *entries_string: str | None = None*)

Node in the expression tree that holds an tensor type variable (e.g. `numpy.array`)

Parameters

- **entries** (*numpy.array or list*) – the array associated with the node. If a list is provided, it is converted to a numpy array
- **name** (*str, optional*) – the name of the node
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list
- **auxiliary_domains** (*dict, optional*) – dictionary of auxiliary domains, defaults to empty dict
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.
- **entries_string** (*str*) – String representing the entries (slow to recalculate when copying)

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications: bool = True*)

See `pybamm.Symbol.new_copy()`.

is_constant()

See *pybamm.Symbol.is_constant()*.

property ndim

returns the number of dimensions of the tensor.

set_id()

See *pybamm.Symbol.set_id()*.

property shape

returns the number of entries along each dimension.

to_equation() → ImmutableDenseNDimArray

Returns the value returned by the node when evaluated.

to_json()

Method to serialise an Array object into JSON.

pybamm.**linspace**(*start: float*, *stop: float*, *num: int = 50*, ***kwargs*) → *Array*

Creates a linearly spaced array by calling *numpy.linspace* with keyword arguments 'kwargs'. For a list of 'kwargs' see the numpy linspace documentation

pybamm.**meshgrid**(*x:* Array, *y:* Array, ***kwargs*) → tuple[*Array*, *Array*]

Return coordinate matrices as from coordinate vectors by calling *numpy.meshgrid* with keyword arguments 'kwargs'. For a list of 'kwargs' see the numpy meshgrid documentation

4.1.7 Matrix

class pybamm.**Matrix**(*entries: ndarray | list[float] | csr_matrix*, *name: str | None = None*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *entries_string: str | None = None*)

Node in the expression tree that holds a matrix type (e.g. `numpy.array`)

Extends: *pybamm.expression_tree.array.Array*

4.1.8 Vector

class pybamm.**Vector**(*entries: ndarray | list[float] | matrix*, *name: str | None = None*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *entries_string: str | None = None*)

node in the expression tree that holds a vector type (e.g. `numpy.array`)

Extends: *pybamm.expression_tree.array.Array*

4.1.9 State Vector

class pybamm.**StateVector**(**y_slices: slice*, *name: str | None = None*, *domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *domains: dict[str, list[str] | str] | None = None*, *evaluation_array: list[bool] | None = None*)

Node in the expression tree that holds a slice to read from an external vector type.

Parameters

- **y_slice** (*slice*) – the slice of an external y to read
- **name** (*str, optional*) – the name of the node
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list
- **auxiliary_domains** (*dict of str, optional*) – dictionary of auxiliary domains
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.
- **evaluation_array** (*list, optional*) – List of boolean arrays representing slices. Default is None, in which case the evaluation_array is computed from y_slices.

Extends: `pybamm.expression_tree.state_vector.StateVectorBase`

diff(*variable:* Symbol)

Differentiate a symbol with respect to a variable. For any symbol that can be differentiated, return *1* if differentiating with respect to yourself, *self._diff(variable)* if *variable* is in the expression tree of the symbol, and zero otherwise.

Parameters
variable (*pybamm.Symbol*) – The variable with respect to which to differentiate

class pybamm.**StateVectorDot**(**y_slices: slice, name: str | None = None, domain: list[str] | str | None = None, auxiliary_domains: dict[str, str] | None = None, domains: dict[str, list[str] | str] | None = None, evaluation_array: list[bool] | None = None*)

Node in the expression tree that holds a slice to read from the ydot.

Parameters

- **y_slice** (*slice*) – the slice of an external ydot to read
- **name** (*str, optional*) – the name of the node
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list
- **auxiliary_domains** (*dict of str, optional*) – dictionary of auxiliary domains
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}. Either 'domain' and 'auxiliary_domains', or just 'domains', should be provided (not both). In future, the 'domain' and 'auxiliary_domains' arguments may be deprecated.
- **evaluation_array** (*list, optional*) – List of boolean arrays representing slices. Default is None, in which case the evaluation_array is computed from y_slices.

Extends: pybamm.expression_tree.state_vector.StateVectorBase

diff(*variable:* Symbol)

Differentiate a symbol with respect to a variable. For any symbol that can be differentiated, return *1* if differentiating with respect to yourself, *self._diff(variable)* if *variable* is in the expression tree of the symbol, and zero otherwise.

Parameters
variable (*pybamm.Symbol*) – The variable with respect to which to differentiate

4.1.10 Binary Operators

class pybamm.**BinaryOperator**(*name: str, left_child: float | ndarray | Symbol, right_child: float | ndarray | Symbol*)

A node in the expression tree representing a binary operator (e.g. +, *)

Derived classes will specify the particular operator

Parameters

- **name** (*str*) – name of the node
- **left** (*Symbol* or Number) – lhs child node (converted to *Scalar* if Number)
- **right** (*Symbol* or Number) – rhs child node (converted to *Scalar* if Number)

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children: list[*Symbol*] | None = None, perform_simplifications: bool = True*)

See pybamm.Symbol.new_copy().

evaluate(*t: float | None = None*, *y: ndarray | None = None*, *y_dot: ndarray | None = None*, *inputs: dict | str | None = None*)

See *pybamm.Symbol.evaluate()*.

is_constant()

See *pybamm.Symbol.is_constant()*.

to_equation()

Convert the node and its subtree into a SymPy equation.

to_json()

Method to serialise a BinaryOperator object into JSON.

class pybamm.**Power**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a ** power operator.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Addition**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing an addition operator.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Subtraction**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a subtraction operator.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Multiplication**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a multiplication operator (Hadamard product). Overloads cases where the "*" operator would usually return a matrix multiplication (e.g. scipy.sparse.coo.coo_matrix)

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**MatrixMultiplication**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a matrix multiplication operator.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

diff(*variable*)

See *pybamm.Symbol.diff()*.

class pybamm.**Division**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a division operator.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Inner**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree which represents the inner (or dot) product. This operator should be used to take the inner product of two mathematical vectors (as opposed to the computational vectors arrived at post-discretisation) of the form v = v_x e_x + v_y e_y + v_z e_z where v_x, v_y, v_z are scalars and e_x, e_y, e_z are x-y-z-directional unit vectors. For v and w mathematical vectors, inner product returns v_x * w_x + v_y * w_y + v_z * w_z. In addition, for some spatial discretisations mathematical vector quantities (such as i = grad(phi)) are evaluated on a different part of the grid to mathematical scalars (e.g. for finite volume mathematical scalars are evaluated on the nodes but mathematical vectors are evaluated on cell edges). Therefore, inner also transfers the inner product of the vector onto the scalar part of the grid if required by a particular discretisation.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.expression_tree.binary_operators.**_Heaviside**(*name: str*, *left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A node in the expression tree representing a heaviside step function. This class is semi-private and should not be called directly, use *EqualHeaviside* or *NotEqualHeaviside* instead, or < or <=.

Adding this operation to the rhs or algebraic equations in a model can often cause a discontinuity in the solution. For the specific cases listed below, this will be automatically handled by the solver. In the general case, you can explicitly tell the solver of discontinuities by adding a `Event` object with `EventType` DISCONTINUITY to the model's list of events.

In the case where the Heaviside function is of the form *pybamm.t* < *x*, *pybamm.t* <= *x*, *x* < *pybamm.t*, or *x* <= *pybamm.t*, where *x* is any constant equation, this DISCONTINUITY event will automatically be added by the solver.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

diff(*variable*)

See *pybamm.Symbol.diff()*.

class pybamm.**EqualHeaviside**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A heaviside function with equality (return 1 when left = right)

Extends: *pybamm.expression_tree.binary_operators._Heaviside*

class pybamm.**NotEqualHeaviside**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

A heaviside function without equality (return 0 when left = right)

Extends: *pybamm.expression_tree.binary_operators._Heaviside*

class pybamm.**Modulo**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

Calculates the remainder of an integer division.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Minimum**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

Returns the smaller of two objects.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

class pybamm.**Maximum**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

Returns the greater of two objects.

Extends: *pybamm.expression_tree.binary_operators.BinaryOperator*

pybamm.**minimum**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*) → *Symbol*

Returns the smaller of two objects, possibly with a smoothing approximation. Not to be confused with *pybamm.min()*, which returns min function of child.

pybamm.**maximum**(*left: float | ndarray | Symbol*, *right: float | ndarray | Symbol*)

Returns the larger of two objects, possibly with a smoothing approximation. Not to be confused with *pybamm.max()*, which returns max function of child.

pybamm.**softminus**(*left: Symbol*, *right: Symbol*, *k: float*)

Softminus approximation to the minimum function. k is the smoothing parameter, set by *pybamm.settings.min_max_smoothing*. The recommended value is k=10.

pybamm.**softplus**(*left: Symbol*, *right: Symbol*, *k: float*)

Softplus approximation to the maximum function. k is the smoothing parameter, set by *pybamm.settings.min_max_smoothing*. The recommended value is k=10.

pybamm.**sigmoid**(*left:* Symbol, *right:* Symbol, *k: float*)

Sigmoidal approximation to the heaviside function. k is the smoothing parameter, set by *pybamm.settings.heaviside_smoothing*. The recommended value is k=10. Note that the concept of deciding which side to pick when left=right does not apply for this smooth approximation. When left=right, the value is (left+right)/2.

pybamm.**source**(*left: int | float | number | Symbol*, *right:* Symbol, *boundary=False*)

A convenience function for creating (part of) an expression tree representing a source term. This is necessary for spatial methods where the mass matrix is not the identity (e.g. finite element formulation with piecwise linear basis functions). The left child is the symbol representing the source term and the right child is the symbol of the equation variable (currently, the finite element formulation in PyBaMM assumes all functions are constructed using the same basis, and the matrix here is constructed accoutning for the boundary conditions of the right child). The method returns the matrix-vector product of the mass matrix (adjusted to account for any Dirichlet boundary conditions imposed the the right symbol) and the discretised left symbol.

Parameters

- **left** (*Symbol*, numeric) – The left child node, which represents the expression for the source term.
- **right** (*Symbol*) – The right child node. This is the symbol whose boundary conditions are accounted for in the construction of the mass matrix.
- **boundary** (*bool, optional*) – If True, then the mass matrix should is assembled over the boundary, corresponding to a source term which only acts on the boundary of the domain. If False (default), the matrix is assembled over the entire domain, corresponding to a source term in the bulk.

4.1.11 Unary Operators

class pybamm.**UnaryOperator**(*name: str*, *child:* Symbol, *domains: dict[str, list[str]] | str] | None = None*)

A node in the expression tree representing a unary operator (e.g. '-', grad, div)

Derived classes will specify the particular operator

Parameters

- **name** (*str*) – name of the node
- **child** (*Symbol*) – child node
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}.

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children: list[*Symbol*] | None = None*, *perform_simplifications: bool = True*)

See pybamm.Symbol.new_copy().

evaluate(*t: float | None = None*, *y: ndarray | None = None*, *y_dot: ndarray | None = None*, *inputs: dict | str | None = None*)

See *pybamm.Symbol.evaluate()*.

is_constant()

See *pybamm.Symbol.is_constant()*.

to_equation()

Convert the node and its subtree into a SymPy equation.

class pybamm.**Negate**(*child*)

A node in the expression tree representing a - negation operator.

Extends: *pybamm.expression_tree.unary_operators.UnaryOperator*

class pybamm.**AbsoluteValue**(*child*)

A node in the expression tree representing an *abs* operator.

Extends: *pybamm.expression_tree.unary_operators.UnaryOperator*

diff(*variable*)

See *pybamm.Symbol.diff()*.

class pybamm.**Sign**(*child*)

A node in the expression tree representing a *sign* operator.

Extends: *pybamm.expression_tree.unary_operators.UnaryOperator*

diff(*variable*)

See *pybamm.Symbol.diff()*.

class pybamm.**Index**(*child*, *index*, *name=None*, *check_size=True*)

A node in the expression tree, which stores the index that should be extracted from its child after the child has been evaluated.

Parameters

- **child** (*pybamm.Symbol*) – The symbol of which to take the index
- **index** (*int or slice*) – The index (if int) or indices (if slice) to extract from the symbol
- **name** (*str, optional*) – The name of the symbol
- **check_size** (*bool, optional*) – Whether to check if the slice size exceeds the child size. Default is True. This should always be True when creating a new symbol so that the appropriate check is performed, but should be False for creating a new copy to avoid unnecessarily repeating the check.

Extends: *pybamm.expression_tree.unary_operators.UnaryOperator*

set_id()

See *pybamm.Symbol.set_id()*

to_json()

Method to serialise an Index object into JSON.

class pybamm.**SpatialOperator**(*name: str*, *child:* Symbol, *domains: dict[str, list[str]* | *str]* | *None* = *None*)

A node in the expression tree representing a unary spatial operator (e.g. grad, div)

Derived classes will specify the particular operator

This type of node will be replaced by the *Discretisation* class with a *Matrix*

Parameters

- **name** (*str*) – name of the node
- **child** (*Symbol*) – child node
- **domains** (*dict*) – A dictionary equivalent to {'primary': domain, auxiliary_domains}.

Extends: *pybamm.expression_tree.unary_operators.UnaryOperator*

to_json()

Method to serialise a Symbol object into JSON.

class pybamm.**Gradient**(*child*)

A node in the expression tree representing a grad operator.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**Divergence**(*child*)

A node in the expression tree representing a div operator.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**Laplacian**(*child*)

A node in the expression tree representing a Laplacian operator. This is currently only implemeted in the weak form for finite element formulations.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**GradientSquared**(*child*)

A node in the expression tree representing a the inner product of the grad operator with itself. In particular, this is useful in the finite element formualtion where we only require the (sclar valued) square of the gradient, and not the gradient itself.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**Mass**(*child*)

Returns the mass matrix for a given symbol, accounting for Dirchlet boundary conditions where necessary (e.g. in the finite element formualtion)

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**BoundaryMass**(*child*)

Returns the mass matrix for a given symbol assembled over the boundary of the domain, accounting for Dirchlet boundary conditions where necessary (e.g. in the finite element formualtion)

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**Integral**(*child*, *integration_variable: list*[IndependentVariable] | IndependentVariable)

A node in the expression tree representing an integral operator.

$$I = \int_{u_{min}}^{u_{max}} f(u)\, dq,$$

where $u \in$ domain is a spatial variable, u_{min} and u_{max} are the values of u at the left-hand and right-hand boundaries of the domain respectively, and dq is given by,

$dq = du$ for cartesian coordinates,

$dq = 2\pi u du$ for cylindrical coordinates,

$dq = 4\pi u^2 du$ for spherical coordinates.

Parameters

- **function** (*pybamm.Symbol*) – The function to be integrated (will become self.children[0])
- **integration_variable** (*pybamm.IndependentVariable*) – The variable over which to integrate

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**IndefiniteIntegral**(*child*, *integration_variable*)

A node in the expression tree representing an indefinite integral operator.

$$I = \int_{x_e xtmin}^{x} f(u)\, du$$

where $u \in$ domain which can represent either a spatial or temporal variable.

Parameters

- **function** (*pybamm.Symbol*) – The function to be integrated (will become self.children[0])
- **integration_variable** (*pybamm.IndependentVariable*) – The variable over which to integrate

Extends: pybamm.expression_tree.unary_operators.BaseIndefiniteIntegral

class pybamm.**DefiniteIntegralVector**(*child*, *vector_type='row'*)

A node in the expression tree representing an integral of the basis used for discretisation

$$I = \int_{a}^{b} \psi(x)\, dx,$$

where a and b are the left-hand and right-hand boundaries of the domain respectively and ψ is the basis function.

Parameters

- **variable** (*pybamm.Symbol*) – The variable whose basis will be integrated over the entire domain (will become self.children[0])
- **vector_type** (*str, optional*) – Whether to return a row or column vector (default is row)

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**BoundaryIntegral**(*child*, *region='entire'*)

A node in the expression tree representing an integral operator over the boundary of a domain

$$I = \int_{\partial a} f(u)\, du,$$

where ∂a is the boundary of the domain, and $u \in$ domain boundary.

Parameters

- **function** (*pybamm.Symbol*) – The function to be integrated (will become self.children[0])
- **region** (*str, optional*) – The region of the boundary over which to integrate. If region is *entire* (default) the integration is carried out over the entire boundary. If region is *negative tab* or *positive tab* then the integration is only carried out over the appropriate part of the boundary corresponding to the tab.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**DeltaFunction**(*child*, *side*, *domain*)

Delta function. Currently can only be implemented at the edge of a domain.

Parameters

- **child** (*pybamm.Symbol*) – The variable that sets the strength of the delta function
- **side** (*str*) – Which side of the domain to implement the delta function on

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

evaluate_for_shape()

See pybamm.Symbol.evaluate_for_shape_using_domain()

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**BoundaryOperator**(*name*, *child*, *side*)

A node in the expression tree which gets the boundary value of a variable on its primary domain.

Parameters

- **name** (*str*) – The name of the symbol
- **child** (*pybamm.Symbol*) – The variable whose boundary value to take
- **side** (*str*) – Which side to take the boundary value on ("left" or "right")

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**BoundaryValue**(*child*, *side*)

A node in the expression tree which gets the boundary value of a variable on its primary domain.

Parameters

- **child** (*pybamm.Symbol*) – The variable whose boundary value to take
- **side** (*str*) – Which side to take the boundary value on ("left" or "right")

Extends: *pybamm.expression_tree.unary_operators.BoundaryOperator*

class pybamm.**BoundaryGradient**(*child*, *side*)

A node in the expression tree which gets the boundary flux of a variable on its primary domain.

Parameters

- **child** (*pybamm.Symbol*) – The variable whose boundary flux to take
- **side** (*str*) – Which side to take the boundary flux on ("left" or "right")

Extends: *pybamm.expression_tree.unary_operators.BoundaryOperator*

class pybamm.**EvaluateAt**(*child*, *position*)

A node in the expression tree which evaluates a symbol at a given position in space in its primary domain. Currently this is only implemented for 1D primary domains.

Parameters

- **child** (*pybamm.Symbol*) – The variable to evaluate
- **position** (*pybamm.Symbol*) – The position in space on the symbol's primary domain at which to evaluate the symbol.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

set_id()

See *pybamm.Symbol.set_id()*

class pybamm.**UpwindDownwind**(*name*, *child*)

A node in the expression tree representing an upwinding or downwinding operator. Usually to be used for better stability in convection-dominated equations.

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

class pybamm.**Upwind**(*child*)

Upwinding operator. To be used if flow velocity is positive (left to right).

Extends: *pybamm.expression_tree.unary_operators.UpwindDownwind*

class pybamm.**Downwind**(*child*)

Downwinding operator. To be used if flow velocity is negative (right to left).

Extends: *pybamm.expression_tree.unary_operators.UpwindDownwind*

pybamm.**grad**(*symbol*)

convenience function for creating a *Gradient*

Parameters
symbol (*Symbol*) – the gradient will be performed on this sub-symbol

Returns
the gradient of `symbol`

Return type
Gradient

pybamm.**div**(*symbol*)

convenience function for creating a *Divergence*

Parameters
symbol (*Symbol*) – the divergence will be performed on this sub-symbol

Returns
the divergence of `symbol`

Return type
Divergence

pybamm.**laplacian**(*symbol*)

convenience function for creating a *Laplacian*

Parameters
symbol (*Symbol*) – the Laplacian will be performed on this sub-symbol

Returns
the Laplacian of `symbol`

Return type
Laplacian

pybamm.**grad_squared**(*symbol*)

convenience function for creating a *GradientSquared*

Parameters
symbol (*Symbol*) – the inner product of the gradient with itself will be performed on this sub-symbol

Returns
inner product of the gradient of `symbol` with itself

Return type
GradientSquared

pybamm.**surf**(*symbol*)

convenience function for creating a right *BoundaryValue*, usually in the spherical geometry.

Parameters
symbol (*pybamm.Symbol*) – the surface value of this symbol will be returned

Returns
the surface value of `symbol`

Return type
pybamm.BoundaryValue

pybamm.**x_average**(*symbol:* Symbol) → *Symbol*

Convenience function for creating an average in the x-direction.

Parameters
symbol (*pybamm.Symbol*) – The function to be averaged

Returns
the new averaged symbol

Return type
Symbol

pybamm.**r_average**(*symbol:* Symbol) → *Symbol*

Convenience function for creating an average in the r-direction.

Parameters
symbol (*pybamm.Symbol*) – The function to be averaged

Returns
the new averaged symbol

Return type
Symbol

pybamm.**size_average**(*symbol:* Symbol, *f_a_dist:* Symbol | *None* = *None*) → *Symbol*

Convenience function for averaging over particle size R using the area-weighted particle-size distribution.

Parameters
symbol (*pybamm.Symbol*) – The function to be averaged

Returns
the new averaged symbol

Return type
Symbol

pybamm.**z_average**(*symbol:* Symbol) → *Symbol*

Convenience function for creating an average in the z-direction.

Parameters
symbol (*pybamm.Symbol*) – The function to be averaged

Returns
the new averaged symbol

Return type
Symbol

pybamm.**yz_average**(*symbol:* Symbol) → *Symbol*

Convenience function for creating an average in the y-z-direction.

Parameters
symbol (*pybamm.Symbol*) – The function to be averaged

Returns
the new averaged symbol

Return type
Symbol

pybamm.**boundary_value**(*symbol*, *side*)

convenience function for creating a *pybamm.BoundaryValue*

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol whose boundary value to take
- **side** (*str*) – Which side to take the boundary value on ("left" or "right")

Returns
the new integrated expression tree

Return type
BoundaryValue

pybamm.**smooth_absolute_value**(*symbol*, *k*)

Smooth approximation to the absolute value function. k is the smoothing parameter, set by *pybamm.settings.abs_smoothing*. The recommended value is k=10.

pybamm.**sign**(*symbol*)

Returns a *Sign* object.

pybamm.**upwind**(*symbol*)

convenience function for creating a *Upwind*

pybamm.**downwind**(*symbol*)

convenience function for creating a *Downwind*

4.1.12 Concatenations

class pybamm.**Concatenation**(**children:* Symbol, *name: str | None = None*, *check_domain=True*, *concat_fun=None*)

A node in the expression tree representing a concatenation of symbols.

Parameters
children (iterable of *pybamm.Symbol*) – The symbols to concatenate

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children: list[*Symbol*] | None = None*, *perform_simplifications: bool = True*)

See `pybamm.Symbol.new_copy()`.

evaluate(*t: float | None = None*, *y: ndarray | None = None*, *y_dot: ndarray | None = None*, *inputs: dict | str | None = None*)

See *pybamm.Symbol.evaluate()*.

get_children_domains(*children: Sequence[Symbol]*)

Combine domains from children, at all levels.

is_constant()

See *pybamm.Symbol.is_constant()*.

to_equation()

Convert the node and its subtree into a SymPy equation.

class pybamm.**NumpyConcatenation**(**children: Symbol*)

A node in the expression tree representing a concatenation of equations, when we *don't* care about domains. The class *pybamm.DomainConcatenation*, which *is* careful about domains and uses broadcasting where appropriate, should be used whenever possible instead.

Upon evaluation, equations are concatenated using numpy concatenation.

Parameters

children (iterable of *pybamm.Symbol*) – The equations to concatenate

Extends: *pybamm.expression_tree.concatenations.Concatenation*

class pybamm.**DomainConcatenation**(*children: Sequence[Symbol]*, *full_mesh: Mesh*, *copy_this: DomainConcatenation | None = None*)

A node in the expression tree representing a concatenation of symbols, being careful about domains.

It is assumed that each child has a domain, and the final concatenated vector will respect the sizes and ordering of domains established in mesh keys

Parameters

- **children** (iterable of *pybamm.Symbol*) – The symbols to concatenate
- **full_mesh** (*pybamm.Mesh*) – The underlying mesh for discretisation, used to obtain the number of mesh points in each domain.
- **copy_this** (*pybamm.DomainConcatenation* (optional)) – if provided, this class is initialised by copying everything except the children from *copy_this*. *mesh* is not used in this case

Extends: *pybamm.expression_tree.concatenations.Concatenation*

to_json()

Method to serialise a DomainConcatenation object into JSON.

class pybamm.**SparseStack**(**children*)

A node in the expression tree representing a concatenation of sparse matrices. As with NumpyConcatenation, we *don't* care about domains. The class *pybamm.DomainConcatenation*, which *is* careful about domains and uses broadcasting where appropriate, should be used whenever possible instead.

Parameters

children (iterable of *Concatenation*) – The equations to concatenate

Extends: *pybamm.expression_tree.concatenations.Concatenation*

pybamm.**numpy_concatenation**(**children*)

Helper function to create numpy concatenations.

pybamm.**domain_concatenation**(*children: list[Symbol]*, *mesh: Mesh*)

Helper function to create domain concatenations.

4.1.13 Broadcasting Operators

class pybamm.**Broadcast**(*child: Symbol*, *domains: dict[str, list[str] | str]*, *name: str | None = None*)

A node in the expression tree representing a broadcasting operator. Broadcasts a child to a specified domain. After discretisation, this will evaluate to an array of the right shape for the specified domain.

For an example of broadcasts in action, see this example notebook

Parameters

- **child** (*Symbol*) – child node
- **domains** (*iterable of str*) – Domain(s) of the symbol after broadcasting
- **name** (*str*) – name of the node

Extends: *pybamm.expression_tree.unary_operators.SpatialOperator*

reduce_one_dimension()

Reduce the broadcast by one dimension.

to_json()

Method to serialise a Symbol object into JSON.

class pybamm.**FullBroadcast**(*child_input: int | float | number | Symbol*, *broadcast_domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *broadcast_domains: dict[str, list[str] | str] | None = None*, *name: str | None = None*)

A class for full broadcasts.

Extends: *pybamm.expression_tree.broadcasts.Broadcast*

check_and_set_domains(*child: Symbol*, *broadcast_domains: dict*)

See `Broadcast.check_and_set_domains()`

reduce_one_dimension()

Reduce the broadcast by one dimension.

class pybamm.**PrimaryBroadcast**(*child: int | float | number | Symbol*, *broadcast_domain: list[str] | str*, *name: str | None = None*)

A node in the expression tree representing a primary broadcasting operator. Broadcasts in a *primary* dimension only. That is, makes explicit copies of the symbol in the domain specified by *broadcast_domain*. This should be used for broadcasting from a "larger" scale to a "smaller" scale, for example broadcasting temperature T(x) from the electrode to the particles, or broadcasting current collector current i(y, z) from the current collector to the electrodes.

Parameters

- **child** (*Symbol*, numeric) – child node
- **broadcast_domain** (*iterable of str*) – Primary domain for broadcast. This will become the domain of the symbol
- **name** (*str*) – name of the node

Extends: *pybamm.expression_tree.broadcasts.Broadcast*

check_and_set_domains(*child: Symbol*, *broadcast_domain: list[str]*)

See `Broadcast.check_and_set_domains()`

reduce_one_dimension()

Reduce the broadcast by one dimension.

class pybamm.**SecondaryBroadcast**(*child:* Symbol, *broadcast_domain: list[str] | str*, *name: str | None = None*)

A node in the expression tree representing a secondary broadcasting operator. Broadcasts in a *secondary* dimension only. That is, makes explicit copies of the symbol in the domain specified by *broadcast_domain*. This should be used for broadcasting from a "smaller" scale to a "larger" scale, for example broadcasting SPM particle concentrations c_s(r) from the particles to the electrodes. Note that this wouldn't be used to broadcast particle concentrations in the DFN, since these already depend on both x and r.

Parameters

- **child** (*Symbol*) – child node
- **broadcast_domain** (*iterable of str*) – Secondary domain for broadcast. This will become the secondary domain of the symbol, shifting the child's *secondary* and *tertiary* (if present) over by one position.
- **name** (*str*) – name of the node

Extends: *pybamm.expression_tree.broadcasts.Broadcast*

check_and_set_domains(*child:* Symbol, *broadcast_domain: list[str]*)

See `Broadcast.check_and_set_domains()`

reduce_one_dimension()

Reduce the broadcast by one dimension.

class pybamm.**FullBroadcastToEdges**(*child: int | float | number | Symbol*, *broadcast_domain: list[str] | str | None = None*, *auxiliary_domains: dict[str, str] | None = None*, *broadcast_domains: dict[str, list[str] | str] | None = None*, *name: str | None = None*)

A full broadcast onto the edges of a domain (edges of primary dimension, nodes of other dimensions)

Extends: *pybamm.expression_tree.broadcasts.FullBroadcast*

reduce_one_dimension()

Reduce the broadcast by one dimension.

class pybamm.**PrimaryBroadcastToEdges**(*child: int | float | number | Symbol*, *broadcast_domain: list[str] | str*, *name: str | None = None*)

A primary broadcast onto the edges of the domain.

Extends: *pybamm.expression_tree.broadcasts.PrimaryBroadcast*

class pybamm.**SecondaryBroadcastToEdges**(*child:* Symbol, *broadcast_domain: list[str] | str*, *name: str | None = None*)

A secondary broadcast onto the edges of a domain.

Extends: *pybamm.expression_tree.broadcasts.SecondaryBroadcast*

pybamm.**ones_like**(**symbols:* Symbol)

Returns an array with the same shape and domains as the sum of the input symbols, with each entry equal to one.

Parameters

symbols (*Symbol*) – Symbols whose shape to copy

pybamm.**zeros_like**(**symbols:* Symbol)

Returns an array with the same shape and domains as the sum of the input symbols, with each entry equal to zero.

Parameters

symbols (*Symbol*) – Symbols whose shape to copy

pybamm.**full_like**(*symbols: tuple[*Symbol*, ...]*, *fill_value: float*) → *Symbol*

Returns an array with the same shape and domains as the sum of the input symbols, with a constant value given by *fill_value*.

Parameters

- **symbols** (*Symbol*) – Symbols whose shape to copy
- **fill_value** (*number*) – Value to assign

4.1.14 Functions

class pybamm.**Function**(*function: Callable*, **children:* Symbol, *name: str | None = None*, *differentiated_function: Callable | None = None*)

A node in the expression tree representing an arbitrary function.

Parameters

- **function** (*method*) – A function can have 0 or many inputs. If no inputs are given, self.evaluate() simply returns func(). Otherwise, self.evaluate(t, y, u) returns func(child0.evaluate(t, y, u), child1.evaluate(t, y, u), etc).
- **children** (*pybamm.Symbol*) – The children nodes to apply the function to
- **differentiated_function** (*method, optional*) – The function which was differentiated to obtain this one. Default is None.

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children: list[*Symbol*] | None = None*, *perform_simplifications: bool = True*)

See `pybamm.Symbol.new_copy()`.

diff(*variable:* Symbol)

See *pybamm.Symbol.diff()*.

evaluate(*t: float | None = None*, *y: ndarray | None = None*, *y_dot: ndarray | None = None*, *inputs: dict | str | None = None*)

See *pybamm.Symbol.evaluate()*.

is_constant()

See *pybamm.Symbol.is_constant()*.

to_equation()

Convert the node and its subtree into a SymPy equation.

to_json()

Method to serialise a Symbol object into JSON.

class pybamm.**SpecificFunction**(*function: Callable*, *child:* Symbol)

Parent class for the specific functions, which implement their own *diff* operators directly.

Parameters

- **function** (*method*) – Function to be applied to child
- **child** (*pybamm.Symbol*) – The child to apply the function to

Extends: *pybamm.expression_tree.functions.Function*

to_json()

Method to serialise a SpecificFunction object into JSON.

class pybamm.**Arcsinh**(*child*)

Arcsinh function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**arcsinh**(*child:* Symbol)

Returns arcsinh function of child.

class pybamm.**Arctan**(*child*)

Arctan function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**arctan**(*child:* Symbol)

Returns hyperbolic tan function of child.

class pybamm.**Cos**(*child*)

Cosine function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**cos**(*child:* Symbol)

Returns cosine function of child.

class pybamm.**Cosh**(*child*)

Hyberbolic cosine function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**cosh**(*child:* Symbol)

Returns hyperbolic cosine function of child.

class pybamm.**Erf**(*child*)

Error function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**erf**(*child:* Symbol)

Returns error function of child.

pybamm.**erfc**(*child:* Symbol)

Returns complementary error function of child.

class pybamm.**Exp**(*child*)

Exponential function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**exp**(*child:* Symbol)

Returns exponential function of child.

class pybamm.**Log**(*child*)

Logarithmic function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**log**(*child*, *base='e'*)

Returns logarithmic function of child (any base, default 'e').

pybamm.**log10**(*child:* Symbol)

Returns logarithmic function of child, with base 10.

class pybamm.**Max**(*child*)

Max function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**max**(*child:* Symbol)

Returns max function of child. Not to be confused with *pybamm.maximum()*, which returns the larger of two objects.

class pybamm.**Min**(*child*)

Min function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**min**(*child:* Symbol)

Returns min function of child. Not to be confused with *pybamm.minimum()*, which returns the smaller of two objects.

pybamm.**sech**(*child:* Symbol)

Returns hyperbolic sec function of child.

class pybamm.**Sin**(*child*)

Sine function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**sin**(*child:* Symbol)

Returns sine function of child.

class pybamm.**Sinh**(*child*)

Hyperbolic sine function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**sinh**(*child:* Symbol)

Returns hyperbolic sine function of child.

class pybamm.**Sqrt**(*child*)

Square root function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**sqrt**(*child:* Symbol)

Returns square root function of child.

class pybamm.**Tanh**(*child*)

Hyperbolic tan function.

Extends: *pybamm.expression_tree.functions.SpecificFunction*

pybamm.**tanh**(*child:* Symbol)

Returns hyperbolic tan function of child.

pybamm.**normal_pdf**(*x:* Symbol, *mu:* Symbol | *float*, *sigma:* Symbol | *float*)

Returns the normal probability density function at x.

Parameters

- **x** (pybamm.Symbol) – The value at which to evaluate the normal distribution
- **mu** (pybamm.Symbol *or* *float*) – The mean of the normal distribution

- **sigma** (*pybamm.Symbol or float*) – The standard deviation of the normal distribution

Returns
The value of the normal distribution at x

Return type
pybamm.Symbol

pybamm.**normal_cdf**(*x:* Symbol, *mu:* Symbol | *float*, *sigma:* Symbol | *float*)

Returns the normal cumulative distribution function at x.

Parameters

- **x** (*pybamm.Symbol*) – The value at which to evaluate the normal distribution
- **mu** (*pybamm.Symbol or float*) – The mean of the normal distribution
- **sigma** (*pybamm.Symbol or float*) – The standard deviation of the normal distribution

Returns
The value of the normal distribution at x

Return type
pybamm.Symbol

4.1.15 Input Parameter

class pybamm.**InputParameter**(*name: str*, *domain: list[str] | str | None = None*, *expected_size: int | None = None*)

A node in the expression tree representing an input parameter.

This node's value can be set at the point of solving, allowing parameter estimation and control

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str, or str*) – list of domains over which the node is valid (empty list indicates the symbol is valid over all domains)
- **expected_size** (*int*) – The size of the input parameter expected, defaults to 1 (scalar input)

Extends: *pybamm.expression_tree.symbol.Symbol*

create_copy(*new_children=None*, *perform_simplifications=True*) → *InputParameter*

See pybamm.Symbol.new_copy().

to_json()

Method to serialise an InputParameter object into JSON.

4.1.16 Interpolant

class pybamm.**Interpolant**(*x: ndarray | Sequence[ndarray]*, *y: ndarray*, *children: Sequence[*Symbol*]* | Time, *name: str | None = None*, *interpolator: str | None = 'linear'*, *extrapolate: bool = True*, *entries_string: str | None = None*, *_num_derivatives: int = 0*)

Interpolate data in 1D, 2D, or 3D. Interpolation in 3D requires the input data to be on a regular grid (as per scipy.interpolate.RegularGridInterpolator).

Parameters

- **x** (iterable of numpy.ndarray) – The data point coordinates. If 1-D, then this is an array(s) of real values. If, 2D or 3D interpolation, then this is to ba a tuple of 1D arrays (one for each dimension) which together define the coordinates of the points.
- **y** (numpy.ndarray) – The values of the function to interpolate at the data points. In 2D and 3D, this should be a matrix of two and three dimensions respectively.
- **children** (iterable of *pybamm.Symbol*) – Node(s) to use when evaluating the interpolant. Each child corresponds to an entry of x
- **name** (*str, optional*) – Name of the interpolant. Default is None, in which case the name "interpolating function" is given.
- **interpolator** (*str, optional*) – Which interpolator to use. Can be "linear", "cubic", or "pchip". Default is "linear". For 3D interpolation, only "linear" an "cubic" are currently supported.
- **extrapolate** (*bool, optional*) – Whether to extrapolate for points that are outside of the parametrisation range, or return NaN (following default behaviour from scipy). Default is True. Generally, it is best to set this to be False for 3D interpolation due to the higher potential for errors in extrapolation.

Extends: *pybamm.expression_tree.functions.Function*

create_copy(*new_children=None*, *perform_simplifications=True*)

See pybamm.Symbol.new_copy().

set_id()

See *pybamm.Symbol.set_id()*.

to_json()

Method to serialise an Interpolant object into JSON.

4.1.17 Operations on expression trees

Classes and functions that operate on the expression tree

EvaluatorPython

class pybamm.**EvaluatorPython**(*symbol:* Symbol)

Converts a pybamm expression tree into pure python code that will calculate the result of calling *evaluate(t, y)* on the given expression tree.

Parameters

symbol (*pybamm.Symbol*) – The symbol to convert to python code

Jacobian

class pybamm.**Jacobian**(*known_jacs: dict*[Symbol, Symbol] | *None* = *None*, *clear_domain: bool* = *True*)

Helper class to calculate the Jacobian of an expression.

Parameters

- **known_jacs** (dict {variable ids -> *pybamm.Symbol*}) – cached jacobians
- **clear_domain** (*bool*) – whether or not the Jacobian clears the domain (default True)

jac(*symbol:* Symbol, *variable:* Symbol) → *Symbol*

This function recurses down the tree, computing the Jacobian using the Jacobians defined in classes derived from pybamm.Symbol. E.g. the Jacobian of a 'pybamm.Multiplication' is computed via the product rule. If the Jacobian of a symbol has already been calculated, the stored value is returned. Note: The Jacobian is the derivative of a symbol with respect to a (slice of) a State Vector.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to calculate the Jacobian of
- **variable** (*pybamm.Symbol*) – The variable with respect to which to differentiate

Returns
Symbol representing the Jacobian

Return type
pybamm.Symbol

Convert to CasADi

class pybamm.**CasadiConverter**(*casadi_symbols=None*)

convert(*symbol:* Symbol, *t: MX*, *y: MX*, *y_dot: MX*, *inputs: dict | None*) → MX

This function recurses down the tree, converting the PyBaMM expression tree to a CasADi expression tree

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to convert
- **t** (casadi.MX) – A casadi symbol representing time
- **y** (casadi.MX) – A casadi symbol representing state vectors
- **y_dot** (casadi.MX) – A casadi symbol representing time derivatives of state vectors
- **inputs** (*dict*) – A dictionary of casadi symbols representing parameters

Returns
The converted symbol

Return type
casadi.MX

Serialise

class pybamm.expression_tree.operations.serialise.**Serialise**

Converts a discretised model to and from a JSON file.

load_model(*filename: str*, *battery_model:* BaseModel | *None* = *None*) → *BaseModel*

Loads a discretised, ready to solve model into PyBaMM.

A new pybamm battery model instance will be created, which can be solved and the results plotted as usual.

Currently only available for pybamm models which have previously been written out using the *save_model()* option.

Warning: This only loads in discretised models. If you wish to make edits to the model or initial conditions, a new model will need to be constructed seperately.

Parameters

- **filename** (*str*) – Path to the JSON file containing the serialised model file

- **battery_model** (*pybamm.BaseModel* (optional)) – PyBaMM model to be created (e.g. pybamm.lithium_ion.SPM), which will override any model names within the file. If None, the function will look for the saved object path, present if the original model came from PyBaMM.

Returns
A PyBaMM model object, of type specified either in the JSON or in *battery_model*.

Return type
pybamm.BaseModel

save_model(*model:* BaseModel, *mesh:* Mesh | *None* = *None*, *variables:* FuzzyDict | *None* = *None*, *filename:* *str* | *None* = *None*)

Saves a discretised model to a JSON file.

As the model is discretised and ready to solve, only the right hand side, algebraic and initial condition variables are saved.

Parameters

- **model** (*pybamm.BaseModel*) – The discretised model to be saved
- **mesh** (*pybamm.Mesh* (optional)) – The mesh the model has been discretised over. Not neccesary to solve the model when read in, but required to use pybamm's plotting tools.
- **variables** (*pybamm.FuzzyDict* (optional)) – The discretised model varaibles. Not necessary to solve a model, but required to use pybamm's plotting tools.
- **filename** (*str (optional)*) – The desired name of the JSON file. If no name is provided, one will be created based on the model name, and the current datetime.

Symbol Unpacker

class pybamm.**SymbolUnpacker**(*classes_to_find: Sequence[*pybamm.Symbol*]* | pybamm.Symbol, *unpacked_symbols: dict* | *None* = *None*)

Helper class to unpack a (set of) symbol(s) to find all instances of a class. Uses caching to speed up the process.

Parameters

- **classes_to_find** (*list of pybamm classes*) – Classes to identify in the equations
- **unpacked_symbols** (*set, optional*) – cached unpacked equations

unpack_list_of_symbols(*list_of_symbols: Sequence[*pybamm.Symbol*]*) → set[*pybamm.Symbol*]

Unpack a list of symbols. See `SymbolUnpacker.unpack()`

Parameters
list_of_symbols (list of *pybamm.Symbol*) – List of symbols to unpack

Returns
Set of unpacked symbols with class in *self.classes_to_find*

Return type
list of *pybamm.Symbol*

unpack_symbol(*symbol: Sequence[*pybamm.Symbol*]* | pybamm.Symbol) → list[*pybamm.Symbol*]

This function recurses down the tree, unpacking the symbols and saving the ones that have a class in *self.classes_to_find*.

Parameters
symbol (list of *pybamm.Symbol*) – The symbols to unpack

Returns
List of unpacked symbols with class in *self.classes_to_find*

Return type
list of *pybamm.Symbol*

4.2 Models

Below is an overview of all the battery models included in PyBaMM. Each of the pre-built models contains a reference to the paper in which it is derived.

The models can be customised using the `options` dictionary defined in the *pybamm.BaseBatteryModel* (which also provides information on which options and models are compatible) Visit our examples page to see how these models can be solved, and compared, using PyBaMM.

4.2.1 Base Models

Base Model

class pybamm.**BaseModel**(*name='Unnamed model'*)

Base model class for other models to extend.

name

A string representing the name of the model.

Type
str

submodels

A dictionary of submodels that the model is composed of.

Type
dict

use_jacobian

Whether to use the Jacobian when solving the model (default is True).

Type
bool

convert_to_format

Specifies the format to convert the expression trees representing the RHS, algebraic equations, Jacobian, and events. Options are:

- None: retain PyBaMM expression tree structure.
- "python": convert to Python code for evaluating *evaluate(t, y)* on expressions.
- "casadi": convert to CasADi expression tree for Jacobian calculation.
- "jax": convert to JAX expression tree.

Default is "casadi".

Type
str

is_discretised

Indicates whether the model has been discretised (default is False).

Type
bool

`y_slices`

Slices of the concatenated state vector after discretisation, used to track different submodels in the full concatenated solution vector.

Type
None or list

`property algebraic`

Returns a dictionary mapping expressions (variables) to expressions that represent the algebraic equations of the model.

`property boundary_conditions`

Returns a dictionary mapping expressions (variables) to expressions representing the boundary conditions of the model.

`check_algebraic_equations`(*post_discretisation*)

Check that the algebraic equations are well-posed. After discretisation, there must be at least one StateVector in each algebraic equation.

`check_discretised_or_discretise_inplace_if_0D`()

Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user

`check_ics_bcs`()

Check that the initial and boundary conditions are well-posed.

`check_no_repeated_keys`()

Check that no equation keys are repeated.

`check_well_determined`(*post_discretisation*)

Check that the model is not under- or over-determined.

`check_well_posedness`(*post_discretisation=False*)

Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic. Overdetermined if more equations than variables, underdetermined if more variables than equations. - There is an initial condition in self.initial_conditions for each variable/equation pair in self.rhs - There are appropriate boundary conditions in self.boundary_conditions for each variable/equation pair in self.rhs and self.algebraic

Parameters
post_discretisation (*boolean*) – A flag indicating tests to be skipped after discretisation

`property concatenated_algebraic`

Returns the concatenated algebraic equations for the model after discretisation.

`property concatenated_initial_conditions`

Returns the initial conditions for all variables after discretization, providing the initial values for the state variables.

`property concatenated_rhs`

Returns the concatenated right-hand side (RHS) expressions for the model after discretisation.

`property coupled_variables`

Returns a dictionary mapping strings to expressions representing variables needed by the model but whose equations were set by other models.

property default_geometry

Returns a dictionary of the default geometry for the model, which is empty by default.

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

property default_solver

Returns the default solver for the model, based on whether it is an ODE/DAE or algebraic model.

property default_spatial_methods

Returns a dictionary of the default spatial methods for the model, which is empty by default.

property default_submesh_types

Returns a dictionary of the default submesh types for the model, which is empty by default.

property default_var_pts

Returns a dictionary of the default variable points for the model, which is empty by default.

classmethod deserialise(*properties: dict*)

Create a model instance from a serialised object.

property events

Returns a dictionary mapping expressions (variables) to expressions that represent the initial conditions for the state variables.

export_casadi_objects(*variable_names*, *input_parameter_order=None*)

Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.

Parameters

- **variable_names** (*list*) – Variables to be exported alongside the model structure
- **input_parameter_order** (*list, optional*) – Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

Returns

casadi_dict – Dictionary of {str: casadi object} pairs representing the model in casadi format

Return type

dict

generate(*filename*, *variable_names*, *input_parameter_order=None*, *cg_options=None*)

Generate the model in C, using CasADi.

Parameters

- **filename** (*str*) – Name of the file to which to save the code
- **variable_names** (*list*) – Variables to be exported alongside the model structure
- **input_parameter_order** (*list, optional*) – Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)
- **cg_options** (*dict*) – Options to pass to the code generator. See https://web.casadi.org/docs/#generating-c-code

property geometry

Returns the geometry of the model.

get_parameter_info(*by_submodel=False*)

Extracts the parameter information and returns it as a dictionary. To get a list of all parameter-like objects without extra information, use `model.parameters`.

Parameters

by_submodel (*bool, optional*) – Whether to return the parameter info sub-model wise or not (default False)

info(*symbol_name*)

Provides helpful summary information for a symbol.

Parameters

symbol_name (*str*)

property initial_conditions

Returns a dictionary mapping expressions (variables) to expressions that represent the initial conditions for the state variables.

property input_parameters

Returns a list of all input parameter symbols used in the model.

property jacobian

Returns the Jacobian matrix for the model, computed automatically if *use_jacobian* is True.

property jacobian_algebraic

Returns the Jacobian matrix for the algebraic part of the model, computed automatically during solver setup if *use_jacobian* is True.

property jacobian_rhs

Returns the Jacobian matrix for the right-hand side (RHS) part of the model, computed if *use_jacobian* is True.

latexify(*filename=None*, *newline=True*, *output_variables=None*)

Converts all model equations in latex.

Parameters

- **filename** (*str (optional)*) – Accepted file formats - any image format, pdf and tex Default is None, When None returns all model equations in latex If not None, returns all model equations in given file format.
- **newline** (*bool (optional)*) – Default is True, If True, returns every equation in a new line. If False, returns the list of all the equations.
- **model** (*Load*)
- **pybamm.lithium_ion.SPM()** (*>>> model =*)
- **png** (*This will returns all model equations in*)
- **doctest** (*>>> model.latexify(newline=False) #*)
- **latex** (*This will return all the model equations in*)
- **doctest**
- **equations** (*This will return first five model*)
- **doctest**

- **equations**
- **model.latexify(newline=False)[1** (>>>)

property mass_matrix

Returns the mass matrix for the system of differential equations after discretisation.

property mass_matrix_inv

Returns the inverse of the mass matrix for the differential equations after discretisation.

new_copy()

Creates a copy of the model, explicitly copying all the mutable attributes to avoid issues with shared objects.

property options

Returns the model options dictionary that allows customization of the model's behavior.

property param

Returns a dictionary to store parameter values for the model.

property parameters

Returns a list of all parameter symbols used in the model.

print_parameter_info(*by_submodel=False*)

Print parameter information in a formatted table from a dictionary of parameters

Parameters

by_submodel (*bool, optional*) – Whether to print the parameter info sub-model wise or not (default False)

process_parameters_and_discretise(*symbol*, *parameter_values*, *disc*)

Process parameters and discretise a symbol using supplied parameter values and discretisation. Note: care should be taken if using spatial operators on dimensional symbols. Operators in pybamm are written in non-dimensional form, so may need to be scaled by the appropriate length scale. It is recommended to use this method on non-dimensional symbols.

Parameters

- **symbol** (*pybamm.Symbol*) – Symbol to be processed
- **parameter_values** (*pybamm.ParameterValues*) – The parameter values to use during processing
- **disc** (*pybamm.Discretisation*) – The discrisation to use

Returns

Processed symbol

Return type

pybamm.Symbol

property rhs

Returns a dictionary mapping expressions (variables) to expressions that represent the right-hand side (RHS) of the model's differential equations.

save_model(*filename=None*, *mesh=None*, *variables=None*)

Write out a discretised model to a JSON file

Parameters

- **filename** (*str, optional*)
- **provided** (*The desired name of the JSON file. If no name is*)

- **created** (*one will be*)
- **name** (*based on the model*)
- **datetime.** (*and the current*)

set_initial_conditions_from(*solution*, *inplace=True*, *return_type='model'*)

Update initial conditions with the final states from a Solution object or from a dictionary. This assumes that, for each variable in self.initial_conditions, there is a corresponding variable in the solution with the same name and size.

Parameters

- **solution** (*pybamm.Solution*, or dict) – The solution to use to initialize the model
- **inplace** (*bool, optional*) – Whether to modify the model inplace or create a new model (default True)
- **return_type** (*str, optional*) – Whether to return the model (default) or initial conditions ("ics")

update(**submodels*)

Update model to add new physics from submodels

Parameters

submodel (iterable of *pybamm.BaseModel*) – The submodels from which to create new model

property variables

Returns a dictionary mapping strings to expressions representing the model's useful variables.

property variables_and_events

Returns a dictionary containing both models variables and events.

Base Battery Model

class pybamm.**BaseBatteryModel**(*options=None*, *name='Unnamed battery model'*)

Base model class with some default settings and required variables

Parameters

- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model. If this is a dict (and not a subtype of dict), it will be processed by *pybamm.BatteryModelOptions* to ensure that the options are valid. If this is a subtype of dict, it is assumed that the options have already been processed and are valid. This allows for the use of custom options classes. The default options are given by *pybamm.BatteryModelOptions*.
- **name** (*str, optional*) – The name of the model. The default is "Unnamed battery model".

Extends: *pybamm.models.base_model.BaseModel*

property default_geometry

Returns a dictionary of the default geometry for the model, which is empty by default.

property default_spatial_methods

Returns a dictionary of the default spatial methods for the model, which is empty by default.

property default_submesh_types

Returns a dictionary of the default submesh types for the model, which is empty by default.

property default_var_pts

Returns a dictionary of the default variable points for the model, which is empty by default.

classmethod deserialise(*properties: dict*)

Create a model instance from a serialised object.

property options

Returns the model options dictionary that allows customization of the model's behavior.

save_model(*filename=None*, *mesh=None*, *variables=None*)

Write out a discretised model to a JSON file

Parameters

- **filename** (*str, optional*)
- **provided** (*The desired name of the JSON file. If no name is*)
- **created** (*one will be*)
- **name** (*based on the model*)
- **datetime.** (*and the current*)

set_degradation_variables()

Set variables that quantify degradation. This function is overriden by the base battery models

set_external_circuit_submodel()

Define how the external circuit defines the boundary conditions for the model, e.g. (not necessarily constant-) current, voltage, etc

set_soc_variables()

Set variables relating to the state of charge. This function is overriden by the base battery models

class pybamm.BatteryModelOptions(*extra_options*)

options

A dictionary of options to be passed to the model. The options that can be set are listed below. Note that not all of the options are compatible with each other and with all of the models implemented in PyBaMM. Each option is optional and takes a default value if not provided. In general, the option provided must be a string, but there are some cases where a 2-tuple of strings can be provided instead to indicate a different option for the negative and positive electrodes.

- **"calculate discharge energy": str**
  Whether to calculate the discharge energy, throughput energy and throughput capacity in addition to discharge capacity. Must be one of "true" or "false". "false" is the default, since calculating discharge energy can be computationally expensive for simple models like SPM.
- **"cell geometry"**
  [str] Sets the geometry of the cell. Can be "arbitrary" (default) or "pouch". The arbitrary geometry option solves a 1D electrochemical model with prescribed cell volume and cross-sectional area, and (if thermal effects are included) solves a lumped thermal model with prescribed surface area for cooling.
- **"calculate heat source for isothermal models"**
  [str] Whether to calculate the heat source terms during isothermal operation. Can be "true" or "false". If "false", the heat source terms are set to zero. Default is "false" since this option may require additional parameters not needed by the electrochemical model.

- **"convection"**
  [str] Whether to include the effects of convection in the model. Can be "none" (default), "uniform transverse" or "full transverse". Must be "none" for lithium-ion models.
- **"current collector"**
  [str] Sets the current collector model to use. Can be "uniform" (default), "potential pair" or "potential pair quite conductive".
- **"diffusivity"**
  [str] Sets the model for the diffusivity. Can be "single" (default) or "current sigmoid". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"dimensionality"**
  [int] Sets the dimension of the current collector problem. Can be 0 (default), 1 or 2.
- **"electrolyte conductivity"**
  [str] Can be "default" (default), "full", "leading order", "composite" or "integrated".
- **"exchange-current density"**
  [str] Sets the model for the exchange-current density. Can be "single" (default) or "current sigmoid". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"hydrolysis"**
  [str] Whether to include hydrolysis in the model. Only implemented for lead-acid models. Can be "false" (default) or "true". If "true", then "surface form" cannot be 'false'.
- **"intercalation kinetics"**
  [str] Model for intercalation kinetics. Can be "symmetric Butler-Volmer" (default), "asymmetric Butler-Volmer", "linear", "Marcus", "Marcus-Hush-Chidsey" (which uses the asymptotic form from Zeng 2014), or "MSMR" (which uses the form from Baker 2018). A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"interface utilisation": str**
  Can be "full" (default), "constant", or "current-driven".
- **"lithium plating"**
  [str] Sets the model for lithium plating. Can be "none" (default), "reversible", "partially reversible", or "irreversible".
- **"lithium plating porosity change"**
  [str] Whether to include porosity change due to lithium plating, can be "false" (default) or "true".
- **"loss of active material"**
  [str] Sets the model for loss of active material. Can be "none" (default), "stress-driven", "reaction-driven", "current-driven", or "stress and reaction-driven". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"number of MSMR reactions"**
  [str] Sets the number of reactions to use in the MSMR model in each electrode. A 2-tuple can be provided to give a different number of reactions in the negative and positive electrodes. Default is "none". Can be any 2-tuple of strings of integers. For example, set to ("6", "4") for a negative electrode with 6 reactions and a positive electrode with 4 reactions.
- **"open-circuit potential"**
  [str] Sets the model for the open circuit potential. Can be "single" (default), "current sigmoid", "Wycisk", or "MSMR". If "MSMR" then the "particle" option must also be "MSMR". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"operating mode"**
  [str] Sets the operating mode for the model. This determines how the current is set. Can be:

  - "current" (default) : the current is explicity supplied
  - "voltage"/"power"/"resistance" : solve an algebraic equation for current such that voltage/power/resistance is correct
  - "differential power"/"differential resistance" : solve a differential equation for the power or resistance
  - "explicit power"/"explicit resistance" : current is defined in terms of the voltage such that power/resistance is correct
  - "CCCV": a special implementation of the common constant-current constant-voltage charging protocol, via an ODE for the current
  - callable : if a callable is given as this option, the function defines the residual of an algebraic equation. The applied current will be solved for such that the algebraic constraint is satisfied.

- **"particle"**
  [str] Sets the submodel to use to describe behaviour within the particle. Can be "Fickian diffusion" (default), "uniform profile", "quadratic profile", "quartic profile", or "MSMR". If "MSMR" then the "open-circuit potential" option must also be "MSMR". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"particle mechanics"**
  [str] Sets the model to account for mechanical effects such as particle swelling and cracking. Can be "none" (default), "swelling only", or "swelling and cracking". A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"particle phases": str**
  Number of phases present in the electrode. A 2-tuple can be provided for different behaviour in negative and positive electrodes. For example, set to ("2", "1") for a negative electrode with 2 phases, e.g. graphite and silicon.
- **"particle shape"**
  [str] Sets the model shape of the electrode particles. This is used to calculate the surface area to volume ratio. Can be "spherical" (default), or "no particles".
- **"particle size"**
  [str] Sets the model to include a single active particle size or a distribution of sizes at any macroscale location. Can be "single" (default) or "distribution". Option applies to both electrodes.
- **"SEI"**
  [str] Set the SEI submodel to be used. Options are:
  - "none": *pybamm.sei.NoSEI* (no SEI growth)
  - "constant": `pybamm.sei.Constant` (constant SEI thickness)
  - "reaction limited", "reaction limited (asymmetric)", "solvent-diffusion limited", "electron-migration limited", "interstitial-diffusion limited", "ec reaction limited" , "VonKolzenberg2020", "tunnelling limited", or "ec reaction limited (asymmetric)": *pybamm.sei.SEIGrowth*
- **"SEI film resistance"**
  [str] Set the submodel for additional term in the overpotential due to SEI. The default value is "none" if the "SEI" option is "none", and "distributed" otherwise. This is because the "distributed" model is more complex than the model with no additional resistance, which adds unnecessary complexity if there is no SEI in the first place

  - **"none": no additional resistance**

$$\eta_r = \frac{F}{RT} * (\phi_s - \phi_e - U)$$

  - **"distributed": properly included additional resistance term**

$$\eta_r = \frac{F}{RT} * (\phi_s - \phi_e - U - R_{sei} * L_{sei} * j)$$

  - **"average": constant additional resistance term (approximation to the true model). This model can give similar results to the "distributed" case without needing to make j an algebraic state**

$$\eta_r = \frac{F}{RT} * (\phi_s - \phi_e - U - R_{sei} * L_{sei} * \frac{I}{aL})$$

- **"SEI on cracks"**
  [str] Whether to include SEI growth on particle cracks, can be "false" (default) or "true".
- **"SEI porosity change"**
  [str] Whether to include porosity change due to SEI formation, can be "false" (default) or "true".
- **"stress-induced diffusion"**
  [str] Whether to include stress-induced diffusion, can be "false" or "true". The default is "false" if "particle mechanics" is "none" and "true" otherwise. A 2-tuple can be provided for different behaviour in negative and positive electrodes.
- **"surface form"**
  [str] Whether to use the surface formulation of the problem. Can be "false" (default), "differential" or "algebraic".
- **"surface temperature"**
  [str] Sets the surface temperature model to use. Can be "ambient" (default), which sets the surface temperature equal to the ambient temperature, or "lumped", which adds an ODE for the surface temperature (e.g. to model internal heating of a thermal chamber).
- **"thermal"**
  [str] Sets the thermal model to use. Can be "isothermal" (default), "lumped", "x-lumped", or "x-full". The 'cell geometry' option must be set to 'pouch' for 'x-lumped' or 'x-full' to be valid. Using the 'x-lumped' option with 'dimensionality' set to 0 is equivalent to using the 'lumped' option.
- **"total interfacial current density as a state"**
  [str] Whether to make a state for the total interfacial current density and solve an algebraic equation for it. Default is "false", unless "SEI film resistance" is distributed in which case it is automatically set to "true".
- **"voltage as a state"**
  [str] Whether to make a state for the voltage and solve an algebraic equation for it. Default is "false".
- **"working electrode"**
  [str] Can be "both" (default) for a standard battery or "positive" for a half-cell where the negative electrode is replaced with a lithium metal counter electrode.
- **"x-average side reactions": str**
  Whether to average the side reactions (SEI growth, lithium plating and the respective porosity change) over the x-axis in Single Particle Models, can be "false" or "true". Default is "false" for SPMe and "true" for SPM.

Type
dict

Extends: *pybamm.util.FuzzyDict*

property negative

Returns the options for the negative electrode

property positive

Returns the options for the positive electrode

print_detailed_options()

Print the docstring for Options

print_options()

Print the possible options with the ones currently selected

Event

class pybamm.**Event**(*name*, *expression*, *event_type=EventType.TERMINATION*)

Defines an event for use within a pybamm model

name

A string giving the name of the event.

Type
str

expression

An expression that defines when the event occurs.

Type
pybamm.Symbol

event_type

An enum defining the type of event. By default it is set to TERMINATION.

Type
pybamm.EventType (optional)

evaluate(*t: float | None = None*, *y: ndarray | None = None*, *y_dot: ndarray | None = None*, *inputs: dict | None = None*)

Acts as a drop-in replacement for *pybamm.Symbol.evaluate()*

to_json()

Method to serialise an Event object into JSON.

The expression is written out seperately, See pybamm.Serialise._SymbolEncoder.default()

class pybamm.**EventType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Defines the type of event, see *pybamm.Event*

TERMINATION indicates an event that will terminate the solver, the expression should return 0 when the event is triggered

DISCONTINUITY indicates an expected discontinuity in the solution, the expression should return the time that the discontinuity occurs. The solver will integrate up to the discontinuity and then restart just after the discontinuity.

INTERPOLANT_EXTRAPOLATION indicates that a pybamm.Interpolant object has been evaluated outside of the range.

SWITCH indicates an event switch that is used in CasADI "fast with events" model.

Extends: enum.Enum

4.2.2 Lithium-ion Models

Base Lithium-ion Model

class pybamm.lithium_ion.**BaseModel**(*options=None*, *name='Unnamed lithium-ion model'*, *build=False*)

Overwrites default parameters from Base Model with default parameters for lithium-ion models

Parameters

- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model. If this is a dict (and not a subtype of dict), it will be processed by *pybamm.BatteryModelOptions* to ensure that the options are valid. If this is a subtype of dict, it is assumed that the options have already been processed and are valid. This allows for the use of custom options classes. The default options are given by *pybamm.BatteryModelOptions*.
- **name** (*str, optional*) – The name of the model. The default is "Unnamed battery model".
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

Extends: *pybamm.models.full_battery_models.base_battery_model.BaseBatteryModel*

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

insert_reference_electrode(*position=None*)

Insert a reference electrode to measure the electrolyte potential at a given position in space. Adds model variables for the electrolyte potential at the reference electrode and for the potential difference between the electrode potentials measured at the electrode/current collector interface and the reference electrode. Only implemented for 1D models (i.e. where the 'dimensionality' option is 0).

Parameters

position (*pybamm.Symbol*, optional) – The position in space at which to measure the electrolyte potential. If None, defaults to the mid-point of the separator.

set_default_summary_variables()

Sets the default summary variables.

set_degradation_variables()

Sets variables that quantify degradation (LAM, LLI, etc)

Single Particle Model (SPM)

class pybamm.lithium_ion.**SPM**(*options=None*, *name='Single Particle Model'*, *build=True*)

Single Particle Model (SPM) of a lithium-ion battery, from Marquis *et al.*[1]. See *pybamm.lithium_ion.BaseModel* for more details.

[1] Scott G. Marquis, Valentin Sulzer, Robert Timms, Colin P. Please, and S. Jon Chapman. An asymptotic derivation of a single particle model with electrolyte. *Journal of The Electrochemical Society*, 166(15):A3693–A3706, 2019. doi:10.1149/2.0341915jes.

Examples

```
>>> model = pybamm.lithium_ion.SPM()
>>> model.name
'Single Particle Model'
```

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

class pybamm.lithium_ion.**BasicSPM**(*name='Single Particle Model'*)

Single Particle Model (SPM) model of a lithium-ion battery, from Marquis *et al.*[Page 77, 1].

This class differs from the *pybamm.lithium_ion.SPM* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in combining different physical effects, and in general the main SPM class should be used instead.

Parameters

name (*str, optional*) – The name of the model.

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

References

Single Particle Model with Electrolyte (SPMe)

class pybamm.lithium_ion.**SPMe**(*options=None*, *name='Single Particle Model with electrolyte'*, *build=True*)

Single Particle Model with Electrolyte (SPMe) of a lithium-ion battery, from Marquis *et al.*[1]. Inherits most submodels from SPM, only modifies potentials and electrolyte. See *pybamm.lithium_ion.BaseModel* for more details.

Examples

```
>>> model = pybamm.lithium_ion.SPMe()
>>> model.name
'Single Particle Model with electrolyte'
```

Extends: *pybamm.models.full_battery_models.lithium_ion.spm.SPM*

References

Many Particle Model (MPM)

class pybamm.lithium_ion.**MPM**(*options=None*, *name='Many-Particle Model'*, *build=True*)

Many-Particle Model (MPM) of a lithium-ion battery with particle-size distributions for each electrode, from Kirk *et al.*[1]. See *pybamm.lithium_ion.BaseModel* for more details.

Examples

```
>>> model = pybamm.lithium_ion.MPM()
>>> model.name
'Many-Particle Model'
```

[1] Scott G. Marquis, Valentin Sulzer, Robert Timms, Colin P. Please, and S. Jon Chapman. An asymptotic derivation of a single particle model with electrolyte. *Journal of The Electrochemical Society*, 166(15):A3693–A3706, 2019. doi:10.1149/2.0341915jes.

[1] Toby L. Kirk, Jack Evans, Colin P. Please, and S. Jonathan Chapman. Modelling electrode heterogeneity in lithium-ion batteries: unimodal and bimodal particle-size distributions. arXiv:2006.12208, 2020. URL: https://arxiv.org/abs/2006.12208, arXiv:2006.12208.

Extends: *pybamm.models.full_battery_models.lithium_ion.spm.SPM*

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

References

Doyle-Fuller-Newman (DFN)

class pybamm.lithium_ion.**DFN**(*options=None*, *name='Doyle-Fuller-Newman model'*, *build=True*)

Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from Marquis *et al.*[1]. See *pybamm.lithium_ion.BaseModel* for more details.

Examples

```
>>> model = pybamm.lithium_ion.DFN()
>>> model.name
'Doyle-Fuller-Newman model'
```

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

class pybamm.lithium_ion.**BasicDFN**(*name='Doyle-Fuller-Newman model'*)

Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from Marquis *et al.*[1].

This class differs from the *pybamm.lithium_ion.DFN* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in comparing different physical effects, and in general the main DFN class should be used instead.

Parameters

name (*str, optional*) – The name of the model.

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

class pybamm.lithium_ion.**BasicDFNComposite**(*name='Composite graphite/silicon Doyle-Fuller-Newman model'*)

Doyle-Fuller-Newman (DFN) model of a lithium-ion battery with composite particles of graphite and silicon, from Ai *et al.*[2].

This class differs from the *pybamm.lithium_ion.DFN* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in comparing different physical effects, and in general the main DFN class should be used instead.

Parameters

name (*str, optional*) – The name of the model.

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

[1] Scott G. Marquis, Valentin Sulzer, Robert Timms, Colin P. Please, and S. Jon Chapman. An asymptotic derivation of a single particle model with electrolyte. *Journal of The Electrochemical Society*, 166(15):A3693–A3706, 2019. doi:10.1149/2.0341915jes.

[2] Weilong Ai, Niall Kirkaldy, Yang Jiang, Gregory Offer, Huizhi Wang, and Billy Wu. A composite electrode model for lithium-ion batteries with silicon/graphite negative electrodes. *Journal of Power Sources*, 527:231142, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0378775322001604, doi:https://doi.org/10.1016/j.jpowsour.2022.231142.

class pybamm.lithium_ion.**BasicDFNHalfCell**(*options=None*, *name='Doyle-Fuller-Newman half cell model'*)

Doyle-Fuller-Newman (DFN) model of a lithium-ion battery with lithium counter electrode, adapted from Doyle *et al.*[3].

This class differs from the *pybamm.lithium_ion.BasicDFN* model class in that it is for a cell with a lithium counter electrode (half cell). This is a feature under development (for example, it cannot be used with the Experiment class for the moment) and in the future it will be incorporated as a standard model with the full functionality.

The electrode labeled "positive electrode" is the working electrode, and the electrode labeled "negative electrode" is the counter electrode. This facilitates compatibility with the full-cell models.

Parameters

- **options** (*dict*) – A dictionary of options to be passed to the model. For the half cell it should include which is the working electrode.
- **name** (*str, optional*) – The name of the model.

Extends: *pybamm.models.full_battery_models.lithium_ion.base_lithium_ion_model.BaseModel*

References

Newman-Tobias

class pybamm.lithium_ion.**NewmanTobias**(*options=None*, *name='Newman-Tobias model'*, *build=True*)

Newman-Tobias model of a lithium-ion battery based on the formulation in Newman and Tobias[1]. This model assumes a uniform concentration profile in the electrolyte. Unlike the model posed in Newman and Tobias[1], this model accounts for nonlinear Butler-Volmer kinetics. It also tracks the average concentration in the solid phase in each electrode, which is equivalent to including an equation for the local state of charge as in Chu *et al.*[2]. The user can pass the "particle" option to include mass transport in the particles.

See *pybamm.lithium_ion.BaseModel* for more details.

Extends: *pybamm.models.full_battery_models.lithium_ion.dfn.DFN*

References

Multi-Species Multi-Reaction (MSMR) Model

class pybamm.lithium_ion.**MSMR**(*options=None*, *name='MSMR'*, *build=True*)

property **default_parameter_values**

Returns the default parameter values for the model (an empty set of parameters by default).

Yang et al 2017

class pybamm.lithium_ion.**Yang2017**(*options=None*, *name='Yang2017'*, *build=True*)

[3] Marc Doyle, Thomas F. Fuller, and John 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.

[1] John S Newman and Charles W Tobias. Theoretical analysis of current distribution in porous electrodes. *Journal of The Electrochemical Society*, 109(12):1183, 1962.

[2] Howie N Chu, Sun Ung Kim, Saeed Khaleghi Rahimian, Jason B Siegel, and Charles W Monroe. Parameterization of prismatic lithium–iron–phosphate cells through a streamlined thermal/electrochemical model. *Journal of Power Sources*, 453:227787, 2020.

Electrode SOH models

class pybamm.lithium_ion.**ElectrodeSOHSolver**(*parameter_values*, *param=None*, *known_value='cyclable lithium capacity'*, *options=None*)

Class used to check if the electrode SOH model is feasible, and solve it if it is.

Parameters

- **parameter_values** (pybamm.ParameterValues.Parameters) – The parameters of the simulation
- **param** (*pybamm.LithiumIonParameters*, optional) – Specific instance of the symbolic lithium-ion parameter class. If not provided, the default set of symbolic lithium-ion parameters will be used.
- **known_value** (*str, optional*) – The known value needed to complete the electrode SOH model. Can be "cyclable lithium capacity" (default) or "cell capacity".
- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model, see *pybamm.BatteryModelOptions*.

get_initial_ocps(*initial_value*, *tol=1e-06*, *inputs=None*)

Calculate initial open-circuit potentials to start off the simulation at a particular state of charge, given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **initial_value** (*float*) – Target initial value. If integer, interpreted as SOC, must be between 0 and 1. If string e.g. "4 V", interpreted as voltage, must be between V_min and V_max.
- **tol** (*float, optional*) – Tolerance for the solver used in calculating initial stoichiometries.
- **inputs** (*dict, optional*) – A dictionary of input parameters passed to the model.

Returns

The initial open-circuit potentials at the desired initial state of charge

Return type

Un, Up

get_initial_stoichiometries(*initial_value*, *tol=1e-06*, *inputs=None*)

Calculate initial stoichiometries to start off the simulation at a particular state of charge, given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **initial_value** (*float*) – Target initial value. If integer, interpreted as SOC, must be between 0 and 1. If string e.g. "4 V", interpreted as voltage, must be between V_min and V_max.
- **tol** (*float, optional*) – The tolerance for the solver used to compute the initial stoichiometries. A lower value results in higher precision but may increase computation time. Default is 1e-6.
- **inputs** (*dict, optional*) – A dictionary of input parameters passed to the model.

Returns

The initial stoichiometries that give the desired initial state of charge

Return type

x, y

get_min_max_ocps()

Calculate min/max open-circuit potentials given voltage limits, open-circuit potentials, etc defined by parameter_values

Returns
The min/max ocps

Return type
Un_0, Un_100, Up_100, Up_0

get_min_max_stoichiometries(*inputs=None*)

Calculate min/max stoichiometries given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters
inputs (*dict, optional*) – A dictionary of input parameters passed to the model.

Returns
The min/max stoichiometries

Return type
x_0, x_100, y_100, y_0

pybamm.lithium_ion.**get_initial_stoichiometries**(*initial_value*, *parameter_values*, *param=None*, *known_value='cyclable lithium capacity'*, *options=None*, *tol=1e-06*, *inputs=None*)

Calculate initial stoichiometries to start off the simulation at a particular state of charge, given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **initial_value** (*float*) – Target initial value. If integer, interpreted as SOC, must be between 0 and 1. If string e.g. "4 V", interpreted as voltage, must be between V_min and V_max.
- **parameter_values** (*pybamm.ParameterValues*) – The parameter values class that will be used for the simulation. Required for calculating appropriate initial stoichiometries.
- **param** (*pybamm.LithiumIonParameters*, optional) – The symbolic parameter set to use for the simulation. If not provided, the default parameter set will be used.
- **known_value** (*str, optional*) – The known value needed to complete the electrode SOH model. Can be "cyclable lithium capacity" (default) or "cell capacity".
- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model, see *pybamm.BatteryModelOptions*.
- **tol** (*float, optional*) – The tolerance for the solver used to compute the initial stoichiometries. Default is 1e-6.
- **inputs** (*dict, optional*) – A dictionary of input parameters passed to the model.

Returns
The initial stoichiometries that give the desired initial state of charge

Return type
x, y

pybamm.lithium_ion.**get_min_max_stoichiometries**(*parameter_values*, *param=None*, *known_value='cyclable lithium capacity'*, *options=None*)

Calculate min/max stoichiometries given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **parameter_values** (*pybamm.ParameterValues*) – The parameter values class that will be used for the simulation. Required for calculating appropriate initial stoichiometries.
- **param** (*pybamm.LithiumIonParameters*, optional) – The symbolic parameter set to use for the simulation. If not provided, the default parameter set will be used.
- **known_value** (*str, optional*) – The known value needed to complete the electrode SOH model. Can be "cyclable lithium capacity" (default) or "cell capacity".
- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model, see *pybamm.BatteryModelOptions*.

Returns
The min/max stoichiometries

Return type
x_0, x_100, y_100, y_0

pybamm.lithium_ion.**get_initial_ocps**(*initial_value*, *parameter_values*, *param=None*, *known_value='cyclable lithium capacity'*, *options=None*, *tol=1e-06*, *inputs=None*)

Calculate initial open-circuit potentials to start off the simulation at a particular state of charge, given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **initial_value** (*float*) – Target initial value. If integer, interpreted as SOC, must be between 0 and 1. If string e.g. "4 V", interpreted as voltage, must be between V_min and V_max.
- **parameter_values** (*pybamm.ParameterValues*) – The parameter values class that will be used for the simulation. Required for calculating appropriate initial stoichiometries.
- **param** (*pybamm.LithiumIonParameters*, optional) – The symbolic parameter set to use for the simulation. If not provided, the default parameter set will be used.
- **known_value** (*str, optional*) – The known value needed to complete the electrode SOH model. Can be "cyclable lithium capacity" (default) or "cell capacity".
- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model, see *pybamm.BatteryModelOptions*.
- **tol** (*float, optional*) – Tolerance for the solver used in calculating initial open-circuit potentials.
- **inputs** (*dict, optional*) – A dictionary of input parameters passed to the model.

Returns
The initial electrode OCPs that give the desired initial state of charge

Return type
Un, Up

pybamm.lithium_ion.**get_min_max_ocps**(*parameter_values*, *param=None*, *known_value='cyclable lithium capacity'*, *options=None*)

Calculate min/max open-circuit potentials given voltage limits, open-circuit potentials, etc defined by parameter_values

Parameters

- **parameter_values** (*pybamm.ParameterValues*) – The parameter values class that will be used for the simulation. Required for calculating appropriate initial open-circuit potentials.
- **param** (*pybamm.LithiumIonParameters*, optional) – The symbolic parameter set to use for the simulation. If not provided, the default parameter set will be used.
- **known_value** (*str, optional*) – The known value needed to complete the electrode SOH model. Can be "cyclable lithium capacity" (default) or "cell capacity".
- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model, see *pybamm.BatteryModelOptions*.

Returns
The min/max OCPs

Return type
Un_0, Un_100, Up_100, Up_0

Equivalent Circuit Model with Split OCV (SplitOCVR)

class pybamm.lithium_ion.**SplitOCVR**(*name='ECM with split OCV'*)

Basic Equivalent Circuit Model that uses two OCV functions for each electrode. This model is easily parameterizable with minimal parameters. This class differs from the :class: pybamm.equivalent_circuit.Thevenin() due to dual OCV functions to make up the voltage from each electrode.

Parameters
name (*str, optional*) – The name of the model.

Extends: *pybamm.models.base_model.BaseModel*

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

4.2.3 Lead Acid Models

Base Model

class pybamm.lead_acid.**BaseModel**(*options=None*, *name='Unnamed lead-acid model'*, *build=False*)

Overwrites default parameters from Base Model with default parameters for lead-acid models

Parameters

- **options** (*dict-like, optional*) – A dictionary of options to be passed to the model. If this is a dict (and not a subtype of dict), it will be processed by *pybamm.BatteryModelOptions* to ensure that the options are valid. If this is a subtype of dict, it is assumed that the options have already been processed and are valid. This allows for the use of custom options classes. The default options are given by *pybamm.BatteryModelOptions*.
- **name** (*str, optional*) – The name of the model. The default is "Unnamed battery model".
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

Extends: *pybamm.models.full_battery_models.base_battery_model.BaseBatteryModel*

property default_geometry

Returns a dictionary of the default geometry for the model, which is empty by default.

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

property default_var_pts

Returns a dictionary of the default variable points for the model, which is empty by default.

set_soc_variables()

Set variables relating to the state of charge.

Leading-Order Quasi-Static Model

class pybamm.lead_acid.**LOQS**(*options=None*, *name='LOQS model'*, *build=True*)

Leading-Order Quasi-Static model for lead-acid, from Sulzer *et al.*[1]. See *pybamm.lead_acid.BaseModel* for more details.

Extends: *pybamm.models.full_battery_models.lead_acid.base_lead_acid_model.BaseModel*

set_external_circuit_submodel()

Define how the external circuit defines the boundary conditions for the model, e.g. (not necessarily constant-) current, voltage, etc

References

Full Model

class pybamm.lead_acid.**Full**(*options=None*, *name='Full model'*, *build=True*)

Porous electrode model for lead-acid, from Sulzer *et al.*[1], based on the Newman-Tiedemann model. See *pybamm.lead_acid.BaseModel* for more details.

Extends: *pybamm.models.full_battery_models.lead_acid.base_lead_acid_model.BaseModel*

class pybamm.lead_acid.**BasicFull**(*name='Basic full model'*)

Porous electrode model for lead-acid, from Sulzer *et al.*[1].

This class differs from the *pybamm.lead_acid.Full* model class in that it shows the whole model in a single class. This comes at the cost of flexibility in comparing different physical effects, and in general the main DFN class should be used instead.

Parameters

name (*str, optional*) – The name of the model.

Extends: *pybamm.models.full_battery_models.lead_acid.base_lead_acid_model.BaseModel*

References

4.2.4 Equivalent Circuit Models

Thevenin Model

[1] Valentin Sulzer, S. Jon Chapman, Colin P. Please, David A. Howey, and Charles W. 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.

[1] Valentin Sulzer, S. Jon Chapman, Colin P. Please, David A. Howey, and Charles W. 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.

class pybamm.equivalent_circuit.**Thevenin**(*name='Thevenin Equivalent Circuit Model'*, *options=None*, *build=True*)

The classical Thevenin Equivalent Circuit Model of a battery as described in, for example, Barletta *et al.*[1].

This equivalent circuit model consists of an OCV element, a resistor element, and a number of RC elements (by default 1). The model is coupled to two lumped thermal models, one for the cell and one for the surrounding jig. Heat generation terms for each element follow equation (1) of Nieto *et al.*[2].

Parameters

- **name** (*str, optional*) – The name of the model. The default is "Thevenin Equivalent Circuit Model".
- **options** (*dict, optional*) – A dictionary of options to be passed to the model. The default is None. Possible options are:
  - **"number of rc elements"**
    [str] The number of RC elements to be added to the model. The default is 1.
  - **"calculate discharge energy": str**
    Whether to calculate the discharge energy, throughput energy and throughput capacity in addition to discharge capacity. Must be one of "true" or "false". "false" is the default, since calculating discharge energy can be computationally expensive for simple models like SPM.
  - **"diffusion element"**
    [str] Whether to include the diffusion element to the model. Must be one of "true" or "false". "false" is the default.
  - **"operating mode"**
    [str] Sets the operating mode for the model. This determines how the current is set. Can be:
    * "current" (default) : the current is explicity supplied
    * "voltage"/"power"/"resistance" : solve an algebraic equation for current such that voltage/power/resistance is correct
    * "differential power"/"differential resistance" : solve a differential equation for the power or resistance
    * "CCCV": a special implementation of the common constant-current constant-voltage charging protocol, via an ODE for the current
    * callable : if a callable is given as this option, the function defines the residual of an algebraic equation. The applied current will be solved for such that the algebraic constraint is satisfied.
- **build** (*bool, optional*) – Whether to build the model on instantiation. Default is True. Setting this option to False allows users to change any number of the submodels before building the complete model (submodels cannot be changed after the model is built).

[1] Giulio Barletta, Piera DiPrima, and Davide Papurello. Thévenin's battery model parameter estimation based on simulink. *Energies*, 15(17):6207, 2022.

[2] Nerea Nieto, Luis Diaz, Jon Gastelurrutia, Isabel Alava, Francisco Blanco, Juan Ramos, and Alejandro Rivas. Thermal modeling of large format lithium-ion cells. *Journal of the Electrochemical Society*, 160:A212–A217, 11 2012. doi:10.1149/2.042302jes.

Examples

```
>>> model = pybamm.equivalent_circuit.Thevenin()
>>> model.name
'Thevenin Equivalent Circuit Model'
```

Extends: *pybamm.models.base_model.BaseModel*

property default_geometry

Returns a dictionary of the default geometry for the model, which is empty by default.

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

property default_spatial_methods

Returns a dictionary of the default spatial methods for the model, which is empty by default.

property default_submesh_types

Returns a dictionary of the default submesh types for the model, which is empty by default.

property default_var_pts

Returns a dictionary of the default variable points for the model, which is empty by default.

set_external_circuit_submodel()

Define how the external circuit defines the boundary conditions for the model, e.g. (not necessarily constant-) current, voltage, etc

References

4.2.5 Submodels

Base Submodel

class pybamm.**BaseSubModel**(*param*, *domain=None*, *name='Unnamed submodel'*, *external=False*, *options=None*, *phase=None*)

The base class for all submodels. All submodels inherit from this class and must only provide public methods which overwrite those in this base class. Any methods added to a submodel that do not overwrite those in this bass class are made private with the prefix '_', providing a consistent public interface for all submodels.

Parameters

- **param** (*parameter class*) – The model parameter symbols
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **name** (*str*) – A string giving the name of the submodel
- **external** (*bool, optional*) – Whether the variables defined by the submodel will be provided externally by the users. Default is 'False'.
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is None).

`param`

The model parameter symbols.

Type
parameter class

`domain`

The domain of the submodel, could be either 'Negative', 'Positive', 'Separator', or None.

Type
str

`name`

The name of the submodel.

Type
str

`external`

A boolean flag indicating whether the variables defined by the submodel will be provided externally by the user. Set to False by default.

Type
bool

`options`

A dictionary or an instance of *pybamm.BatteryModelOptions* that stores configuration options for the submodel.

Type
dict or *pybamm.BatteryModelOptions*

`phase_name`

A string representing the phase of the submodel, which could be "primary", "secondary", or an empty string if there is only one phase.

Type
str

`phase`

The current phase of the submodel, which could be "primary", "secondary", or None.

Type
str or None

`boundary_conditions`

A dictionary mapping variables to their respective boundary conditions.

Type
dict

`variables`

A dictionary mapping variable names (strings) to expressions or objects that represent the useful variables for the submodel.

Type
dict

Extends: *pybamm.models.base_model.BaseModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

property domain_Domain

Returns a tuple containing the current domain and its capitalized form.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

get_parameter_info(*by_submodel=False*)

Extracts the parameter information and returns it as a dictionary. To get a list of all parameter-like objects without extra information, use `model.parameters`.

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

Active Material

Submodels for (loss of) active material

Base Model

class pybamm.active_material.**BaseModel**(*param*, *domain*, *options*, *phase='primary'*)

Base class for active material volume fraction

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – Additional options to pass to the model
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Constant Active Material

class pybamm.active_material.**Constant**(*param*, *domain*, *options*, *phase='primary'*)

Submodel for constant active material

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – Additional options to pass to the model
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.active_material.base_active_material.BaseModel*

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Loss of Active Material

class pybamm.active_material.**LossActiveMaterial**(*param*, *domain*, *options*, *x_average*, *phase*)

Submodel for varying active material volume fraction from Ai *et al.*[1] and Reniers *et al.*[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – Additional options to pass to the model
- **x_average** (*bool*) – Whether to use x-averaged variables (SPM, SPMe, etc) or full variables (DFN)

Extends: *pybamm.models.submodels.active_material.base_active_material.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

[1] Weilong Ai, Ludwig Kraft, Johannes Sturm, Andreas Jossen, and Billy Wu. Electrochemical thermal-mechanical modelling of stress inhomogeneity in lithium-ion pouch cells. *Journal of The Electrochemical Society*, 167(1):013512, 2019. doi:10.1149/2.0122001JES.

[2] Jorn M. Reniers, Grietus Mulder, and David A. Howey. Review and performance comparison of mechanical-chemical degradation models for lithium-ion batteries. *Journal of The Electrochemical Society*, 166(14):A3189, 2019. doi:10.1149/2.0281914jes.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Current Collector

Base Model

class pybamm.current_collector.**BaseModel**(*param*)

Base class for current collector submodels

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Effective Current collector Resistance models

class pybamm.current_collector.**EffectiveResistance**(*options=None*, *name='Effective resistance in current collector model'*)

A model which calculates the effective Ohmic resistance of the current collectors in the limit of large electrical conductivity. For details see Timms *et al.*[1]. Note that this formulation assumes uniform *potential* across the tabs. See pybamm.AlternativeEffectiveResistance2D for the formulation that assumes a uniform *current density* at the tabs (in 1D the two formulations are equivalent).

Parameters

- **options** (*dict*) – A dictionary of options to be passed to the model. The options that can be set are listed below.
  - **"dimensionality"**
    [int, optional] Sets the dimension of the current collector problem. Can be 1 (default) or 2.
- **name** (*str, optional*) – The name of the model.

Extends: pybamm.models.submodels.current_collector.effective_resistance_current_collector.BaseEffectiveResistance

post_process(*solution*, *param_values*, *V_av*, *I_av*)

Calculates the potentials in the current collector and the terminal voltage given the average voltage and current. Note: This takes in the *processed* V_av and I_av from a 1D simulation representing the average cell behaviour and returns a dictionary of processed potentials.

class pybamm.current_collector.**AlternativeEffectiveResistance2D**

A model which calculates the effective Ohmic resistance of the 2D current collectors in the limit of large electrical conductivity. This model assumes a uniform *current density* at the tabs and the solution is computed by first solving and auxilliary problem which is the related to the resistances.

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

Extends: pybamm.models.submodels.current_collector.effective_resistance_current_collector.BaseEffectiveResistance

post_process(*solution*, *param_values*, *V_av*, *I_av*)

Calculates the potentials in the current collector given the average voltage and current. Note: This takes in the *processed* V_av and I_av from a 1D simulation representing the average cell behaviour and returns a dictionary of processed potentials.

References

Uniform

class pybamm.current_collector.**Uniform**(*param*)

A submodel for uniform potential in the current collectors which is valid in the limit of fast conductivity in the current collectors.

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.current_collector.base_current_collector.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Potential Pair models

class pybamm.current_collector.**BasePotentialPair**(*param*)

A submodel for Ohm's law plus conservation of current in the current collectors. For details on the potential pair formulation see Timms *et al.*[1] and Marquis[2].

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

[2] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.current_collector.base_current_collector.BaseModel*

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic(variables)`

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions(variables)`

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`class pybamm.current_collector.PotentialPair2plus1D(param)`

Base class for a 2+1D potential pair model

Extends: *pybamm.models.submodels.current_collector.potential_pair.BasePotentialPair*

`set_boundary_conditions(variables)`

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`class pybamm.current_collector.PotentialPair1plus1D(param)`

Base class for a 1+1D potential pair model.

Extends: *pybamm.models.submodels.current_collector.potential_pair.BasePotentialPair*

`set_boundary_conditions(variables)`

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Convection

The convection submodels are split up into "through-cell", which is the x-direction problem in the electrode domains, and "transverse", which is the z-direction problem in the separator domain

Base Convection

class pybamm.convection.**BaseModel**(*param*, *options=None*)

Base class for convection submodels.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Through-cell Convection

Base Model

class pybamm.convection.through_cell.**BaseThroughCellModel**(*param*, *options=None*)

Base class for convection submodels in the through-cell direction.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.convection.base_convection.BaseModel*

No Convection

class pybamm.convection.through_cell.**NoConvection**(*param*, *options=None*)

A submodel for case where there is no convection.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.convection.through_cell.base_through_cell_convection.BaseThroughCellModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters

variables (*dict*) – The variables in the whole model.

Returns

The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Leading-Order Through-cell Model

`class pybamm.convection.through_cell.Explicit(param)`

A submodel for the leading-order approximation of pressure-driven convection

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.convection.through_cell.base_through_cell_convection.BaseThroughCellModel*

`get_coupled_variables(variables)`

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Full Through-cell Model

`class pybamm.convection.through_cell.Full(param)`

Submodel for the full model of pressure-driven convection

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.convection.through_cell.base_through_cell_convection.BaseThroughCellModel*

`get_coupled_variables(variables)`

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic(variables)`

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_boundary_conditions(variables)`

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions(variables)`

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Transverse Convection

Base Model

`class pybamm.convection.transverse.BaseTransverseModel(param, options=None)`

Base class for convection submodels in transverse directions.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.convection.base_convection.BaseModel*

No Transverse Convection

class pybamm.convection.transverse.**NoConvection**(*param*, *options=None*)

Submodel for no convection in transverse directions

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.convection.transverse.base_transverse_convection.BaseTransverseModel*

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Uniform Transverse Model

class pybamm.convection.transverse.**Uniform**(*param*)

Submodel for uniform convection in transverse directions

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.convection.transverse.base_transverse_convection.BaseTransverseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Full Transverse Convection

class pybamm.convection.transverse.**Full**(*param*)

Submodel for the full model of pressure-driven convection in transverse directions

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.convection.transverse.base_transverse_convection.BaseTransverseModel*

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Electrode

Electrode Base Model

class pybamm.electrode.**BaseElectrode**(*param*, *domain*, *options=None*, *set_positive_potential=True*)

Base class for electrode submodels.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **set_positive_potential** (*bool, optional*) – If True the battery model sets the positive potential based on the current. If False, the potential is specified by the user. Default is True.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Ohmic

Base Model

class pybamm.electrode.ohm.**BaseModel**(*param*, *domain*, *options=None*, *set_positive_potential=True*)

A base class for electrode submodels that employ Ohm's law.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrode.base_electrode.BaseElectrode*

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

Leading Order Model

class pybamm.electrode.ohm.**LeadingOrder**(*param*, *domain*, *options=None*, *set_positive_potential=True*)

An electrode submodel that employs Ohm's law the leading-order approximation to governing equations.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **set_positive_potential** (*bool, optional*) – If True the battery model sets the positve potential based on the current. If False, the potential is specified by the user. Default is True.

Extends: *pybamm.models.submodels.electrode.ohm.base_ohm.BaseModel*

`get_coupled_variables`(*variables*)

Returns variables which are derived from the fundamental variables in the model.

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Composite Model

`class` pybamm.electrode.ohm.`Composite`(*param*, *domain*, *options=None*)

An explicit composite leading and first order solution to solid phase current conservation with ohm's law. Note that the returned current density is only the leading order approximation.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'Negative electrode' or 'Positive electrode'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrode.ohm.base_ohm.BaseModel*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Full Model

`class` pybamm.electrode.ohm.`Full`(*param*, *domain*, *options=None*)

Full model of electrode employing Ohm's law.

Extends: *pybamm.models.submodels.electrode.ohm.base_ohm.BaseModel*

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrode.ohm.base_ohm.BaseModel*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic`(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Surface Form

class pybamm.electrode.ohm.**SurfaceForm**(*param*, *domain*, *options=None*)

A submodel for the electrode with Ohm's law in the surface potential formulation.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrode.ohm.base_ohm.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Explicit potential drop for lithium metal

class pybamm.electrode.ohm.**LithiumMetalExplicit**(*param*, *domain*, *options=None*)

Explicit model for potential drop across a lithium metal electrode.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: pybamm.models.submodels.electrode.ohm.li_metal.LithiumMetalBaseModel

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Electrolyte Conductivity

Base Electrolyte Conductivity Submodel

class pybamm.electrolyte_conductivity.**BaseElectrolyteConductivity**(*param*, *domain=None*, *options=None*)

Base class for conservation of charge in the electrolyte.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str, optional*) – The domain in which the model holds
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

Leading Order Model

class pybamm.electrolyte_conductivity.**LeadingOrder**(*param*, *domain=None*, *options=None*)

Leading-order model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations. (Leading refers to leading-order in the asymptotic reduction)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str, optional*) – The domain in which the model holds
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_conductivity.base_electrolyte_conductivity.BaseElectrolyteConductivity*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters

variables (*dict*) – The variables in the whole model.

Returns

The variables created in this submodel which depend on variables in other submodels.

Return type

dict

Composite Model

class pybamm.electrolyte_conductivity.**Composite**(*param*, *domain=None*, *options=None*)

Base class for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str, optional*) – The domain in which the model holds
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **higher_order_terms** (*str*) – What kind of higher-order terms to use ('composite' or 'first-order')

Extends: *pybamm.models.submodels.electrolyte_conductivity.base_electrolyte_conductivity.BaseElectrolyteConductivity*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Integrated Model

class pybamm.electrolyte_conductivity.**Integrated**(*param*, *domain=None*, *options=None*)

Integrated model for conservation of charge in the electrolyte derived from integrating the Stefan-Maxwell constitutive equations, from Brosa Planella *et al.*[1].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str, optional*) – The domain in which the model holds
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_conductivity.base_electrolyte_conductivity.BaseElectrolyteConductivity*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

[1] Ferran Brosa Planella, Muhammad Sheikh, and W. Dhammika Widanage. Systematic derivation and validation of a reduced thermal-electrochemical model for lithium-ion batteries using asymptotic methods. *Electrochimica Acta*, 388:138524, 2021. doi:10.1016/j.electacta.2021.138524.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Full Model

class pybamm.electrolyte_conductivity.**Full**(*param*, *options=None*)

Full model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations. (Full refers to unreduced by asymptotic methods)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_conductivity.base_electrolyte_conductivity.BaseElectrolyteConductivity*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

property algebraic

Returns a dictionary mapping expressions (variables) to expressions that represent the algebraic equations of the model.

property boundary_conditions

Returns a dictionary mapping expressions (variables) to expressions representing the boundary conditions of the model.

check_algebraic_equations(*post_discretisation*)

Check that the algebraic equations are well-posed. After discretisation, there must be at least one StateVector in each algebraic equation.

check_discretised_or_discretise_inplace_if_0D()

Discretise model if it isn't already discretised This only works with purely 0D models, as otherwise the mesh and spatial method should be specified by the user

check_ics_bcs()

Check that the initial and boundary conditions are well-posed.

check_no_repeated_keys()

Check that no equation keys are repeated.

check_well_determined(*post_discretisation*)

Check that the model is not under- or over-determined.

check_well_posedness(*post_discretisation=False*)

Check that the model is well-posed by executing the following tests: - Model is not over- or underdetermined, by comparing keys and equations in rhs and algebraic. Overdetermined if more equations than variables, underdetermined if more variables than equations. - There is an initial condition in

self.initial_conditions for each variable/equation pair in self.rhs - There are appropriate boundary conditions in self.boundary_conditions for each variable/equation pair in self.rhs and self.algebraic

Parameters

post_discretisation (*boolean*) – A flag indicating tests to be skipped after discretisation

property concatenated_algebraic

Returns the concatenated algebraic equations for the model after discretisation.

property concatenated_initial_conditions

Returns the initial conditions for all variables after discretization, providing the initial values for the state variables.

property concatenated_rhs

Returns the concatenated right-hand side (RHS) expressions for the model after discretisation.

property coupled_variables

Returns a dictionary mapping strings to expressions representing variables needed by the model but whose equations were set by other models.

property default_geometry

Returns a dictionary of the default geometry for the model, which is empty by default.

property default_parameter_values

Returns the default parameter values for the model (an empty set of parameters by default).

property default_quick_plot_variables

Returns the default variables for quick plotting (None by default).

property default_solver

Returns the default solver for the model, based on whether it is an ODE/DAE or algebraic model.

property default_spatial_methods

Returns a dictionary of the default spatial methods for the model, which is empty by default.

property default_submesh_types

Returns a dictionary of the default submesh types for the model, which is empty by default.

property default_var_pts

Returns a dictionary of the default variable points for the model, which is empty by default.

classmethod deserialise(*properties: dict*)

Create a model instance from a serialised object.

property domain_Domain

Returns a tuple containing the current domain and its capitalized form.

property events

Returns a dictionary mapping expressions (variables) to expressions that represent the initial conditions for the state variables.

export_casadi_objects(*variable_names*, *input_parameter_order=None*)

Export the constituent parts of the model (rhs, algebraic, initial conditions, etc) as casadi objects.

Parameters

- **variable_names** (*list*) – Variables to be exported alongside the model structure

- **input_parameter_order** (*list, optional*) – Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)

Returns
casadi_dict – Dictionary of {str: casadi object} pairs representing the model in casadi format

Return type
dict

generate(*filename*, *variable_names*, *input_parameter_order=None*, *cg_options=None*)

Generate the model in C, using CasADi.

Parameters

- **filename** (*str*) – Name of the file to which to save the code
- **variable_names** (*list*) – Variables to be exported alongside the model structure
- **input_parameter_order** (*list, optional*) – Order in which the input parameters should be stacked. If input_parameter_order=None and len(self.input_parameters) > 1, a ValueError is raised (this helps to avoid accidentally using the wrong order)
- **cg_options** (*dict*) – Options to pass to the code generator. See https://web.casadi.org/docs/#generating-c-code

property geometry

Returns the geometry of the model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

get_parameter_info(*by_submodel=False*)

Extracts the parameter information and returns it as a dictionary. To get a list of all parameter-like objects without extra information, use `model.parameters`.

info(*symbol_name*)

Provides helpful summary information for a symbol.

Parameters
symbol_name (*str*)

property initial_conditions

Returns a dictionary mapping expressions (variables) to expressions that represent the initial conditions for the state variables.

property input_parameters

Returns a list of all input parameter symbols used in the model.

property jacobian

Returns the Jacobian matrix for the model, computed automatically if *use_jacobian* is True.

property jacobian_algebraic

Returns the Jacobian matrix for the algebraic part of the model, computed automatically during solver setup if *use_jacobian* is True.

property jacobian_rhs

Returns the Jacobian matrix for the right-hand side (RHS) part of the model, computed if *use_jacobian* is True.

latexify(*filename=None*, *newline=True*, *output_variables=None*)

Converts all model equations in latex.

Parameters

- **filename** (*str (optional)*) – Accepted file formats - any image format, pdf and tex Default is None, When None returns all model equations in latex If not None, returns all model equations in given file format.
- **newline** (*bool (optional)*) – Default is True, If True, returns every equation in a new line. If False, returns the list of all the equations.
- **model** (*Load*)
- **pybamm.lithium_ion.SPM()** (*>>> model =*)
- **png** (*This will returns all model equations in*)
- **doctest** (*>>> model.latexify(newline=False) #*)
- **latex** (*This will return all the model equations in*)
- **doctest**
- **equations** (*This will return first five model*)
- **doctest**
- **equations**
- **model.latexify(newline=False)[1** (*>>>*)

property mass_matrix

Returns the mass matrix for the system of differential equations after discretisation.

property mass_matrix_inv

Returns the inverse of the mass matrix for the differential equations after discretisation.

new_copy()

Creates a copy of the model, explicitly copying all the mutable attributes to avoid issues with shared objects.

property options

Returns the model options dictionary that allows customization of the model's behavior.

property param

Returns a dictionary to store parameter values for the model.

property parameters

Returns a list of all parameter symbols used in the model.

print_parameter_info(*by_submodel=False*)

Print parameter information in a formatted table from a dictionary of parameters

Parameters

by_submodel (*bool, optional*) – Whether to print the parameter info sub-model wise or not (default False)

process_parameters_and_discretise(*symbol*, *parameter_values*, *disc*)

Process parameters and discretise a symbol using supplied parameter values and discretisation. Note: care should be taken if using spatial operators on dimensional symbols. Operators in pybamm are written in non-dimensional form, so may need to be scaled by the appropriate length scale. It is recommended to use this method on non-dimensional symbols.

Parameters

- **symbol** (*pybamm.Symbol*) – Symbol to be processed
- **parameter_values** (*pybamm.ParameterValues*) – The parameter values to use during processing
- **disc** (*pybamm.Discretisation*) – The discrisation to use

Returns

Processed symbol

Return type

pybamm.Symbol

property rhs

Returns a dictionary mapping expressions (variables) to expressions that represent the right-hand side (RHS) of the model's differential equations.

save_model(*filename=None*, *mesh=None*, *variables=None*)

Write out a discretised model to a JSON file

Parameters

- **filename** (*str, optional*)
- **provided** (*The desired name of the JSON file. If no name is*)
- **created** (*one will be*)
- **name** (*based on the model*)
- **datetime.** (*and the current*)

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions_from(*solution*, *inplace=True*, *return_type='model'*)

Update initial conditions with the final states from a Solution object or from a dictionary. This assumes that, for each variable in self.initial_conditions, there is a corresponding variable in the solution with the same name and size.

Parameters

- **solution** (*pybamm.Solution*, or dict) – The solution to use to initialize the model
- **inplace** (*bool, optional*) – Whether to modify the model inplace or create a new model (default True)
- **return_type** (*str, optional*) – Whether to return the model (default) or initial conditions ("ics")

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

update(**submodels*)

Update model to add new physics from submodels

Parameters
submodel (iterable of *pybamm.BaseModel*) – The submodels from which to create new model

property variables

Returns a dictionary mapping strings to expressions representing the model's useful variables.

property variables_and_events

Returns a dictionary containing both models variables and events.

Surface Form

Full Model

class pybamm.electrolyte_conductivity.surface_potential_form.**FullDifferential**(*param*, *domain*, *options=None*)

Full model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations and where capacitance is present. (Full refers to unreduced by asymptotic methods)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: pybamm.models.submodels.electrolyte_conductivity.surface_potential_form.full_surface_form_conductivity.BaseModel

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

class pybamm.electrolyte_conductivity.surface_potential_form.**FullAlgebraic**(*param*, *domain*, *options=None*)

Full model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations. (Full refers to unreduced by asymptotic methods)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: pybamm.models.submodels.electrolyte_conductivity.surface_potential_form.full_surface_form_conductivity.BaseModel

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

Leading Order Model

class pybamm.electrolyte_conductivity.surface_potential_form.**LeadingOrderDifferential**(*param*, *domain*, *options=None*)

Leading-order model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations employing the surface potential difference formulation and where capacitance is present.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: pybamm.models.submodels.electrolyte_conductivity.surface_potential_form.leading_surface_form_conductivity.BaseLeadingOrderSurfaceForm

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

class pybamm.electrolyte_conductivity.surface_potential_form.**LeadingOrderAlgebraic**(*param*, *domain*, *options=None*)

Leading-order model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations employing the surface potential difference formulation.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: pybamm.models.submodels.electrolyte_conductivity.surface_potential_form.leading_surface_form_conductivity.BaseLeadingOrderSurfaceForm

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

Explicit Model

class pybamm.electrolyte_conductivity.surface_potential_form.**Explicit**(*param*, *domain*, *options*)

Class for deriving surface potential difference variables from the electrode and electrolyte potentials

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain in which the model holds
- **options** (*dict*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_conductivity.base_electrolyte_conductivity.BaseElectrolyteConductivity*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Electrolyte Diffusion

Base Electrolyte Diffusion Submodel

class pybamm.electrolyte_diffusion.**BaseElectrolyteDiffusion**(*param*, *options=None*)

Base class for conservation of mass in the electrolyte.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Constant Concentration

class pybamm.electrolyte_diffusion.**ConstantConcentration**(*param*, *options=None*)

Class for constant concentration of electrolyte

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_diffusion.base_electrolyte_diffusion.BaseElectrolyteDiffusion*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_boundary_conditions(variables)`

We provide boundary conditions even though the concentration is constant so that the gradient of the concentration has the correct shape after discretisation.

Leading Order Model

`class pybamm.electrolyte_diffusion.LeadingOrder(param)`

Class for conservation of mass in the electrolyte employing the Stefan-Maxwell constitutive equations. (Leading refers to leading order of asymptotic reduction)

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.electrolyte_diffusion.base_electrolyte_diffusion.BaseElectrolyteDiffusion*

`get_coupled_variables(variables)`

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Full Model

`class` pybamm.electrolyte_diffusion.`Full`(*param*, *options=None*)

Class for conservation of mass in the electrolyte employing the Stefan-Maxwell constitutive equations. (Full refers to unreduced by asymptotic methods)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.electrolyte_diffusion.base_electrolyte_diffusion.BaseElectrolyteDiffusion*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

External circuit

Models to enforce different boundary conditions (as imposed by an imaginary external circuit) such as constant current, constant voltage, constant power, or any other relationship between the current and voltage. "Current control" enforces these directly through boundary conditions, while "Function control" submodels add an algebraic equation (for the current) and hence can be used to set any variable to be constant.

Discharge and throughput variables

Calculates the discharge and throughput variables (capacity and power) for the battery.

`class` pybamm.external_circuit.`DischargeThroughput`(*param*, *options*)

Model calculate discharge and throughput capacity and energy.

Extends: pybamm.models.submodels.external_circuit.base_external_circuit.BaseModel

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Explicit control external circuit

Current is explicitly specified, either by a function or in terms of other variables.

class pybamm.external_circuit.**ExplicitCurrentControl**(*param*, *options*)

External circuit with current control.

Extends: pybamm.models.submodels.external_circuit.base_external_circuit.BaseModel

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

class pybamm.external_circuit.**ExplicitPowerControl**(*param*, *options*)

External circuit with current set explicitly to hit target power.

Extends: pybamm.models.submodels.external_circuit.base_external_circuit.BaseModel

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

class pybamm.external_circuit.**ExplicitResistanceControl**(*param*, *options*)

External circuit with current set explicitly to hit target resistance.

Extends: pybamm.models.submodels.external_circuit.base_external_circuit.BaseModel

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Function control external circuit

class pybamm.external_circuit.**FunctionControl**(*param*, *external_circuit_function*, *options*, *control='algebraic'*)

External circuit with an arbitrary function, implemented as a control on the current either via an algebraic equation, or a differential equation.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **external_circuit_function** (*callable*) – The function that controls the current
- **options** (*dict*) – Dictionary of options to use for the submodel
- **control** (*str, optional*) – The type of control to use. Must be one of 'algebraic' (default) or 'differential'.

Extends: pybamm.models.submodels.external_circuit.base_external_circuit.BaseModel

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic`(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

class pybamm.external_circuit.**VoltageFunctionControl**(*param*, *options*)

External circuit with voltage control, implemented as an extra algebraic equation.

Extends: *pybamm.models.submodels.external_circuit.function_control_external_circuit.FunctionControl*

class pybamm.external_circuit.**PowerFunctionControl**(*param*, *options*, *control='algebraic'*)

External circuit with power control.

Extends: *pybamm.models.submodels.external_circuit.function_control_external_circuit.FunctionControl*

class pybamm.external_circuit.**ResistanceFunctionControl**(*param*, *options*, *control='algebraic'*)

External circuit with resistance control.

Extends: *pybamm.models.submodels.external_circuit.function_control_external_circuit.FunctionControl*

class pybamm.external_circuit.**CCCVFunctionControl**(*param*, *options*)

External circuit with constant-current constant-voltage control, as implemented in Mohtat *et al.*[1].

References

Extends: *pybamm.models.submodels.external_circuit.function_control_external_circuit.FunctionControl*

[1] Peyman Mohtat, Sravan Pannala, Valentin Sulzer, Jason B Siegel, and Anna G Stefanopoulou. An algorithmic safety vest for li-ion batteries during fast charging. *arXiv preprint arXiv:2108.07833*, 2021.

Interface

Interface Base Model

class pybamm.interface.**BaseInterface**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Base class for interfacial currents

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Total Interfacial Current Model

class pybamm.interface.**TotalInterfacialCurrent**(*param*, *chemistry*, *options*)

Total interfacial current, summing up contributions from all reactions

Parameters

- **param** – model parameters
- **chemistry** (*str*) – The name of the battery chemistry whose reactions need to be summed up
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

get_coupled_variables(*variables*)

Get variables associated with interfacial current over the whole cell domain This function also creates the "total source term" variables by summing all the reactions.

Interface Utilisation

Utilisation Base Model

class pybamm.interface.interface_utilisation.**BaseModel**(*param*, *domain*, *options*)

Base class for interface utilisation

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Constant Utilisation

class pybamm.interface.interface_utilisation.**Constant**(*param*, *domain*, *options*)

Submodel for constant interface utilisation

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.interface.interface_utilisation.base_utilisation.BaseModel*

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns

The variables created by the submodel which are independent of variables in other submodels.

Return type

dict

CurrentDriven Utilisation

class pybamm.interface.interface_utilisation.**CurrentDriven**(*param*, *domain*, *options*, *reaction_loc*)

Current-driven ODE for interface utilisation

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **reaction_loc** (*str*) – Where the reaction happens: "x-average" (SPM, SPMe, etc), "full electrode" (full DFN), or "interface" (half-cell model)

Extends: *pybamm.models.submodels.interface.interface_utilisation.base_utilisation.BaseModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Full Utilisation

`class` pybamm.interface.interface_utilisation.`Full`(*param*, *domain*, *options*)

Submodel for constant interface utilisation

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'negative' or 'positive'
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.interface.interface_utilisation.base_utilisation.BaseModel*

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Kinetics

Base Kinetics

`class` pybamm.kinetics.`BaseKinetics`(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Base submodel for kinetics

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic`(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Butler Volmer

class pybamm.kinetics.**SymmetricButlerVolmer**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Submodel which implements the symmetric forward Butler-Volmer equation:

$$j = 2 * j_0(c) * \sinh(ne * F * \eta_r(c)/RT)$$

Parameters

- **param** (*parameter class*) – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

class pybamm.kinetics.**AsymmetricButlerVolmer**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Submodel which implements the asymmetric forward Butler-Volmer equation

Parameters

- **param** (*parameter class*) – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

Diffusion-limited

class pybamm.kinetics.**DiffusionLimited**(*param*, *domain*, *reaction*, *options*, *order*)

Submodel for diffusion-limited kinetics

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **order** (*str*) – The order of the model ("leading" or "full")

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Linear

class pybamm.kinetics.**Linear**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Submodel which implements linear kinetics. Valid for small overpotentials/currents.

Parameters

- **param** (*parameter class*) – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

Marcus

class pybamm.kinetics.**Marcus**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Submodel which implements Marcus kinetics.

Parameters

- **param** (*parameter class*) – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

NoReaction

class pybamm.kinetics.**NoReaction**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Base submodel for when no reaction occurs

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*

- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Tafel

class pybamm.kinetics.**ForwardTafel**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Base submodel which implements the forward Tafel equation:

$$j = u * j_0(c) * \exp((ne * alpha * F * \eta_r(c)/RT)$$

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

MSMR Butler Volmer

class pybamm.kinetics.**MSMRButlerVolmer**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Submodel which implements the forward Butler-Volmer equation in the MSMR formulation in which the interfacial current density is summed over all reactions.

Parameters

- **param** (*parameter class*) – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.kinetics.base_kinetics.BaseKinetics*

Total Main Kinetics

class pybamm.kinetics.**TotalMainKinetics**(*param*, *domain*, *reaction*, *options*)

Class summing up contributions to the main (e.g. intercalation) reaction for cases with primary, secondary, ... reactions e.g. silicon-graphite

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Inverse Kinetics

Inverse Butler-Volmer

class pybamm.kinetics.**InverseButlerVolmer**(*param*, *domain*, *reaction*, *options=None*)

A submodel that implements the inverted form of the Butler-Volmer relation to solve for the reaction overpotential.

Parameters

- **param** – Model parameters
- **domain** (*iter of str, optional*) – The domain(s) in which to compute the interfacial current.
- **reaction** (*str*) – The name of the reaction being implemented

- **options** (*dict*) – A dictionary of options to be passed to the model. In this case "SEI film resistance" is the important option. See *pybamm.BaseBatteryModel*

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Lithium Plating

Base Plating

class pybamm.lithium_plating.**BasePlating**(*param*, *domain*, *options=None*, *phase='primary'*)

Base class for lithium plating models, from O'Kane *et al.*[1] and O'Kane *et al.*[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

[1] Simon E. J. O'Kane, Ian D. Campbell, Mohamed W. J. Marzook, Gregory J. Offer, and Monica Marinescu. Physical origin of the differential voltage minimum associated with lithium plating in li-ion batteries. *Journal of The Electrochemical Society*, 167(9):090540, may 2020. URL: https://doi.org/10.1149/1945-7111/ab90ac, doi:10.1149/1945-7111/ab90ac.

[2] Simon E. J. O'Kane, Weilong Ai, Ganesh Madabattula, Diego Alonso-Alvarez, Robert Timms, Valentin Sulzer, Jacqueline Sophie Edge, Billy Wu, Gregory J. Offer, and Monica Marinescu. Lithium-ion battery degradation: how to model it. *Phys. Chem. Chem. Phys.*, 24:7909–7922, 2022. URL: http://dx.doi.org/10.1039/D2CP00417H, doi:10.1039/D2CP00417H.

References

No Plating

class pybamm.lithium_plating.**NoPlating**(*param*, *domain*, *options=None*, *phase='primary'*)

Base class for no lithium plating/stripping.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.interface.lithium_plating.base_plating.BasePlating*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Plating

class pybamm.lithium_plating.**Plating**(*param*, *domain*, *x_average*, *options*, *phase='primary'*)

Class for lithium plating, from O'Kane *et al.*[1] and O'Kane *et al.*[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **x_average** (*bool*) – Whether to use x-averaged variables (SPM, SPMe, etc) or full variables (DFN)
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

[1] Simon E. J. O'Kane, Ian D. Campbell, Mohamed W. J. Marzook, Gregory J. Offer, and Monica Marinescu. Physical origin of the differential voltage minimum associated with lithium plating in li-ion batteries. *Journal of The Electrochemical Society*, 167(9):090540, may 2020. URL: https://doi.org/10.1149/1945-7111/ab90ac, doi:10.1149/1945-7111/ab90ac.

[2] Simon E. J. O'Kane, Weilong Ai, Ganesh Madabattula, Diego Alonso-Alvarez, Robert Timms, Valentin Sulzer, Jacqueline Sophie Edge, Billy Wu, Gregory J. Offer, and Monica Marinescu. Lithium-ion battery degradation: how to model it. *Phys. Chem. Chem. Phys.*, 24:7909–7922, 2022. URL: http://dx.doi.org/10.1039/D2CP00417H, doi:10.1039/D2CP00417H.

Extends: *pybamm.models.submodels.interface.lithium_plating.base_plating.BasePlating*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Open-circuit potential models

Base Open Circuit Potential

`class` pybamm.open_circuit_potential.`BaseOpenCircuitPotential`(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Base class for open-circuit potentials

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel

- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **reaction** (*str*) – The name of the reaction being implemented
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

Current Sigmoid Open Circuit Potential

class pybamm.open_circuit_potential.**CurrentSigmoidOpenCircuitPotential**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Single Open Circuit Potential

class pybamm.open_circuit_potential.**SingleOpenCircuitPotential**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

MSMR Open Circuit Potential

class pybamm.open_circuit_potential.**MSMROpenCircuitPotential**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Class for open-circuit potential within the Multi-Species Multi-Reaction framework Baker and Verbrugge[1]. The thermodynamic model is presented in Verbrugge *et al.*[2], along with parameter values for a number of substitutional materials.

Extends: *pybamm.models.submodels.interface.open_circuit_potential.base_ocp.BaseOpenCircuitPotential*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Wycisk Open Circuit Potential

class pybamm.open_circuit_potential.**WyciskOpenCircuitPotential**(*param*, *domain*, *reaction*, *options*, *phase='primary'*)

Class for open-circuit potential with hysteresis based on the approach outlined by Wycisk :footcite:t:'Wycisk2022'. This approach employs a differential capacity hysteresis state variable. The decay and switching of the hysteresis state is tunable via two additional parameters. The hysteresis state is updated based on the current and the differential capacity.

Extends: *pybamm.models.submodels.interface.open_circuit_potential.base_ocp.BaseOpenCircuitPotential*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

[1] Daniel R Baker and Mark W Verbrugge. Multi-species, multi-reaction model for porous intercalation electrodes: part i. model formulation and a perturbation solution for low-scan-rate, linear-sweep voltammetry of a spinel lithium manganese oxide electrode. *Journal of The Electrochemical Society*, 165(16):A3952, 2018.

[2] Mark Verbrugge, Daniel Baker, Brian Koch, Xingcheng Xiao, and Wentian Gu. Thermodynamic model for substitutional materials: application to lithiated graphite, spinel manganese oxide, iron phosphate, and layered nickel-manganese-cobalt oxide. *Journal of The Electrochemical Society*, 164(11):E3243, 2017.

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

SEI models

SEI Base Model

class pybamm.sei.**BaseModel**(*param*, *domain*, *options*, *phase='primary'*, *cracks=False*)

Base class for SEI models.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict*) – A dictionary of options to be passed to the model.
- **phase** (*str, optional*) – Phase of the particle (default is "primary")
- **cracks** (*bool, optional*) – Whether this is a submodel for standard SEI or SEI on cracks

Extends: *pybamm.models.submodels.interface.base_interface.BaseInterface*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Constant SEI

class pybamm.sei.**ConstantSEI**(*param*, *domain*, *options*, *phase='primary'*)

Class for SEI with constant thickness.

Note that there is no SEI current, so we don't need to update the "sum of interfacial current densities" variables from *pybamm.interface.BaseInterface*

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict*) – A dictionary of options to be passed to the model.
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.interface.sei.base_sei.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

No SEI

class pybamm.sei.**NoSEI**(*param*, *domain*, *options*, *phase='primary'*, *cracks=False*)

Class for no SEI.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict*) – A dictionary of options to be passed to the model.
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

- **cracks** (*bool, optional*) – Whether this is a submodel for standard SEI or SEI on cracks

Extends: *pybamm.models.submodels.interface.sei.base_sei.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

SEI Growth

class pybamm.sei.**SEIGrowth**(*param*, *domain*, *reaction_loc*, *options*, *phase='primary'*, *cracks=False*)

Class for SEI growth.

Most of the models are from Section 5.6.4 of Marquis[1] and references therein.

The ec reaction limited model is from Yang *et al.*[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **reaction_loc** (*str*) – Where the reaction happens: "x-average" (SPM, SPMe, etc), "full electrode" (full DFN), or "interface" (half-cell model)
- **options** (*dict*) – A dictionary of options to be passed to the model.
- **phase** (*str, optional*) – Phase of the particle (default is "primary")
- **cracks** (*bool, optional*) – Whether this is a submodel for standard SEI or SEI on cracks

Extends: *pybamm.models.submodels.interface.sei.base_sei.BaseModel*

[1] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

[2] Xiao Guang Yang, Yongjun Leng, Guangsheng Zhang, Shanhai Ge, and Chao Yang Wang. Modeling of lithium plating induced aging of lithium-ion batteries: transition from linear to nonlinear aging. *Journal of Power Sources*, 360:28–40, 2017. doi:10.1016/j.jpowsour.2017.05.110.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Total SEI

class pybamm.sei.**TotalSEI**(*param*, *domain*, *options*, *cracks=False*)

Class summing up contributions to the SEI reaction for cases with primary, secondary, ... reactions e.g. silicon-graphite

Parameters

- **param** – model parameters
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Oxygen Diffusion

Base Model

class pybamm.oxygen_diffusion.**BaseModel**(*param*)

Base class for conservation of mass of oxygen.

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Full Model

class pybamm.oxygen_diffusion.**Full**(*param*)

Class for conservation of mass of oxygen. (Full refers to unreduced by asymptotic methods) In this model, extremely fast oxygen kinetics in the negative electrode imposes zero oxygen concentration there, and so the oxygen variable only lives in the separator and positive electrode. The boundary condition at the negative electrode/ separator interface is homogeneous Dirichlet.

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.oxygen_diffusion.base_oxygen_diffusion.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Leading Order Model

`class` pybamm.oxygen_diffusion.`LeadingOrder`(*param*)

Class for conservation of mass of oxygen. (Leading refers to leading order of asymptotic reduction)

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.oxygen_diffusion.base_oxygen_diffusion.BaseModel*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of

whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

No Oxygen

`class` pybamm.oxygen_diffusion.**NoOxygen**(*param*)

Class for when there is no oxygen

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.oxygen_diffusion.base_oxygen_diffusion.BaseModel*

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Particle

Particle Base Model

`class` pybamm.particle.**BaseParticle**(*param*, *domain*, *options*, *phase='primary'*)

Base class for molar conservation in particles.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel

- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

class pybamm.particle.**TotalConcentration**(*param*, *domain*, *options*, *phase='primary'*)

Class to calculate total particle concentrations

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.particle.base_particle.BaseParticle*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Fickian Diffusion

class pybamm.particle.**FickianDiffusion**(*param*, *domain*, *options*, *phase='primary'*, *x_average=False*)

Class for molar conservation in particles, employing Fick's law

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")
- **x_average** (*bool*) – Whether the particle concentration is averaged over the x-direction

Extends: *pybamm.models.submodels.particle.base_particle.BaseParticle*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Polynomial Profile

class pybamm.particle.**PolynomialProfile**(*param*, *domain*, *options*, *phase='primary'*)

Class for molar conservation in particles employing Fick's law, assuming a polynomial concentration profile in r, and allowing variation in the electrode domain. Model equations from Subramanian *et al.*[1].

[1] Venkat R. Subramanian, Vinten D. Diwakar, and Deepak Tapriyal. Efficient macro-micro scale coupled modeling of batteries. *Journal of The Electrochemical Society*, 152(10):A2002, 2005. doi:10.1149/1.2032427.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.particle.base_particle.BaseParticle*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_algebraic`(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

X-averaged Polynomial Profile

class pybamm.particle.**XAveragedPolynomialProfile**(*param*, *domain*, *options*, *phase='primary'*)

Class for molar conservation in a single x-averaged particle employing Fick's law, with an assumed polynomial concentration profile in r. Model equations from Subramanian *et al.*[1].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.particle.polynomial_profile.PolynomialProfile*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_algebraic(*variables*)

A method to set the differential equations which do not contain a time derivative. Note: this method modifies the state of self.algebraic. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

[1] Venkat R. Subramanian, Vinten D. Diwakar, and Deepak Tapriyal. Efficient macro-micro scale coupled modeling of batteries. *Journal of The Electrochemical Society*, 152(10):A2002, 2005. doi:10.1149/1.2032427.

set_initial_conditions(*variables*)

For single or x-averaged particle models, initial conditions can't depend on x or r so we take the r- and x-average of the initial conditions.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

MSMR Diffusion

class pybamm.particle.**MSMRDiffusion**(*param*, *domain*, *options*, *phase='primary'*, *x_average=False*)

Class for molar conservation in particles within the Multi-Species Multi-Reaction framework Baker and Verbrugge[1]. The thermodynamic model is presented in Verbrugge *et al.*[2], along with parameter values for a number of substitutional materials.

In this submodel, the stoichiometry depends on the potential in the particle and the temperature, so dUdT is not used. See *:meth: `pybamm.LithiumIonParameters.dUdT* for more explanation.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")
- **x_average** (*bool*) – Whether the particle concentration is averaged over the x-direction

Extends: *pybamm.models.submodels.particle.base_particle.BaseParticle*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

[1] Daniel R Baker and Mark W Verbrugge. Multi-species, multi-reaction model for porous intercalation electrodes: part i. model formulation and a perturbation solution for low-scan-rate, linear-sweep voltammetry of a spinel lithium manganese oxide electrode. *Journal of The Electrochemical Society*, 165(16):A3952, 2018.

[2] Mark Verbrugge, Daniel Baker, Brian Koch, Xingcheng Xiao, and Wentian Gu. Thermodynamic model for substitutional materials: application to lithiated graphite, spinel manganese oxide, iron phosphate, and layered nickel-manganese-cobalt oxide. *Journal of The Electrochemical Society*, 164(11):E3243, 2017.

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

class pybamm.particle.**MSMRStoichiometryVariables**(*param*, *domain*, *options*, *phase='primary'*, *x_average=False*)

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Particle Cracking

Base Particle Mechanics Model

class pybamm.particle_mechanics.**BaseMechanics**(*param*, *domain*, *options*, *phase='primary'*)

Base class for particle mechanics models, referenced from Ai *et al.*[1] and Deshpande *et al.*[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*dict, optional*) – Dictionary of either the electrode for "positive" or "Negative"
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

References

Crack Propagation Model

class pybamm.particle_mechanics.**CrackPropagation**(*param*, *domain*, *x_average*, *options*, *phase='primary'*)

Cracking behaviour in electrode particles. See Ai *et al.*[1] for mechanical model (thickness change) and Deshpande *et al.*[2] for cracking model.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- **x_average** (*bool*) – Whether to use x-averaged variables (SPM, SPMe, etc) or full variables (DFN)
- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.particle_mechanics.base_mechanics.BaseMechanics*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters

variables (*dict*) – The variables in the whole model.

[1] Weilong Ai, Ludwig Kraft, Johannes Sturm, Andreas Jossen, and Billy Wu. Electrochemical thermal-mechanical modelling of stress inhomogeneity in lithium-ion pouch cells. *Journal of The Electrochemical Society*, 167(1):013512, 2019. doi:10.1149/2.0122001JES.

[2] Rutooj Deshpande, Mark Verbrugge, Yang-Tse Cheng, John Wang, and Ping Liu. Battery cycle life prediction with coupled chemical degradation and fatigue mechanics. *Journal of the Electrochemical Society*, 159(10):A1730, 2012. doi:10.1149/2.049210jes.

[1] Weilong Ai, Ludwig Kraft, Johannes Sturm, Andreas Jossen, and Billy Wu. Electrochemical thermal-mechanical modelling of stress inhomogeneity in lithium-ion pouch cells. *Journal of The Electrochemical Society*, 167(1):013512, 2019. doi:10.1149/2.0122001JES.

[2] Rutooj Deshpande, Mark Verbrugge, Yang-Tse Cheng, John Wang, and Ping Liu. Battery cycle life prediction with coupled chemical degradation and fatigue mechanics. *Journal of the Electrochemical Society*, 159(10):A1730, 2012. doi:10.1149/2.049210jes.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Swelling Only Model

class pybamm.particle_mechanics.**SwellingOnly**(*param*, *domain*, *options*, *phase='primary'*)

Class for swelling only (no cracking), from Ai *et al.*[1].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'

[1] Weilong Ai, Ludwig Kraft, Johannes Sturm, Andreas Jossen, and Billy Wu. Electrochemical thermal-mechanical modelling of stress inhomogeneity in lithium-ion pouch cells. *Journal of The Electrochemical Society*, 167(1):013512, 2019. doi:10.1149/2.0122001JES.

- **options** (*dict*) – A dictionary of options to be passed to the model. See *pybamm.BaseBatteryModel*
- **phase** (*str, optional*) – Phase of the particle (default is "primary")

Extends: *pybamm.models.submodels.particle_mechanics.base_mechanics.BaseMechanics*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

References

Porosity

Base Model

class pybamm.porosity.**BaseModel**(*param*, *options*)

Base class for porosity

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Constant Porosity

class pybamm.porosity.**Constant**(*param*, *options*)

Submodel for constant porosity

Parameters
param (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.models.submodels.porosity.base_porosity.BaseModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Reaction-driven Model

class pybamm.porosity.**ReactionDriven**(*param*, *options*, *x_average*)

Reaction-driven porosity changes as a multiple of SEI/plating thicknesses

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict*) – Options dictionary passed from the full model
- **x_average** (*bool*) – Whether to use x-averaged variables (SPM, SPMe, etc) or full variables (DFN)

Extends: *pybamm.models.submodels.porosity.base_porosity.BaseModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Reaction-driven Model as an ODE

class pybamm.porosity.**ReactionDrivenODE**(*param*, *options*, *x_average*)

Reaction-driven porosity changes as an ODE

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict*) – Options dictionary passed from the full model
- **x_average** (*bool*) – Whether to use x-averaged variables (SPM, SPMe, etc) or full variables (DFN)

Extends: *pybamm.models.submodels.porosity.base_porosity.BaseModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Thermal

Base Thermal

class pybamm.thermal.**BaseThermal**(*param*, *options=None*, *x_average=False*)

Base class for thermal effects

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Isothermal Model

class pybamm.thermal.isothermal.**Isothermal**(*param*, *options=None*, *x_average=False*)

Class for isothermal submodel.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.thermal.base_thermal.BaseThermal*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

Lumped Model

class pybamm.thermal.lumped.**Lumped**(*param*, *options=None*, *x_average=False*)

Class for lumped thermal submodel. For more information see Timms *et al.*[1] and Marquis[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.thermal.base_thermal.BaseThermal*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

[2] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

Parameters
variables (*dict*) – The variables in the whole model.

References

Pouch Cell

One Dimensional Model

class pybamm.thermal.pouch_cell.x_full.**OneDimensionalX**(*param*, *options=None*, *x_average=False*)

Class for one-dimensional (x-direction) thermal submodel. Note: this model assumes infinitely large electrical and thermal conductivity in the current collectors, so that the contribution to the Ohmic heating from the current collectors is zero and the boundary conditions are applied at the edges of the electrodes (at x=0 and x=1, in non-dimensional coordinates). For more information see Timms *et al.*[1] and Marquis[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.thermal.base_thermal.BaseThermal*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_boundary_conditions(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

[2] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Thermal Model for "1+1D" Pouch Cell

`class` pybamm.thermal.pouch_cell.`CurrentCollector1D`(*param*, *options=None*, *x_average=True*)

Class for one-dimensional thermal submodel for use in the "1+1D" pouch cell model. The thermal model is averaged in the x-direction and is therefore referred to as 'x-lumped'. For more information see Timms *et al.*[1] and Marquis[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.thermal.base_thermal.BaseThermal*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

[2] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_boundary_conditions`(*variables*)

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Thermal Model for "2+1D" Pouch Cell

`class` pybamm.thermal.pouch_cell.`CurrentCollector2D`(*param*, *options=None*, *x_average=True*)

Class for two-dimensional thermal submodel for use in the "2+1D" pouch cell model. The thermal model is averaged in the x-direction and is therefore referred to as 'x-lumped'. For more information see Timms *et al.*[1] and Marquis[2].

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.thermal.base_thermal.BaseThermal*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

[1] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

[2] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables()`

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_boundary_conditions(variables)`

A method to set the boundary conditions for the submodel. Note: this method modifies the state of self.boundary_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_initial_conditions(variables)`

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs(variables)`

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

References

Transport Efficiency

Base Model

`class pybamm.transport_efficiency.BaseModel(param, component, options=None)`

Base class for transport_efficiency

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

Bruggeman Transport Efficiency Model

class pybamm.transport_efficiency.**Bruggeman**(*param*, *component*, *options=None*)

Submodel for Bruggeman transport_efficiency, Bruggeman[1]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Cation-Exchange Membrane Transport Efficiency Model

class pybamm.transport_efficiency.**CationExchangeMembrane**(*param*, *component*, *options=None*)

Submodel for Cation Exchange Membrane transport_efficiency, Bruggeman[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

[1] Von DAG Bruggeman. Berechnung verschiedener physikalischer konstanten von heterogenen substanzen. i. dielektrizitätskonstanten und leitfähigkeiten der mischkörper aus isotropen substanzen. *Annalen der physik*, 416(7):636–664, 1935.

[1] Von DAG Bruggeman. Berechnung verschiedener physikalischer konstanten von heterogenen substanzen. i. dielektrizitätskonstanten und leitfähigkeiten der mischkörper aus isotropen substanzen. *Annalen der physik*, 416(7):636–664, 1935.

[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Heterogeneous Catalyst Transport Efficiency Model

class pybamm.transport_efficiency.**HeterogeneousCatalyst**(*param*, *component*, *options=None*)

Submodel for Heterogeneous Catalyst transport_efficiency Beeckman[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Hyperbola of Revolution Transport Efficiency Model

class pybamm.transport_efficiency.**HyperbolaOfRevolution**(*param*, *component*, *options=None*)

Submodel for Hyperbola of revolution transport_efficiency Petersen[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

[1] JW Beeckman. Mathematical description of heterogeneous materials. *Chemical engineering science*, 45(8):2603–2610, 1990.
[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.
[1] EE Petersen. Diffusion in a pore of varying cross section. *AIChE Journal*, 4(3):343–345, 1958.
[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Ordered Packing Transport Efficiency Model

class pybamm.transport_efficiency.**OrderedPacking**(*param*, *component*, *options=None*)

Submodel for Ordered Packing transport_efficiency Akanni *et al.*[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Overlapping Spheres Transport Efficiency Model

[1] KA Akanni, JW Evans, and IS Abramson. Effective transport coefficients in heterogeneous media. *Chemical Engineering Science*, 42(8):1945–1954, 1987.

[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.

class pybamm.transport_efficiency.**OverlappingSpheres**(*param*, *component*, *options=None*)

Submodel for Overlapping Spheres transport_efficiency Weissberg[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Random Overlapping Cylinders Transport Efficiency Model

class pybamm.transport_efficiency.**RandomOverlappingCylinders**(*param*, *component*, *options=None*)

Submodel for Random Overlapping Cylinders transport_efficiency, Tomadakis and Sotirchos[1], Shen and Chen[2]

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

[1] Harold L Weissberg. Effective diffusion coefficient in porous media. *Journal of Applied Physics*, 34(9):2636–2639, 1963.
[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.
[1] Manolis M Tomadakis and Stratis V Sotirchos. Transport properties of random arrays of freely overlapping cylinders with various orientation distributions. *The Journal of chemical physics*, 98(1):616–626, 1993.
[2] Lihua Shen and Zhangxin Chen. Critical review of the impact of tortuosity on diffusion. *Chemical Engineering Science*, 62(14):3748–3755, 2007.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

References

Tortuosity Factor Transport Efficiency Model

class pybamm.transport_efficiency.**TortuosityFactor**(*param*, *component*, *options=None*)

Submodel for user supplied tortuosity factor transport_efficiency

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **component** (*str*) – The material for the model ('electrolyte' or 'electrode').
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.transport_efficiency.base_transport_efficiency.BaseModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

Equivalent Circuit Elements

OCV Element

class pybamm.equivalent_circuit_elements.**OCVElement**(*param*, *options=None*)

Open-circuit Voltage (OCV) element for equivalent circuits.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Resistor Element

class pybamm.equivalent_circuit_elements.**ResistorElement**(*param*, *options=None*)

Resistor element for equivalent circuits.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other

submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

RC Element

class pybamm.equivalent_circuit_elements.**RCElement**(*param*, *element_number*, *options=None*)

Parallel Resistor-Capacitor (RC) element for equivalent circuits.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **element_number** (*int*) – The number of the element (i.e. whether it is the first, second, third, etc. element)
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

get_fundamental_variables()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

set_initial_conditions(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

`set_rhs`(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Thermal SubModel

`class` pybamm.equivalent_circuit_elements.`ThermalSubModel`(*param*, *options=None*)

Thermal SubModel for use with equivalent circuits.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

`get_coupled_variables`(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

`get_fundamental_variables`()

A public method that creates and returns the variables in a submodel which can be created independent of other submodels. For example, the electrolyte concentration variables can be created independent of whether any other variables have been defined in the model. As a rule, if a variable can be created without variables from other submodels, then it should be placed in this method.

Returns
The variables created by the submodel which are independent of variables in other submodels.

Return type
dict

`set_initial_conditions`(*variables*)

A method to set the initial conditions for the submodel. Note: this method modifies the state of self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

set_rhs(*variables*)

A method to set the right hand side of the differential equations which contain a time derivative. Note: this method modifies the state of self.rhs. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

Voltage Model

class pybamm.equivalent_circuit_elements.**VoltageModel**(*param*, *options=None*)

Voltage model for use with equivalent circuits. This model is used to calculate the voltage and total overpotentials from the other elements in the circuit.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **options** (*dict, optional*) – A dictionary of options to be passed to the model.

Extends: *pybamm.models.submodels.base_submodel.BaseSubModel*

add_events_from(*variables*)

A method to set events related to the state of submodel variable. Note: this method modifies the state of self.events. Unless overwritten by a submodel, the default behaviour of 'pass' is used as implemented in *pybamm.BaseSubModel*.

Parameters
variables (*dict*) – The variables in the whole model.

get_coupled_variables(*variables*)

A public method that creates and returns the variables in a submodel which require variables in other submodels to be set first. For example, the exchange current density requires the concentration in the electrolyte to be created before it can be created. If a variable can be created independent of other submodels then it should be created in 'get_fundamental_variables' instead of this method.

Parameters
variables (*dict*) – The variables in the whole model.

Returns
The variables created in this submodel which depend on variables in other submodels.

Return type
dict

4.3 Parameters

4.3.1 Parameter Values

class pybamm.**ParameterValues**(*values*)

The parameter values for a simulation.

Note that this class does not inherit directly from the python dictionary class as this causes issues with saving and loading simulations.

Parameters
values (*dict or string*) – Explicit set of parameters, or reference to an inbuilt parameter set If string and matches one of the inbuilt parameter sets, returns that parameter set.

Examples

```
>>> values = {"some parameter": 1, "another parameter": 2}
>>> param = pybamm.ParameterValues(values)
>>> param["some parameter"]
1
>>> param = pybamm.ParameterValues("Marquis2019")
>>> param["Reference temperature [K]"]
298.15
```

copy()

Returns a copy of the parameter values. Makes sure to copy the internal dictionary.

static create_from_bpx(*filename*, *target_soc: float = 1*)

Parameters

- **filename** (*str*) – The filename of the BPX file
- **target_soc** (*float, optional*) – Target state of charge. Must be between 0 and 1. Default is 1.

Returns

A parameter values object with the parameters in the bpx file

Return type

ParameterValues

static create_from_bpx_obj(*bpx_obj*, *target_soc: float = 1*)

Parameters

- **bpx_obj** (*dict*) – A dictionary containing the parameters in the BPX format
- **target_soc** (*float, optional*) – Target state of charge. Must be between 0 and 1. Default is 1.

Returns

A parameter values object with the parameters in the bpx file

Return type

ParameterValues

evaluate(*symbol*, *inputs=None*)

Process and evaluate a symbol.

Parameters

symbol (*pybamm.Symbol*) – Symbol or Expression tree to evaluate

Returns

The evaluated symbol

Return type

number or array

get(*key*, *default=None*)

Return item corresponding to key if it exists, otherwise return default

items()

Get the items of the dictionary

keys()

Get the keys of the dictionary

print_evaluated_parameters(*evaluated_parameters*, *output_file*)

Print a dictionary of evaluated parameters to an output file

Parameters

- **evaluated_parameters** (*defaultdict*) – The evaluated parameters, for further processing if needed
- **output_file** (*string, optional*) – The file to print parameters to. If None, the parameters are not printed, and this function simply acts as a test that all the parameters can be evaluated

print_parameters(*parameters*, *output_file=None*)

Return dictionary of evaluated parameters, and optionally print these evaluated parameters to an output file.

Parameters

- **parameters** (class or dict containing *pybamm.Parameter* objects) – Class or dictionary containing all the parameters to be evaluated
- **output_file** (*string, optional*) – The file to print parameters to. If None, the parameters are not printed, and this function simply acts as a test that all the parameters can be evaluated, and returns the dictionary of evaluated parameters.

Returns

evaluated_parameters – The evaluated parameters, for further processing if needed

Return type

defaultdict

Notes

A C-rate of 1 C is the current required to fully discharge the battery in 1 hour, 2 C is current to discharge the battery in 0.5 hours, etc

process_boundary_conditions(*model*)

Process boundary conditions for a model Boundary conditions are dictionaries {"left": left bc, "right": right bc} in general, but may be imposed on the tabs (or *not* on the tab) for a small number of variables, e.g. {"negative tab": neg. tab bc, "positive tab": pos. tab bc "no tab": no tab bc}.

process_geometry(*geometry*)

Assign parameter values to a geometry (inplace).

Parameters

geometry (*dict*) – Geometry specs to assign parameter values to

process_model(*unprocessed_model*, *inplace=True*)

Assign parameter values to a model. Currently inplace, could be changed to return a new model.

Parameters

- **unprocessed_model** (*pybamm.BaseModel*) – Model to assign parameter values for
- **inplace** (*bool, optional*) – If True, replace the parameters in the model in place. Otherwise, return a new model with parameter values set. Default is True.

Raises
pybamm.ModelError – If an empty model is passed (*model.rhs = {}* and *model.algebraic = {}* and *model.variables = {}*)

process_symbol(*symbol*)

Walk through the symbol and replace any Parameter with a Value. If a symbol has already been processed, the stored value is returned.

Parameters
symbol (*pybamm.Symbol*) – Symbol or Expression tree to set parameters for

Returns
symbol – Symbol with Parameter instances replaced by Value

Return type
pybamm.Symbol

search(*key*, *print_values=True*)

Search dictionary for keys containing 'key'.

See *pybamm.FuzzyDict.search()*.

set_initial_ocps(*initial_value*, *param=None*, *known_value='cyclable lithium capacity'*, *inplace=True*, *options=None*)

Set the initial OCP of each electrode, based on the initial SOC or voltage

set_initial_stoichiometries(*initial_value*, *param=None*, *known_value='cyclable lithium capacity'*, *inplace=True*, *options=None*, *inputs=None*, *tol=1e-06*)

Set the initial stoichiometry of each electrode, based on the initial SOC or voltage

set_initial_stoichiometry_half_cell(*initial_value*, *param=None*, *known_value='cyclable lithium capacity'*, *inplace=True*, *options=None*, *inputs=None*)

Set the initial stoichiometry of the working electrode, based on the initial SOC or voltage

update(*values*, *check_conflict=False*, *check_already_exists=True*, *path=''*)

Update parameter dictionary, while also performing some basic checks.

Parameters

- **values** (*dict*) – Dictionary of parameter values to update parameter dictionary with
- **check_conflict** (*bool, optional*) – Whether to check that a parameter in *values* has not already been defined in the parameter class when updating it, and if so that its value does not change. This is set to True during initialisation, when parameters are combined from different sources, and is False by default otherwise
- **check_already_exists** (*bool, optional*) – Whether to check that a parameter in *values* already exists when trying to update it. This is to avoid cases where an intended change in the parameters is ignored due a typo in the parameter name, and is True by default but can be manually overridden.
- **path** (*string, optional*) – Path from which to load functions

values()

Get the values of the dictionary

4.3.2 Geometric Parameters

class pybamm.**GeometricParameters**(*options=None*)

Standard geometric parameters

Extends: `pybamm.parameters.base_parameters.BaseParameters`

4.3.3 Electrical Parameters

class pybamm.**ElectricalParameters**

Standard electrical parameters

Extends: `pybamm.parameters.base_parameters.BaseParameters`

4.3.4 Thermal Parameters

class pybamm.**ThermalParameters**

Standard thermal parameters

Extends: `pybamm.parameters.base_parameters.BaseParameters`

4.3.5 Lithium-ion Parameters

class pybamm.**LithiumIonParameters**(*options=None*)

Standard parameters for lithium-ion battery models

Parameters

options (*dict, optional*) – A dictionary of options to be passed to the parameters, see *pybamm.BatteryModelOptions*.

Extends: `pybamm.parameters.base_parameters.BaseParameters`

4.3.6 Lead-Acid Parameters

class pybamm.**LeadAcidParameters**

Standard Parameters for lead-acid battery models

Extends: `pybamm.parameters.base_parameters.BaseParameters`

4.3.7 Parameters Sets

PyBaMM provides *pre-defined parameters* for common chemistries, as well as, a growing set of *third-party parameter sets*.

class pybamm.parameters.parameter_sets.**ParameterSets**

Dict-like interface for accessing registered pybamm parameter sets. Access via `pybamm.parameter_sets`

Examples

Listing available parameter sets:

```
>>> import pybamm
>>> list(pybamm.parameter_sets)
['Ai2020', 'Chayambuka2022', ...]
```

Get the docstring for a parameter set:

```
>>> print(pybamm.parameter_sets.get_docstring("Ai2020"))

Parameters for the Enertech cell (Ai2020), from the papers :footcite:t:`Ai2019`,
:footcite:t:`Rieger2016` and references therein.
...
```

See also: *Adding Parameter Sets*

Extends: `collections.abc.Mapping`

get_docstring(*key*)

Return the docstring for the `key` parameter set

Adding Parameter Sets

Parameter sets can be added to PyBaMM by creating a python package, and registering a entry point to `pybamm_parameter_sets`. At a minimum, the package (`cell_parameters`) should consist of the following:

```
cell_parameters
├── pyproject.toml        # and/or setup.cfg, setup.py
└── src
    └── cell_parameters
        └── cell_alpha.py
```

The actual parameter set is defined within `cell_alpha.py`, as shown below. For an example, see the Marquis2019 parameter sets.

```python
import pybamm


def get_parameter_values():
    """Doc string for cell-alpha"""
    return {
        "chemistry": "lithium_ion",
        "citation": "@book{van1995python, title={Python reference manual}}",
        # ...
    }
```

Then register `get_parameter_values` to `pybamm_parameter_sets` in `pyproject.toml`:

```
[project.entry-points.pybamm_parameter_sets]
cell_alpha = "cell_parameters.cell_alpha:get_parameter_values"
```

If you are using `setup.py` or `setup.cfg` to setup your package, please see SetupTools' documentation for registering entry points.

If you're willing to open-source your parameter set, let us know, and we can add an entry to *Third-Party Parameter Sets*.

Third-Party Parameter Sets

Registered a new parameter set to `pybamm_parameter_sets`? Let us know, and we'll update our list.

Bundled Parameter Sets

PyBaMM provides pre-defined parameter sets for several common chemistries, listed below. See *Adding Parameter Sets* for information on registering new parameter sets with PyBaMM.

Lead-acid Parameter Sets

{% for k,v in parameter_sets.items() if v.chemistry == "lead_acid" %} {{k}} ———————————- {{ parameter_sets.get_docstring(k) }} {% endfor %}

Lithium-ion Parameter Sets

{% for k,v in parameter_sets.items() if v.chemistry == "lithium_ion" %} {{k}} ———————————— {{ parameter_sets.get_docstring(k) }} {% endfor %}

4.3.8 Process Parameter Data

pybamm.parameters.**process_1D_data**(*name*, *path=None*)

Process 1D data from a csv file

pybamm.parameters.**process_2D_data**(*name*, *path=None*)

Process 2D data from a JSON file

pybamm.parameters.**process_2D_data_csv**(*name*, *path=None*)

Process 2D data from a csv file. Assumes data is in the form of a three columns and that all data points lie on a regular grid. The first column is assumed to be the 'slowest' changing variable and the second column the 'fastest' changing variable, which is the C convention for indexing multidimensional arrays (as opposed to the Fortran convention where the 'fastest' changing variable comes first).

Parameters

- **name** (*str*) – The name to be given to the function
- **path** (*str*) – The path to the file where the three dimensional data is stored.

Returns

formatted_data – A tuple containing the name of the function and the data formatted correctly for use within three-dimensional interpolants.

Return type

tuple

pybamm.parameters.**process_3D_data_csv**(*name*, *path=None*)

Process 3D data from a csv file. Assumes data is in the form of four columns and that all data points lie on a regular grid. The first column is assumed to be the 'slowest' changing variable and the third column the 'fastest' changing variable, which is the C convention for indexing multidimensional arrays (as opposed to the Fortran convention where the 'fastest' changing variable comes first).

Parameters

- **name** (*str*) – The name to be given to the function
- **path** (*str*) – The path to the file where the three dimensional data is stored.

Returns

formatted_data – A tuple containing the name of the function and the data formatted correctly for use within three-dimensional interpolants.

Return type

tuple

4.4 Geometry

4.4.1 Geometry

class pybamm.**Geometry**(*geometry*)

A geometry class to store the details features of the cell geometry.

The values assigned to each domain are dictionaries containing the spatial variables in that domain, along with expression trees giving their min and maximum extents. For example, the following dictionary structure would represent a Geometry with a single domain "negative electrode", defined using the variable *x_n* which has a range from 0 to the pre-defined parameter *l_n*.

```
{"negative electrode": {x_n: {"min": pybamm.Scalar(0), "max": l_n}}}
```

Parameters

geometries (*dict*) – The dictionary to create the geometry with

Extends: builtins.dict

property parameters

Returns all the parameters in the geometry

print_parameter_info()

Prints all the parameters' information

4.4.2 Battery Geometry

pybamm.**battery_geometry**(*include_particles=True*, *options=None*, *form_factor='pouch'*)

A convenience function to create battery geometries.

Parameters

- **include_particles** (*bool, optional*) – Whether to include particle domains. Can be True (default) or False.
- **options** (*dict, optional*) – Dictionary of model options. Necessary for "particle-size geometry", relevant for lithium-ion chemistries.
- **form_factor** (*str, optional*) – The form factor of the cell. Can be "pouch" (default) or "cylindrical".

Returns

A geometry class for the battery

Return type

pybamm.Geometry

4.5 Meshes

4.5.1 Meshes

class pybamm.**Mesh**(*geometry*, *submesh_types*, *var_pts*)

Mesh contains a list of submeshes on each subdomain.

Parameters

- **geometry** – contains the geometry of the problem.

- **submesh_types** (*dict*) – contains the types of submeshes to use (e.g. Uniform1DSubMesh)
- **submesh_pts** (*dict*) – contains the number of points on each subdomain

Extends: `builtins.dict`

add_ghost_meshes()

Create meshes for potential ghost nodes on either side of each submesh, using self.submeshclass This will be useful for calculating the gradient with Dirichlet BCs.

combine_submeshes(**submeshnames*)

Combine submeshes into a new submesh, using self.submeshclass Raises pybamm.DomainError if submeshes to be combined do not match up (edges are not aligned).

Parameters
submeshnames (*list of str*) – The names of the submeshes to be combined

Returns
submesh – A new submesh with the class defined by self.submeshclass

Return type
`self.submeshclass`

class pybamm.**SubMesh**

Base submesh class. Contains the position of the nodes, the number of mesh points, and (optionally) information about the tab locations.

class pybamm.**MeshGenerator**(*submesh_type*, *submesh_params=None*)

Base class for mesh generator objects that are used to generate submeshes.

Parameters

- **submesh_type** (*pybamm.SubMesh*) – The type of submesh to use (e.g. Uniform1DSubMesh).
- **submesh_params** (*dict, optional*) – Contains any parameters required by the submesh.

4.5.2 0D Sub Mesh

class pybamm.**SubMesh0D**(*position*, *npts=None*)

0D submesh class. Contains the position of the node.

Parameters

- **position** (*dict*) – A dictionary that contains the position of the 0D submesh (a signle point) in space
- **npts** (*dict, optional*) – Number of points to be used. Included for compatibility with other meshes, but ignored by this mesh class

Extends: *pybamm.meshes.meshes.SubMesh*

4.5.3 1D Sub Meshes

class pybamm.**SubMesh1D**(*edges*, *coord_sys*, *tabs=None*)

1D submesh class. Contains the position of the nodes, the number of mesh points, and (optionally) information about the tab locations.

Parameters

- **edges** (*array_like*) – An array containing the points corresponding to the edges of the submesh
- **coord_sys** (*string*) – The coordinate system of the submesh
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs

Extends: *pybamm.meshes.meshes.SubMesh*

class pybamm.**Uniform1DSubMesh**(*lims*, *npts*)

A class to generate a uniform submesh on a 1D domain

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.

Extends: *pybamm.meshes.one_dimensional_submeshes.SubMesh1D*

class pybamm.**Exponential1DSubMesh**(*lims*, *npts*, *side='symmetric'*, *stretch=None*)

A class to generate a submesh on a 1D domain in which the points are clustered close to one or both of boundaries using an exponential formula on the interval [a,b].

If side is "left", the gridpoints are given by

$$x_k = (b-a) + \frac{\mathrm{e}^{\alpha k/N} - 1}{\mathrm{e}^{\alpha} - 1} + a,$$

for k = 1, …, N, where N is the number of nodes.

Is side is "right", the gridpoints are given by

$$x_k = (b-a) + \frac{\mathrm{e}^{-\alpha k/N} - 1}{\mathrm{e}^{-\alpha} - 1} + a,$$

for k = 1, …, N.

If side is "symmetric", the first half of the interval is meshed using the gridpoints

$$x_k = (b/2-a) + \frac{\mathrm{e}^{\alpha k/N} - 1}{\mathrm{e}^{\alpha} - 1} + a,$$

for k = 1, …, N. The grid spacing is then reflected to contruct the grid on the full interval [a,b].

In the above, alpha is a stretching factor. As the number of gridpoints tends to infinity, the ratio of the largest and smallest grid cells tends to exp(alpha).

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **side** (*str, optional*) – Whether the points are clustered near to the left or right boundary, or both boundaries. Can be "left", "right" or "symmetric". Default is "symmetric"
- **stretch** (*float, optional*) – The factor (alpha) which appears in the exponential. If side is "symmetric" then the default stretch is 1.15. If side is "left" or "right" then the default stretch is 2.3.

Extends: *pybamm.meshes.one_dimensional_submeshes.SubMesh1D*

class pybamm.**Chebyshev1DSubMesh**(*lims*, *npts*, *tabs=None*)

A class to generate a submesh on a 1D domain using Chebyshev nodes on the interval (a, b), given by

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos(\frac{2k-1}{2N}\pi),$$

for k = 1, ..., N, where N is the number of nodes. Note: this mesh then appends the boundary edges, so that the mesh edges are given by

$$a < x_1 < ... < x_N < b.$$

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs

Extends: *pybamm.meshes.one_dimensional_submeshes.SubMesh1D*

class pybamm.**UserSupplied1DSubMesh**(*lims*, *npts*, *edges=None*)

A class to generate a submesh on a 1D domain from a user supplied array of edges.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **edges** (*array_like*) – The array of points which correspond to the edges of the mesh.

Extends: *pybamm.meshes.one_dimensional_submeshes.SubMesh1D*

4.5.4 2D Sub Meshes

class pybamm.**ScikitSubMesh2D**(*edges*, *coord_sys*, *tabs*)

2D submesh class. Contains information about the 2D finite element mesh. Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **edges** (*array_like*) – An array containing the points corresponding to the edges of the submesh
- **coord_sys** (*string*) – The coordinate system of the submesh
- **tabs** (*dict, optional*) – A dictionary that contains information about the size and location of the tabs

Extends: *pybamm.meshes.meshes.SubMesh*

on_boundary(*y*, *z*, *tab*)

A method to get the degrees of freedom corresponding to the subdomains for the tabs.

class pybamm.**ScikitUniform2DSubMesh**(*lims*, *npts*)

Contains information about the 2D finite element mesh with uniform grid spacing (can be different spacing in y and z). Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable

Extends: *pybamm.meshes.scikit_fem_submeshes.ScikitSubMesh2D*

class pybamm.**ScikitExponential2DSubMesh**(*lims*, *npts*, *side='top'*, *stretch=2.3*)

Contains information about the 2D finite element mesh generated by taking the tensor product of a uniformly spaced grid in the y direction, and a unequally spaced grid in the z direction in which the points are clustered close to the top boundary using an exponential formula on the interval [a,b]. The gridpoints in the z direction are given by

$$z_k = (b-a) + \frac{\exp -\alpha k/N - 1}{\exp -\alpha - 1} + a,$$

for k = 1, ..., N, where N is the number of nodes. Here alpha is a stretching factor. As the number of gridpoints tends to infinity, the ratio of the largest and smallest grid cells tends to exp(alpha).

Note: in the future this will be extended to allow points to be clustered near any of the boundaries.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable
- **side** (*str, optional*) – Whether the points are clustered near to a particular boundary. At present, can only be "top". Default is "top".
- **stretch** (*float, optional*) – The factor (alpha) which appears in the exponential. Default is 2.3.

Extends: *pybamm.meshes.scikit_fem_submeshes.ScikitSubMesh2D*

class pybamm.**ScikitChebyshev2DSubMesh**(*lims*, *npts*)

Contains information about the 2D finite element mesh generated by taking the tensor product of two 1D meshes which use Chebyshev nodes on the interval (a, b), given by

$$x_k = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos(\frac{2k-1}{2N}\pi),$$

for k = 1, ..., N, where N is the number of nodes. Note: this mesh then appends the boundary edgess, so that the 1D mesh edges are given by

$$a < x_1 < ... < x_N < b.$$

Note: This class only allows for the use of piecewise-linear triangular finite elements.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of each spatial variable
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable

Extends: *pybamm.meshes.scikit_fem_submeshes.ScikitSubMesh2D*

class pybamm.**UserSupplied2DSubMesh**(*lims*, *npts*, *y_edges=None*, *z_edges=None*)

A class to generate a tensor product submesh on a 2D domain by using two user supplied vectors of edges: one for the y-direction and one for the z-direction. Note: this mesh should be created using `UserSupplied2DSubMeshGenerator`.

Parameters

- **lims** (*dict*) – A dictionary that contains the limits of the spatial variables
- **npts** (*dict*) – A dictionary that contains the number of points to be used on each spatial variable. Note: the number of nodes (located at the cell centres) is npts, and the number of edges is npts+1.
- **y_edges** (*array_like*) – The array of points which correspond to the edges in the y direction of the mesh.
- **z_edges** (*array_like*) – The array of points which correspond to the edges in the z direction of the mesh.

Extends: *pybamm.meshes.scikit_fem_submeshes.ScikitSubMesh2D*

4.6 Discretisation and spatial methods

4.6.1 Discretisation

class pybamm.**Discretisation**(*mesh=None*, *spatial_methods=None*, *check_model=True*, *remove_independent_variables_from_rhs=False*)

The discretisation class, with methods to process a model and replace Spatial Operators with Matrices and Variables with StateVectors

Parameters

- **mesh** (pybamm.Mesh) – contains all submeshes to be used on each domain
- **spatial_methods** (*dict*) – a dictionary of the spatial methods to be used on each domain. The keys correspond to the model domains and the values to the spatial method.
- **check_model** (*bool, optional*) – If True, model checks are performed after discretisation. For large systems these checks can be slow, so can be skipped by setting this option to False. When developing, testing or debugging it is recommended to leave this option as True as it may help to identify any errors. Default is True.
- **remove_independent_variables_from_rhs** (*bool, optional*) – If True, model checks to see whether any variables from the RHS are used in any other equation. If a variable meets all of the following criteria (not used anywhere in the model, len(rhs)>1), then the variable is moved to be explicitly integrated when called by the solution object. Default is False.

check_model(*model*)

Perform some basic checks to make sure the discretised model makes sense.

check_tab_conditions(*symbol*, *bcs*)

Check any boundary conditions applied on "negative tab", "positive tab" and "no tab". For 1D current collector meshes, these conditions are converted into boundary conditions on "left" (tab at z=0) or "right" (tab at z=l_z) depending on the tab location stored in the mesh. For 2D current collector meshes, the boundary conditions can be applied on the tabs directly.

Parameters

- **symbol** (*pybamm.expression_tree.symbol.Symbol*) – The symbol on which the boundary conditions are applied.
- **bcs** (*dict*) – The dictionary of boundary conditions (a dict of {side: equation}).

Returns
The dictionary of boundary conditions, with the keys changed to "left" and "right" where necessary.

Return type
dict

create_mass_matrix(*model*)

Creates mass matrix of the discretised model. Note that the model is assumed to be of the form M*y_dot = f(t,y), where M is the (possibly singular) mass matrix.

Parameters
model (*pybamm.BaseModel*) – Discretised model. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})

Returns

- *pybamm.Matrix* – The mass matrix
- *pybamm.Matrix* – The inverse of the ode part of the mass matrix (required by solvers which only accept the ODEs in explicit form)

process_boundary_conditions(*model*)

Discretise model boundary_conditions, also converting keys to ids

Parameters
model (*pybamm.BaseModel*) – Model to dicretise. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})

Returns
Dictionary of processed boundary conditions

Return type
dict

process_dict(*var_eqn_dict*, *ics=False*)

Discretise a dictionary of {variable: equation}, broadcasting if necessary (can be model.rhs, model.algebraic, model.initial_conditions or model.variables).

Parameters

- **var_eqn_dict** (*dict*) – Equations ({variable: equation} dict) to dicretise (can be model.rhs, model.algebraic, model.initial_conditions or model.variables)
- **ics** (*bool, optional*) – Whether the equations are initial conditions. If True, the equations are scaled by the reference value of the variable, if given

Returns
new_var_eqn_dict – Discretised equations

Return type
dict

process_initial_conditions(*model*)

Discretise model initial_conditions.

Parameters
model (*pybamm.BaseModel*) – Model to dicretise. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})

Returns
Tuple of processed_initial_conditions (dict of initial conditions) and concatenated_initial_conditions (numpy array of concatenated initial conditions)

Return type
tuple

process_model(*model*, *inplace=True*)

Discretise a model. Currently inplace, could be changed to return a new model.

Parameters

- **model** (*pybamm.BaseModel*) – Model to dicretise. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})
- **inplace** (*bool, optional*) – If True, discretise the model in place. Otherwise, return a new discretised model. Default is True.

Returns
model_disc – The discretised model. Note that if `inplace` is True, model will have also been discretised in place so model == model_disc. If `inplace` is False, model != model_disc

Return type
pybamm.BaseModel

Raises
pybamm.ModelError – If an empty model is passed (*model.rhs = {}* and *model.algebraic = {}* and *model.variables = {}*)

process_rhs_and_algebraic(*model*)

Discretise model equations - differential ('rhs') and algebraic.

Parameters
model (*pybamm.BaseModel*) – Model to dicretise. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})

Returns
Tuple of processed_rhs (dict of processed differential equations), processed_concatenated_rhs, processed_algebraic (dict of processed algebraic equations) and processed_concatenated_algebraic

Return type
tuple

process_symbol(*symbol*)

Discretise operators in model equations. If a symbol has already been discretised, the stored value is returned.

Parameters
symbol (*pybamm.expression_tree.symbol.Symbol*) – Symbol to discretise

Returns
Discretised symbol

Return type
pybamm.expression_tree.symbol.Symbol

set_internal_boundary_conditions(*model*)

A method to set the internal boundary conditions for the submodel. These are required to properly calculate the gradient. Note: this method modifies the state of self.boundary_conditions.

set_variable_slices(*variables*)

Sets the slicing for variables.

Parameters
variables (iterable of pybamm.Variables) – The variables for which to set slices

4.6.2 Spatial Method

class pybamm.**SpatialMethod**(*options=None*)

A general spatial methods class, with default (trivial) behaviour for some spatial operations. All spatial methods will follow the general form of SpatialMethod in that they contain a method for broadcasting variables onto a mesh, a gradient operator, and a divergence operator.

boundary_integral(*child*, *discretised_child*, *region*)

Implements the boundary integral for a spatial method.

Parameters

- **child** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_child** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **region** (*str*) – The region of the boundary over which to integrate. If region is None (default) the integration is carried out over the entire boundary. If region is *negative tab* or *positive tab* then the integration is only carried out over the appropriate part of the boundary corresponding to the tab.

Returns
Contains the result of acting the discretised boundary integral on the child discretised_symbol

Return type
class: *pybamm.Array*

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

Returns the boundary value or flux using the appropriate expression for the spatial method. To do this, we create a sparse vector 'bv_vector' that extracts either the first (for side="left") or last (for side="right") point from 'discretised_child'.

Parameters

- **symbol** (*pybamm.Symbol*) – The boundary value or flux symbol
- **discretised_child** (*pybamm.StateVector*) – The discretised variable from which to calculate the boundary value
- **bcs** (*dict (optional)*) – The boundary conditions. If these are supplied and "use bcs" is True in the options, then these will be used to improve the accuracy of the extrapolation.

Returns
The variable representing the surface value.

Return type
pybamm.MatrixMultiplication

broadcast(*symbol*, *domains*, *broadcast_type*)

Broadcast symbol to a specified domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to be broadcasted
- **domains** (*dict of strings*) – The domains for broadcasting
- **broadcast_type** (*str*) – The type of broadcast: 'primary to node', 'primary to edges', 'secondary to nodes', 'secondary to edges', 'tertiary to nodes', 'tertiary to edges', 'full to nodes' or 'full to edges'

Returns

broadcasted_symbol – The discretised symbol of the correct size for the spatial method

Return type

class: *pybamm.Symbol*

concatenation(*disc_children*)

Discrete concatenation object.

Parameters

disc_children (*list*) – List of discretised children

Returns

Concatenation of the discretised children

Return type

pybamm.DomainConcatenation

delta_function(*symbol*, *discretised_symbol*)

Implements the delta function on the approriate side for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size

divergence(*symbol*, *discretised_symbol*, *boundary_conditions*)

Implements the divergence for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"left": left bc, "right": right bc}})

Returns

Contains the result of acting the discretised divergence on the child discretised_symbol

Return type

class: *pybamm.Array*

evaluate_at(*symbol*, *discretised_child*, *position*)

Returns the symbol evaluated at a given position in space.

Parameters

- **symbol** (*pybamm.Symbol*) – The boundary value or flux symbol
- **discretised_child** (*pybamm.StateVector*) – The discretised variable from which to calculate the boundary value
- **position** (*pybamm.Scalar*) – The point in one-dimensional space at which to evaluate the symbol.

Returns
The variable representing the value at the given point.

Return type
pybamm.MatrixMultiplication

gradient(*symbol*, *discretised_symbol*, *boundary_conditions*)

Implements the gradient for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"left": left bc, "right": right bc}})

Returns
Contains the result of acting the discretised gradient on the child discretised_symbol

Return type
class: *pybamm.Array*

gradient_squared(*symbol*, *discretised_symbol*, *boundary_conditions*)

Implements the inner product of the gradient with itself for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"left": left bc, "right": right bc}})

Returns
Contains the result of taking the inner product of the result of acting the discretised gradient on the child discretised_symbol with itself

Return type
class: *pybamm.Array*

indefinite_integral(*child*, *discretised_child*, *direction*)

Implements the indefinite integral for a spatial method.

Parameters

- **child** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_child** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **direction** (*str*) – The direction of integration

Returns
Contains the result of acting the discretised indefinite integral on the child discretised_symbol

Return type
class: *pybamm.Array*

integral(*child*, *discretised_child*, *integration_dimension*)

Implements the integral for a spatial method.

Parameters

- **child** (*pybamm.Symbol*) – The symbol to which is being integrated
- **discretised_child** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **integration_dimension** (*str, optional*) – The dimension in which to integrate (default is "primary")

Returns
Contains the result of acting the discretised integral on the child discretised_symbol

Return type
class: *pybamm.Array*

internal_neumann_condition(*left_symbol_disc*, *right_symbol_disc*, *left_mesh*, *right_mesh*)

A method to find the internal Neumann conditions between two symbols on adjacent subdomains.

Parameters

- **left_symbol_disc** (*pybamm.Symbol*) – The discretised symbol on the left subdomain
- **right_symbol_disc** (*pybamm.Symbol*) – The discretised symbol on the right subdomain
- **left_mesh** (*list*) – The mesh on the left subdomain
- **right_mesh** (*list*) – The mesh on the right subdomain

laplacian(*symbol*, *discretised_symbol*, *boundary_conditions*)

Implements the Laplacian for a spatial method.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the gradient of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"left": left bc, "right": right bc}})

Returns
Contains the result of acting the discretised Laplacian on the child discretised_symbol

Return type
class: *pybamm.Array*

mass_matrix(*symbol*, *boundary_conditions*)

Calculates the mass matrix for a spatial method.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"left": left bc, "right": right bc}})

Returns
The (sparse) mass matrix for the spatial method.

Return type
pybamm.Matrix

process_binary_operators(*bin_op*, *left*, *right*, *disc_left*, *disc_right*)

Discretise binary operators in model equations. Default behaviour is to return a new binary operator with the discretised children.

Parameters

- **bin_op** (*pybamm.BinaryOperator*) – Binary operator to discretise
- **left** (*pybamm.Symbol*) – The left child of *bin_op*
- **right** (*pybamm.Symbol*) – The right child of *bin_op*
- **disc_left** (*pybamm.Symbol*) – The discretised left child of *bin_op*
- **disc_right** (*pybamm.Symbol*) – The discretised right child of *bin_op*

Returns
Discretised binary operator

Return type
pybamm.BinaryOperator

spatial_variable(*symbol*)

Convert a *pybamm.SpatialVariable* node to a linear algebra object that can be evaluated (here, a *pybamm.Vector* on either the nodes or the edges).

Parameters
symbol (*pybamm.SpatialVariable*) – The spatial variable to be discretised.

Returns
Contains the discretised spatial variable

Return type
pybamm.Vector

4.6.3 Finite Volume

class pybamm.**FiniteVolume**(*options=None*)

A class which implements the steps specific to the finite volume method during discretisation.

For broadcast and mass_matrix, we follow the default behaviour from SpatialMethod.

Parameters
options (*dict-like, optional*) – A dictionary of options to be passed to the spatial method. The only option currently available is "extrapolation", which has options for "order" and "use_bcs". It sets the order separately for *pybamm.BoundaryValue* and *pybamm.BoundaryGradient*. Default is "linear" for the value and quadratic for the gradient.

Extends: *pybamm.spatial_methods.spatial_method.SpatialMethod*

add_ghost_nodes(*symbol*, *discretised_symbol*, *bcs*)

Add ghost nodes to a symbol.

For Dirichlet bcs, for a boundary condition "y = a at the left-hand boundary", we concatenate a ghost node to the start of the vector y with value "2*a - y1" where y1 is the value of the first node. Similarly for the right-hand boundary condition.

For Neumann bcs no ghost nodes are added. Instead, the exact value provided by the boundary condition is used at the cell edge when calculating the gradient (see *pybamm.FiniteVolume.add_neumann_values()*).

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_symbol** (*pybamm.Vector*) – Contains the discretised variable
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")

Returns

Matrix @ discretised_symbol + bcs_vector. When evaluated, this gives the discretised_symbol, with appropriate ghost nodes concatenated at each end.

Return type

pybamm.Symbol

add_neumann_values(*symbol*, *discretised_gradient*, *bcs*, *domain*)

Add the known values of the gradient from Neumann boundary conditions to the discretised gradient.

Dirichlet bcs are implemented using ghost nodes, see *pybamm.FiniteVolume.add_ghost_nodes()*.

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_gradient** (*pybamm.Vector*) – Contains the discretised gradient of symbol
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")
- **domain** (*list of strings*) – The domain of the gradient of the symbol (may include ghost nodes)

Returns

Matrix @ discretised_gradient + bcs_vector. When evaluated, this gives the discretised_gradient, with the values of the Neumann boundary conditions concatenated at each end (if given).

Return type

pybamm.Symbol

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

Uses extrapolation to get the boundary value or flux of a variable in the Finite Volume Method.

See `pybamm.SpatialMethod.boundary_value()`

concatenation(*disc_children*)

Discrete concatenation, taking *edge_to_node* for children that evaluate on edges. See *pybamm.SpatialMethod.concatenation()*

definite_integral_matrix(*child*, *vector_type='row'*, *integration_dimension='primary'*)

Matrix for finite-volume implementation of the definite integral in the primary dimension

$$I = \int_a^b f(s)\,ds$$

for where a and b are the left-hand and right-hand boundaries of the domain respectively

Parameters

- **child** (*pybamm.Symbol*) – The symbol being integrated
- **vector_type** (*str, optional*) – Whether to return a row or column vector in the primary dimension (default is row)
- **integration_dimension** (*str, optional*) – The dimension in which to integrate (default is "primary")

Returns

The finite volume integral matrix for the domain

Return type

pybamm.Matrix

delta_function(*symbol*, *discretised_symbol*)

Delta function. Implemented as a vector whose only non-zero element is the first (if symbol.side = "left") or last (if symbol.side = "right"), with appropriate value so that the integral of the delta function across the whole domain is the same as the integral of the discretised symbol across the whole domain.

See *pybamm.SpatialMethod.delta_function()*

divergence(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the divergence operator. See *pybamm.SpatialMethod.divergence()*

divergence_matrix(*domains*)

Divergence matrix for finite volumes in the appropriate domain. Equivalent to div(N) = (N[1:] - N[:-1])/dx

Parameters

domains (*dict*) – The domain(s) and auxiliary domain in which to compute the divergence matrix

Returns

The (sparse) finite volume divergence matrix for the domain

Return type

pybamm.Matrix

edge_to_node(*discretised_symbol*, *method='arithmetic'*)

Convert a discretised symbol evaluated on the cell edges to a discretised symbol evaluated on the cell nodes. See *pybamm.FiniteVolume.shift()*

evaluate_at(*symbol*, *discretised_child*, *position*)

Returns the symbol evaluated at a given position in space.

Parameters

- **symbol** (*pybamm.Symbol*) – The boundary value or flux symbol
- **discretised_child** (*pybamm.StateVector*) – The discretised variable from which to calculate the boundary value

- **position** (*pybamm.Scalar*) – The point in one-dimensional space at which to evaluate the symbol.

Returns
The variable representing the value at the given point.

Return type
pybamm.MatrixMultiplication

gradient(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the gradient operator. See *pybamm.SpatialMethod.gradient()*

gradient_matrix(*domain*, *domains*)

Gradient matrix for finite volumes in the appropriate domain. Equivalent to grad(y) = (y[1:] - y[:-1])/dx

Parameters
domains (*list*) – The domain in which to compute the gradient matrix, including ghost nodes

Returns
The (sparse) finite volume gradient matrix for the domain

Return type
pybamm.Matrix

indefinite_integral(*child*, *discretised_child*, *direction*)

Implementation of the indefinite integral operator.

indefinite_integral_matrix_edges(*domains*, *direction*)

Matrix for finite-volume implementation of the indefinite integral where the integrand is evaluated on mesh edges (shape (n+1, 1)). The integral will then be evaluated on mesh nodes (shape (n, 1)).

Parameters

- **domains** (*dict*) – The domain(s) and auxiliary domains of integration
- **direction** (*str*) – The direction of integration (forward or backward). See notes.

Returns
The finite volume integral matrix for the domain

Return type
pybamm.Matrix

Notes

Forward integral

$$F(x) = \int_0^x f(u)\,du$$

The indefinite integral must satisfy the following conditions:

- $F(0) = 0$
- $f(x) = \frac{dF}{dx}$

or, in discrete form,

- *BoundaryValue(F, "left") = 0*, i.e. $3 * F_0 - F_1 = 0$
- $f_{i+1/2} = (F_{i+1} - F_i)/dx_{i+1/2}$

Hence we must have

- $F_0 = du_{1/2} * f_{1/2}/2$
- $F_{i+1} = F_i + du_{i+1/2} * f_{i+1/2}$

Note that $f_{-1/2}$ and $f_{end+1/2}$ are included in the discrete integrand vector *f*, so we add a column of zeros at each end of the indefinite integral matrix to ignore these.

Backward integral

$$F(x) = \int_x^{end} f(u)\, du$$

The indefinite integral must satisfy the following conditions:

- $F(end) = 0$
- $f(x) = -\frac{dF}{dx}$

or, in discrete form,

- *BoundaryValue(F, "right") = 0*, i.e. $3 * F_{end} - F_{end-1} = 0$
- $f_{i+1/2} = -(F_{i+1} - F_i)/dx_{i+1/2}$

Hence we must have

- $F_{end} = du_{end+1/2} * f_{end-1/2}/2$
- $F_{i-1} = F_i + du_{i-1/2} * f_{i-1/2}$

Note that $f_{-1/2}$ and $f_{end+1/2}$ are included in the discrete integrand vector *f*, so we add a column of zeros at each end of the indefinite integral matrix to ignore these.

indefinite_integral_matrix_nodes(*domains*, *direction*)

Matrix for finite-volume implementation of the (backward) indefinite integral where the integrand is evaluated on mesh nodes (shape (n, 1)). The integral will then be evaluated on mesh edges (shape (n+1, 1)). This is just a straightforward (backward) cumulative sum of the integrand

Parameters

- **domains** (*dict*) – The domain(s) and auxiliary domains of integration
- **direction** (*str*) – The direction of integration (forward or backward)

Returns

The finite volume integral matrix for the domain

Return type

pybamm.Matrix

integral(*child*, *discretised_child*, *integration_dimension*)

Vector-vector dot product to implement the integral operator.

internal_neumann_condition(*left_symbol_disc*, *right_symbol_disc*, *left_mesh*, *right_mesh*)

A method to find the internal Neumann conditions between two symbols on adjacent subdomains.

Parameters

- **left_symbol_disc** (*pybamm.Symbol*) – The discretised symbol on the left subdomain
- **right_symbol_disc** (*pybamm.Symbol*) – The discretised symbol on the right subdomain

- **left_mesh** (*list*) – The mesh on the left subdomain
- **right_mesh** (*list*) – The mesh on the right subdomain

laplacian(*symbol*, *discretised_symbol*, *boundary_conditions*)

Laplacian operator, implemented as div(grad(.)) See *pybamm.SpatialMethod.laplacian()*

node_to_edge(*discretised_symbol*, *method='arithmetic'*)

Convert a discretised symbol evaluated on the cell nodes to a discretised symbol evaluated on the cell edges. See *pybamm.FiniteVolume.shift()*

process_binary_operators(*bin_op*, *left*, *right*, *disc_left*, *disc_right*)

Discretise binary operators in model equations. Performs appropriate averaging of diffusivities if one of the children is a gradient operator, so that discretised sizes match up. For this averaging we use the harmonic mean [1].

[1] Recktenwald, Gerald. "The control-volume finite-difference approximation to the diffusion equation." (2012).

Parameters

- **bin_op** (*pybamm.BinaryOperator*) – Binary operator to discretise
- **left** (*pybamm.Symbol*) – The left child of *bin_op*
- **right** (*pybamm.Symbol*) – The right child of *bin_op*
- **disc_left** (*pybamm.Symbol*) – The discretised left child of *bin_op*
- **disc_right** (*pybamm.Symbol*) – The discretised right child of *bin_op*

Returns

Discretised binary operator

Return type

pybamm.BinaryOperator

shift(*discretised_symbol*, *shift_key*, *method*)

Convert a discretised symbol evaluated at edges/nodes, to a discretised symbol evaluated at nodes/edges. Can be the arithmetic mean or the harmonic mean.

Note: when computing fluxes at cell edges it is better to take the harmonic mean based on [1].

[1] Recktenwald, Gerald. "The control-volume finite-difference approximation to the diffusion equation." (2012).

Parameters

- **discretised_symbol** (*pybamm.Symbol*) – Symbol to be averaged. When evaluated, this symbol returns either a scalar or an array of shape (n,) or (n+1,), where n is the number of points in the mesh for the symbol's domain (n = self.mesh[symbol.domain].npts)
- **shift_key** (*str*) – Whether to shift from nodes to edges ("node to edge"), or from edges to nodes ("edge to node")
- **method** (*str*) – Whether to use the "arithmetic" or "harmonic" mean

Returns

Averaged symbol. When evaluated, this returns either a scalar or an array of shape (n+1,) (if *shift_key* = *"node to edge"*) or (n,) (if *shift_key* = *"edge to node"*)

Return type

pybamm.Symbol

spatial_variable(*symbol*)

Creates a discretised spatial variable compatible with the FiniteVolume method.

Parameters
symbol (*pybamm.SpatialVariable*) – The spatial variable to be discretised.

Returns
Contains the discretised spatial variable

Return type
pybamm.Vector

upwind_or_downwind(*symbol*, *discretised_symbol*, *bcs*, *direction*)

Implement an upwinding operator. Currently, this requires the symbol to have a Dirichlet boundary condition on the left side (for upwinding) or right side (for downwinding).

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_gradient** (*pybamm.Vector*) – Contains the discretised gradient of symbol
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")
- **direction** (*str*) – Direction in which to apply the operator (upwind or downwind)

4.6.4 Spectral Volume

class pybamm.**SpectralVolume**(*options=None*, *order=2*)

A class which implements the steps specific to the Spectral Volume discretisation. It is implemented in such a way that it is very similar to FiniteVolume; that comes at the cost that it is only compatible with the SpectralVolume1DSubMesh (which is a certain subdivision of any 1D mesh, so it shouldn't be a problem).

For broadcast and mass_matrix, we follow the default behaviour from SpatialMethod. For spatial_variable, divergence, divergence_matrix, laplacian, integral, definite_integral_matrix, indefinite_integral, indefinite_integral_matrix, indefinite_integral_matrix_nodes, indefinite_integral_matrix_edges, delta_function we follow the behaviour from FiniteVolume. This is possible since the node values are integral averages with Spectral Volume, just as with Finite Volume. delta_function assigns the integral value to a CV instead of a SV this way, but that doesn't matter too much. Additional methods that are inherited by FiniteVolume which technically are not suitable for Spectral Volume are boundary_value_or_flux, process_binary_operators, concatenation, node_to_edge, edge_to_node and shift. While node_to_edge (as well as boundary_value_or_flux and process_binary_operators) could utilize the reconstruction approach of Spectral Volume, the inverse edge_to_node would still have to fall back to the Finite Volume behaviour. So these are simply inherited for consistency. boundary_value_or_flux might not benefit from the reconstruction approach at all, as it seems to only preprocess symbols.

Parameters
mesh (*pybamm.Mesh*) – Contains all the submeshes for discretisation

Extends: *pybamm.spatial_methods.finite_volume.FiniteVolume*

chebyshev_collocation_points(*noe*, *a=-1.0*, *b=1.0*)

Calculates Chebyshev collocation points in descending order.

Parameters

- **noe** (*integer*) – The number of the collocation points. "number of edges"

- **a** (*float*) – Left end of the interval on which the Chebyshev collocation points are constructed. Default is -1.
- **b** (*float*) – Right end of the interval on which the Chebyshev collocation points are constructed. Default is 1.

Returns

- numpy.array
- *Chebyshev collocation points on [a,b].*

chebyshev_differentiation_matrices(*noe*, *dod*)

Chebyshev differentiation matrices, from Baltensperger and Trummer[1].

Parameters

- **noe** (*integer*) – The number of the collocation points. "number of edges"
- **dod** (*integer*) – The maximum order of differentiation for which a differentiation matrix shall be calculated. Note that it has to be smaller than 'noe'. "degrees of differentiation"

Returns

The differentiation matrices in ascending order of differentiation order. With exact arithmetic, the diff. matrix of order p would just be the pth matrix power of the diff. matrix of order 1. This method computes the higher orders in a more numerically stable way.

Return type

list(numpy.array)

cv_boundary_reconstruction_matrix(*domains*)

"Broadcasts" the basic edge value reconstruction matrix to the actual shape of the discretised symbols. Note that the product of this and a discretised symbol is a vector which represents duplicate values for all inner SV edges. These are the reconstructed values from both sides.

Parameters

domains (*dict*) – The domains in which to compute the gradient matrix

Returns

The (sparse) CV reconstruction matrix for the domain

Return type

pybamm.Matrix

cv_boundary_reconstruction_sub_matrix()

Coefficients for reconstruction of a function through averages. The resulting matrix is scale-invariant Wang[2].

gradient(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the gradient operator. See *pybamm.SpatialMethod.gradient()*

gradient_matrix(*domain*, *domains*)

Gradient matrix for Spectral Volume in the appropriate domain. Note that it contains the averaging of the duplicate SV edge gradient values, such that the product of it and a reconstructed discretised symbol simply represents CV edge values. On its own, it only works on non-concatenated domains, since only then the

[1] Richard Baltensperger and Manfred R Trummer. Spectral differencing with a twist. *SIAM journal on scientific computing*, 24(5):1465–1487, 2003.

[2] Z. J. Wang. Spectral (finite) volume method for conservation laws on unstructured grids. *Journal of Computational Physics*, 178(1):210–251, 2002. doi:10.1006/jcph.2002.7041.

boundary conditions ensure correct behaviour. More generally, it only works if gradients are a result of boundary conditions rather than continuity conditions. For example, two adjacent SVs with gradient zero in each of them but with different variable values will have zero gradient between them. This is fixed with "penalty_matrix".

Parameters
domains (*dict*) – The domains in which to compute the gradient matrix

Returns
The (sparse) Spectral Volume gradient matrix for the domain

Return type
pybamm.Matrix

penalty_matrix(*domains*)

Penalty matrix for Spectral Volume in the appropriate domain. This works the same as the "gradient_matrix" of FiniteVolume does, just between SVs and not between CVs. Think of it as a continuity penalty.

Parameters
domains (*dict*) – The domains in which to compute the gradient matrix

Returns
The (sparse) Spectral Volume penalty matrix for the domain

Return type
pybamm.Matrix

replace_dirichlet_values(*symbol*, *discretised_symbol*, *bcs*)

Replace the reconstructed value at Dirichlet boundaries with the boundary condition.

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_symbol** (*pybamm.Vector*) – Contains the discretised variable
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")

Returns
Matrix @ discretised_symbol + bcs_vector. When evaluated, this gives the discretised_symbol, with its boundary values replaced by the Dirichlet boundary conditions.

Return type
pybamm.Symbol

replace_neumann_values(*symbol*, *discretised_gradient*, *bcs*)

Replace the known values of the gradient from Neumann boundary conditions into the discretised gradient.

Parameters

- **symbol** (*pybamm.SpatialVariable*) – The variable to be discretised
- **discretised_gradient** (*pybamm.Vector*) – Contains the discretised gradient of symbol
- **bcs** (dict of tuples (*pybamm.Scalar*, str)) – Dictionary (with keys "left" and "right") of boundary conditions. Each boundary condition consists of a value and a flag indicating its type (e.g. "Dirichlet")

Returns
Matrix @ discretised_gradient + bcs_vector. When evaluated, this gives the discretised_gradient, with its boundary values replaced by the Neumann boundary conditions.

Return type
pybamm.Symbol

References

4.6.5 Scikit Finite Elements

class pybamm.**ScikitFiniteElement**(*options=None*)

A class which implements the steps specific to the finite element method during discretisation. The class uses scikit-fem to discretise the problem to obtain the mass and stiffness matrices. At present, this class is only used for solving the Poisson problem -grad^2 u = f in the y-z plane (i.e. not the through-cell direction).

For broadcast, we follow the default behaviour from SpatialMethod.

Extends: *pybamm.spatial_methods.spatial_method.SpatialMethod*

assemble_mass_form(*symbol*, *boundary_conditions*, *region='interior'*)

Assembles the form of the finite element mass matrix over the domain interior or boundary.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})
- **region** (*str, optional*) – The domain over which to assemble the mass matrix form. Can be "interior" (default) or "boundary".

Returns
The (sparse) mass matrix for the spatial method.

Return type
pybamm.Matrix

bc_apply(*M*, *boundary*, *zero=False*)

Adjusts the assembled finite element matrices to account for boundary conditions.

Parameters

- **M** (*scipy.sparse.coo_matrix*) – The assembled finite element matrix to adjust.
- **boundary** (*numpy.array*) – Array of the indices which correspond to the boundary.
- **zero** (*bool, optional*) – If True, the rows of M given by the indicies in boundary are set to zero. If False, the diagonal element is set to one. default is False.

boundary_integral(*child*, *discretised_child*, *region*)

Implementation of the boundary integral operator. See *pybamm.SpatialMethod.boundary_integral()*

boundary_integral_vector(*domain*, *region*)

A node in the expression tree representing an integral operator over the boundary of a domain

$$I = \int_{\partial a} f(u)\, du,$$

where ∂a is the boundary of the domain, and $u \in$ domain boundary.

Parameters

- **domain** (*list*) – The domain(s) of the variable in the integrand
- **region** (*str*) – The region of the boundary over which to integrate. If region is *entire* the integration is carried out over the entire boundary. If region is *negative tab* or *positive tab* then the integration is only carried out over the appropriate part of the boundary corresponding to the tab.

Returns
The finite element integral vector for the domain

Return type
pybamm.Matrix

boundary_mass_matrix(*symbol*, *boundary_conditions*)

Calculates the mass matrix for the finite element method assembled over the boundary.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns
The (sparse) mass matrix for the spatial method.

Return type
pybamm.Matrix

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

Returns the average value of the symbol over the negative tab ("negative tab") or the positive tab ("positive tab") in the Finite Element Method.

Overwrites the default `pybamm.SpatialMethod.boundary_value()`

definite_integral_matrix(*child*, *vector_type='row'*)

Matrix for finite-element implementation of the definite integral over the entire domain

$$I = \int_\Omega f(s)\,dx$$

for where Ω is the domain.

Parameters

- **child** (*pybamm.Symbol*) – The symbol being integrated
- **vector_type** (*str, optional*) – Whether to return a row or column vector (default is row)

Returns
The finite element integral vector for the domain

Return type
pybamm.Matrix

divergence(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the divergence operator. See *pybamm.SpatialMethod.divergence()*

gradient(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the gradient operator. The gradient w of the function u is approximated by the finite element method using the same function space as u, i.e. we solve w = grad(u), which corresponds to the weak form w*v*dx = grad(u)*v*dx, where v is a suitable test function.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the Laplacian of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns

A concatenation that contains the result of acting the discretised gradient on the child discretised_symbol. The first column corresponds to the y-component of the gradient and the second column corresponds to the z component of the gradient.

Return type

class: *pybamm.Concatenation*

gradient_matrix(*symbol*, *boundary_conditions*)

Gradient matrix for finite elements in the appropriate domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol for which we want to calculate the gradient matrix
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns

The (sparse) finite element gradient matrix for the domain

Return type

pybamm.Matrix

gradient_squared(*symbol*, *discretised_symbol*, *boundary_conditions*)

Multiplication to implement the inner product of the gradient operator with itself. See *pybamm.SpatialMethod.gradient_squared()*

indefinite_integral(*child*, *discretised_child*, *direction*)

Implementation of the indefinite integral operator. The input discretised child must be defined on the internal mesh edges. See *pybamm.SpatialMethod.indefinite_integral()*

integral(*child*, *discretised_child*, *integration_dimension*)

Vector-vector dot product to implement the integral operator. See *pybamm.SpatialMethod.integral()*

laplacian(*symbol*, *discretised_symbol*, *boundary_conditions*)

Matrix-vector multiplication to implement the Laplacian operator.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol that we will take the Laplacian of.
- **discretised_symbol** (*pybamm.Symbol*) – The discretised symbol of the correct size
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns
Contains the result of acting the discretised gradient on the child discretised_symbol

Return type
class: *pybamm.Array*

mass_matrix(*symbol*, *boundary_conditions*)

Calculates the mass matrix for the finite element method.

Parameters

- **symbol** (*pybamm.Variable*) – The variable corresponding to the equation for which we are calculating the mass matrix.
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns
The (sparse) mass matrix for the spatial method.

Return type
pybamm.Matrix

spatial_variable(*symbol*)

Creates a discretised spatial variable compatible with the FiniteElement method.

Parameters
symbol (*pybamm.SpatialVariable*) – The spatial variable to be discretised.

Returns
Contains the discretised spatial variable

Return type
pybamm.Vector

stiffness_matrix(*symbol*, *boundary_conditions*)

Laplacian (stiffness) matrix for finite elements in the appropriate domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol for which we want to calculate the Laplacian matrix
- **boundary_conditions** (*dict*) – The boundary conditions of the model ({symbol: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns
The (sparse) finite element stiffness matrix for the domain

Return type
pybamm.Matrix

4.6.6 Zero Dimensional Spatial Method

class pybamm.**ZeroDimensionalSpatialMethod**(*options=None*)

A discretisation class for the zero dimensional mesh

Parameters
mesh – Contains all the submeshes for discretisation

Extends: *pybamm.spatial_methods.spatial_method.SpatialMethod*

boundary_value_or_flux(*symbol*, *discretised_child*, *bcs=None*)

In 0D, the boundary value is the identity operator. See *SpatialMethod.boundary_value_or_flux()*

indefinite_integral(*child*, *discretised_child*, *direction*)

Calculates the zero-dimensional indefinite integral. If 'direction' is forward, this is the identity operator. If 'direction' is backward, this is the negation operator.

integral(*child*, *discretised_child*, *integration_dimension*)

Calculates the zero-dimensional integral, i.e. the identity operator

mass_matrix(*symbol*, *boundary_conditions*)

Calculates the mass matrix for a spatial method. Since the spatial method is zero dimensional, this is simply the number 1.

4.7 Solvers

4.7.1 Base Solver

class pybamm.**BaseSolver**(*method=None*, *rtol=1e-06*, *atol=1e-06*, *root_method=None*, *root_tol=1e-06*, *extrap_tol=None*, *output_variables=None*)

Solve a discretised model.

Parameters

- **method** (*str, optional*) – The method to use for integration, specific to each solver
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str or pybamm algebraic solver class, optional*) – The method to use to find initial conditions (for DAE solvers). If a solver class, must be an algebraic solver class. If "casadi", the solver uses casadi's Newton rootfinding algorithm to find initial conditions. Otherwise, the solver uses 'scipy.optimize.root' with method specified by 'root_method' (e.g. "lm", "hybr", ...)
- **root_tol** (*float, optional*) – The tolerance for the initial-condition solver (default is 1e-6).
- **extrap_tol** (*float, optional*) – The tolerance to assert whether extrapolation occurs or not. Default is 0.
- **output_variables** (*list[str], optional*) – List of variables to calculate and return. If none are specified then the complete state vector is returned (can be very large) (default is [])

calculate_consistent_state(*model*, *time=0*, *inputs=None*)

Calculate consistent state for the algebraic equations through root-finding. model.y0 is used as the initial guess for rootfinding

Parameters

- **model** (*pybamm.BaseModel*) – The model for which to calculate initial conditions.
- **time** (*float*) – The time at which to calculate the initial conditions
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving

Returns

y0_consistent – Initial conditions that are consistent with the algebraic equations (roots of the algebraic equations). If self.root_method == None then returns model.y0.

Return type
array-like, same shape as y0_guess

check_extrapolation(*solution*, *events*)

Check if extrapolation occurred for any of the interpolants. Note that with the current approach (evaluating all the events at the solution times) some extrapolations might not be found if they only occurred for a small period of time.

Parameters

- **solution** (*pybamm.Solution*) – The solution object
- **events** (*dict*) – Dictionary of events

copy()

Returns a copy of the solver

static get_termination_reason(*solution*, *events*)

Identify the cause for termination. In particular, if the solver terminated due to an event, (try to) pinpoint which event was responsible. If an event occurs the event time and state are added to the solution object. Note that the current approach (evaluating all the events and then finding which one is smallest at the final timestep) is pretty crude, but is the easiest one that works for all the different solvers.

Parameters

- **solution** (*pybamm.Solution*) – The solution object
- **events** (*dict*) – Dictionary of events

set_up(*model*, *inputs=None*, *t_eval=None*, *ics_only=False*)

Unpack model, perform checks, and calculate jacobian.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **t_eval** (*numeric type, optional*) – The times at which to stop the integration due to a discontinuity in time.

solve(*model*, *t_eval=None*, *inputs=None*, *nproc=None*, *calculate_sensitivities=False*, *t_interp=None*)

Execute the solver setup and calculate the solution of the model at specified times.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions. All calls to solve must pass in the same model or an error is raised
- **t_eval** (*None, list or ndarray, optional*) – The times (in seconds) at which to compute the solution. Defaults to None.
- **inputs** (*dict or list, optional*) – A dictionary or list of dictionaries describing any input parameters to pass to the model when solving
- **nproc** (*int, optional*) – Number of processes to use when solving for more than one set of input parameters. Defaults to value returned by "os.cpu_count()".
- **calculate_sensitivities** (*list of str or bool, optional*) – Whether the solver calculates sensitivities of all input parameters. Defaults to False. If only a subset of sensitivities are required, can also pass a list of input parameter names.

Limitations: sensitivities are not calculated up to numerical tolerances so are not guarenteed to be within the tolerances set by the solver, please raise an issue if you require this functionality. Also, when using this feature with *pybamm.Experiment*, the sensitivities do not take into account the movement of step-transitions wrt input parameters, so do not use this feature if the timings of your experimental protocol change rapidly with respect to your input parameters.

- **t_interp** (*None, list or ndarray, optional*) – The times (in seconds) at which to interpolate the solution. Defaults to None. Only valid for solvers that support intra-solve interpolation (*IDAKLUSolver*).

Returns

If type of *inputs* is *list*, return a list of corresponding *pybamm.Solution* objects.

Return type

pybamm.Solution or list of *pybamm.Solution* objects.

Raises

- **pybamm.ModelError** – If an empty model is passed (*model.rhs = {}* and *model.algebraic={}* and *model.variables = {}*)
- **RuntimeError** – If multiple calls to *solve* pass in different models

step(*old_solution*, *model*, *dt*, *t_eval=None*, *npts=None*, *inputs=None*, *save=True*, *calculate_sensitivities=False*, *t_interp=None*)

Step the solution of the model forward by a given time increment. The first time this method is called it executes the necessary setup by calling *self.set_up(model)*.

Parameters

- **old_solution** (*pybamm.Solution* or None) – The previous solution to be added to. If *None*, a new solution is created.
- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **dt** (*numeric type*) – The timestep (in seconds) over which to step the solution
- **t_eval** (*list or numpy.ndarray, optional*) – An array of times at which to stop the simulation and return the solution during the step (Note: t_eval is the time measured from the start of the step, so should start at 0 and end at dt). By default, the solution is returned at t0 and t0 + dt.
- **npts** (*deprecated*)
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **save** (*bool, optional*) – Save solution with all previous timesteps. Defaults to True.
- **calculate_sensitivities** (*list of str or bool, optional*) – Whether the solver calculates sensitivities of all input parameters. Defaults to False. If only a subset of sensitivities are required, can also pass a list of input parameter names. **Limitations**: sensitivities are not calculated up to numerical tolerances so are not guarenteed to be within the tolerances set by the solver, please raise an issue if you require this functionality. Also, when using this feature with *pybamm.Experiment*, the sensitivities do not take into account the movement of step-transitions wrt input parameters, so do not use this feature if the timings of your experimental protocol change rapidly with respect to your input parameters.

- **t_interp** (*None, list or ndarray, optional*) – The times (in seconds) at which to interpolate the solution. Defaults to None. Only valid for solvers that support intra-solve interpolation (*IDAKLUSolver*).

Raises

pybamm.ModelError – If an empty model is passed (*model.rhs = {}* and *model.algebraic = {}* and *model.variables = {}*)

4.7.2 Dummy Solver

class pybamm.**DummySolver**

Dummy solver class for empty models.

Extends: *pybamm.solvers.base_solver.BaseSolver*

4.7.3 Scipy Solver

class pybamm.**ScipySolver**(*method='BDF'*, *rtol=1e-06*, *atol=1e-06*, *extrap_tol=None*, *extra_options=None*)

Solve a discretised model, using scipy.integrate.solve_ivp.

Parameters

- **method** (*str, optional*) – The method to use in solve_ivp (default is "BDF")
- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **extrap_tol** (*float, optional*) – The tolerance to assert whether extrapolation occurs or not (default is 0).
- **extra_options** (*dict, optional*) – Any options to pass to the solver. Please consult SciPy documentation for details.

Extends: *pybamm.solvers.base_solver.BaseSolver*

4.7.4 JAX Solver

class pybamm.**JaxSolver**(*method='BDF'*, *root_method=None*, *rtol=1e-06*, *atol=1e-06*, *extrap_tol=None*, *extra_options=None*)

Solve a discretised model using a JAX compiled solver.

Note: this solver will not work with models that have
termination events or are not converted to jax format

Raises

- RuntimeError – if model has any termination events
- RuntimeError – if *model.convert_to_format != 'jax'*

Parameters

- **method** (str, optional (see *jax.experimental.ode.odeint* for details)) –
  - 'BDF' (default) uses custom jax_bdf_integrate (see *jax_bdf_integrate.py* for details)
  - 'RK45' uses jax.experimental.ode.odeint
- **root_method** (*str, optional*) – Method to use to calculate consistent initial conditions. By default, this uses the newton chord method internal to the jax bdf solver, otherwise choose from the set of default options defined in docs for pybamm.BaseSolver

- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **extrap_tol** (*float, optional*) – The tolerance to assert whether extrapolation occurs or not (default is 0).
- **extra_options** (*dict, optional*) – Any options to pass to the solver. Please consult JAX documentation for details.

Extends: *pybamm.solvers.base_solver.BaseSolver*

create_solve(*model*, *t_eval*)

Return a compiled JAX function that solves an ode model with input arguments.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate.
- **t_eval** (numpy.array, size (k,)) – The times at which to compute the solution

Returns

A function with signature *f(inputs)*, where inputs are a dict containing any input parameters to pass to the model when solving

Return type

function

get_solve(*model*, *t_eval*)

Return a compiled JAX function that solves an ode model with input arguments.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate.
- **t_eval** (numpy.array, size (k,)) – The times at which to compute the solution

Returns

A function with signature *f(inputs)*, where inputs are a dict containing any input parameters to pass to the model when solving

Return type

function

pybamm.**jax_bdf_integrate**(*func*, *y0*, *t_eval*, **args*, *rtol=1e-06*, *atol=1e-06*, *mass=None*)

Backward Difference formula (BDF) implicit multistep integrator. The basic algorithm is derived in Byrne and Hindmarsh[1]. This particular implementation follows that implemented in the Matlab routine ode15s described in Shampine and Reichelt[2] and the SciPy implementation Virtanen *et al.*[3] which features the NDF formulas for improved stability, with associated differences in the error constants, and calculates the jacobian at J(t_{n+1}, y^0_{n+1}). This implementation was based on that implemented in the SciPy library Virtanen *et al.*[3], which also mainly follows Shampine and Reichelt[2] but uses the more standard jacobian update.

Parameters

- **func** (*callable*) – function to evaluate the time derivative of the solution *y* at time *t* as *func(y, t, *args)*, producing the same shape/structure as *y0*.

[1] George D. Byrne and Alan C. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 1(1):71–96, 1975.

[2] Lawrence F Shampine and Mark W Reichelt. The matlab ode suite. *SIAM journal on scientific computing*, 18(1):1–22, 1997.

[3] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, and others. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020. doi:10.1038/s41592-019-0686-2.

- **y0** (*ndarray*) – initial state vector
- **t_eval** (*ndarray*) – time points to evaluate the solution, has shape (m,)
- **args** (*(optional)*) – tuple of additional arguments for *fun*, which must be arrays scalars, or (nested) standard Python containers (tuples, lists, dicts, namedtuples, i.e. pytrees) of those types.
- **rtol** (*(optional) float*) – relative tolerance for the solver
- **atol** (*(optional) float*) – absolute tolerance for the solver
- **mass** (*(optional) ndarray*) – diagonal of the mass matrix with shape (n,)

Returns
y – calculated state vector at each of the m time points

Return type
ndarray with shape (n, m)

References

4.7.5 IDAKLU Solver

class pybamm.**IDAKLUSolver**(*rtol=0.0001*, *atol=1e-06*, *root_method='casadi'*, *root_tol=1e-06*, *extrap_tol=None*, *output_variables=None*, *options=None*)

Solve a discretised model, using sundials with the KLU sparse linear solver.

Parameters

- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-4).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str or pybamm algebraic solver class, optional*) – The method to use to find initial conditions (for DAE solvers). If a solver class, must be an algebraic solver class. If "casadi", the solver uses casadi's Newton rootfinding algorithm to find initial conditions. Otherwise, the solver uses 'scipy.optimize.root' with method specified by 'root_method' (e.g. "lm", "hybr", ...)
- **root_tol** (*float, optional*) – The tolerance for the initial-condition solver (default is 1e-6).
- **extrap_tol** (*float, optional*) – The tolerance to assert whether extrapolation occurs or not (default is 0).
- **output_variables** (*list[str], optional*) – List of variables to calculate and return. If none are specified then the complete state vector is returned (can be very large) (default is [])
- **options** (*dict, optional*) – Addititional options to pass to the solver, by default:

```
options = {
    # Print statistics of the solver after every solve
    "print_stats": False,
    # Number of threads available for OpenMP (must be greater than␣
↪or equal to `num_solvers`)
    "num_threads": 1,
    # Number of solvers to use in parallel (for solving multiple␣
↪sets of input parameters in parallel)
    "num_solvers": num_threads,
```

(continues on next page)

(continued from previous page)

```
    # Evaluation engine to use for jax, can be 'jax'(native) or 'iree
↪'
    "jax_evaluator": "jax",
    ## Linear solver interface
    # name of sundials linear solver to use options are:
↪"SUNLinSol_KLU",
    # "SUNLinSol_Dense", "SUNLinSol_Band", "SUNLinSol_SPBCGS",
    # "SUNLinSol_SPFGMR", "SUNLinSol_SPGMR", "SUNLinSol_SPTFQMR",
    "linear_solver": "SUNLinSol_KLU",
    # Jacobian form, can be "none", "dense",
    # "banded", "sparse", "matrix-free"
    "jacobian": "sparse",
    # Preconditioner for iterative solvers, can be "none", "BBDP"
    "preconditioner": "BBDP",
    # For iterative linear solver preconditioner, bandwidth of
    # approximate jacobian
    "precon_half_bandwidth": 5,
    # For iterative linear solver preconditioner, bandwidth of
    # approximate jacobian that is kept
    "precon_half_bandwidth_keep": 5,
    # For iterative linear solvers, max number of iterations
    "linsol_max_iterations": 5,
    # Ratio between linear and nonlinear tolerances
    "epsilon_linear_tolerance": 0.05,
    # Increment factor used in DQ Jacobian-vector product␣
↪approximation
    "increment_factor": 1.0,
    # Enable or disable linear solution scaling
    "linear_solution_scaling": True,
    ## Main solver
    # Maximum order of the linear multistep method
    "max_order_bdf": 5,
    # Maximum number of steps to be taken by the solver in its␣
↪attempt to
    # reach the next output time.
    # Note: this value differs from the IDA default of 500
    "max_num_steps": 100000,
    # Initial step size. The solver default is used if this is␣
↪left at 0.0
    "dt_init": 0.0,
    # Minimum absolute step size. The solver default is used if␣
↪this is
    # left at 0.0
    "dt_min": 0.0,
    # Maximum absolute step size. The solver default is used if␣
↪this is
    # left at 0.0
    "dt_max": 0.0,
    # Maximum number of error test failures in attempting one step
    "max_error_test_failures": 10,
    # Maximum number of nonlinear solver iterations at one step
    # Note: this value differs from the IDA default of 4
```

(continues on next page)

(continued from previous page)

```
    "max_nonlinear_iterations": 40,
    # Maximum number of nonlinear solver convergence failures at␣
↪one step
    # Note: this value differs from the IDA default of 10
    "max_convergence_failures": 100,
    # Safety factor in the nonlinear convergence test
    "nonlinear_convergence_coefficient": 0.33,
    # Suppress algebraic variables from error test
    "suppress_algebraic_error": False,
    # Store Hermite interpolation data for the solution.
    # Note: this option is always disabled if output_variables are␣
↪given
    # or if t_interp values are specified
    "hermite_interpolation": True,
    ## Initial conditions calculation
    # Positive constant in the Newton iteration convergence test␣
↪within the
    # initial condition calculation
    "nonlinear_convergence_coefficient_ic": 0.0033,
    # Maximum number of steps allowed when `init_all_y_ic = False`
    # Note: this value differs from the IDA default of 5
    "max_num_steps_ic": 50,
    # Maximum number of the approximate Jacobian or preconditioner␣
↪evaluations
    # allowed when the Newton iteration appears to be slowly␣
↪converging
    # Note: this value differs from the IDA default of 4
    "max_num_jacobians_ic": 40,
    # Maximum number of Newton iterations allowed in any one␣
↪attempt to solve
    # the initial conditions calculation problem
    # Note: this value differs from the IDA default of 10
    "max_num_iterations_ic": 100,
    # Maximum number of linesearch backtracks allowed in any␣
↪Newton iteration,
    # when solving the initial conditions calculation problem
    "max_linesearch_backtracks_ic": 100,
    # Turn off linesearch
    "linesearch_off_ic": False,
    # How to calculate the initial conditions.
    # "True": calculate all y0 given ydot0
    # "False": calculate y_alg0 and ydot_diff0 given y_diff0
    "init_all_y_ic": False,
    # Calculate consistent initial conditions
    "calc_ic": True,
}
```

Note: These options only have an effect if model.convert_to_format == 'casadi'

Extends: *pybamm.solvers.base_solver.BaseSolver*

jaxify(*model*, *t_eval*, *, *output_variables=None*, *calculate_sensitivities=True*, *t_interp=None*)

JAXify the solver object

Creates a JAX expression representing the IDAKLU-wrapped solver object.

Parameters

- **model** (*pybamm.BaseModel*) – The model to be solved
- **t_eval** (*numeric type, optional*) – The times at which to stop the integration due to a discontinuity in time.
- **output_variables** (*list of str, optional*) – The variables to be returned. If None, all variables in the model are used.
- **calculate_sensitivities** (*bool, optional*) – Whether to calculate sensitivities. Default is True.
- **t_interp** (*None, list or ndarray, optional*) – The times (in seconds) at which to interpolate the solution. Defaults to *None*, which returns the adaptive time-stepping times.

set_up(*model*, *inputs=None*, *t_eval=None*, *ics_only=False*)

Unpack model, perform checks, and calculate jacobian.

Parameters

- **model** (*pybamm.BaseModel*) – The model whose solution to calculate. Must have attributes rhs and initial_conditions
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **t_eval** (*numeric type, optional*) – The times at which to stop the integration due to a discontinuity in time.

4.7.6 IDAKLU-JAX Interface

> **Note**
>
> The IDAKLU-Jax interface is experimental, unstable, and untested on Windows.

class pybamm.**IDAKLUJax**(*solver*, *model*, *t_eval*, *output_variables=None*, *calculate_sensitivities=True*, *t_interp=None*)

JAX wrapper for IDAKLU solver

Objects of this class should be created via an IDAKLUSolver object.

Log information is available for this module via the named 'pybamm.solvers.idaklu_jax' logger.

Parameters

solver (*pybamm.IDAKLUSolver*) – The IDAKLU solver object to be wrapped

get_jaxpr()

Returns a JAX expression representing the IDAKLU-wrapped solver object

Returns

A JAX expression with the following call signature:
f(t, inputs=None)

where:

t
[float | np.ndarray] Time sample or vector of time samples

inputs
[dict, optional] dictionary of input values, e.g. {'Current function [A]': 0.222, 'Separator porosity': 0.3}

Return type
Callable

`get_var`(**args*)

Helper function to extract a single variable

Isolates a single variable from the model output. Can be called on a JAX expression (which returns a JAX expression), or on a numeric (np.ndarray) object (which returns a slice of the output).

Example call using default JAX expression, returns a JAX expression:

```
f = idaklu_jax.get_var("Voltage [V]")
data = f(t, inputs=None)
```

Example call using a custom function, returns a JAX expression:

```
f = idaklu_jax.get_var(jax.jit(f), "Voltage [V]")
data = f(t, inputs=None)
```

Example call to slice a matrix, returns an np.array:

```
data = idaklu_jax.get_var(
    jax.fwd(f, argnums=1)(t_eval, inputs)['Current function [A]'],
    'Voltage [V]'
)
```

Parameters

- **f** (*Callable | np.ndarray, optional*) – Expression or array from which to extract the target variable
- **varname** (*str*) – The name of the variable to extract

Returns

- *Callable* – If called with a JAX expression, returns a JAX expression with the following call signature:

  f(t, inputs=None)

  where:

  t
  [float | np.ndarray] Time sample or vector of time samples

  inputs
  [dict, optional] dictionary of input values, e.g. {'Current function [A]': 0.222, 'Separator porosity': 0.3}

- *np.ndarray* – If called with a numeric (np.ndarray) object, returns a slice of the output corresponding to the target variable.

get_vars(**args*)

Helper function to extract a list of variables

Isolates a list of variables from the model output. Can be called on a JAX expression (which returns a JAX expression), or on a numeric (np.ndarray) object (which returns a slice of the output).

Example call using default JAX expression, returns a JAX expression:

```
f = idaklu_jax.get_vars(["Voltage [V]", "Current [A]"])
data = f(t, inputs=None)
```

Example call using a custom function, returns a JAX expression:

```
f = idaklu_jax.get_vars(jax.jit(f), ["Voltage [V]", "Current [A]"])
data = f(t, inputs=None)
```

Example call to slice a matrix, returns an np.array:

```
data = idaklu_jax.get_vars(
    jax.fwd(f, argnums=1)(t_eval, inputs)['Current function [A]'],
    ["Voltage [V]", "Current [A]"]
)
```

Parameters

- **f** (*Callable | np.ndarray, optional*) – Expression or array from which to extract the target variables
- **varname** (*list of str*) – The names of the variables to extract

Returns

- *Callable* – If called with a JAX expression, returns a JAX expression with the following call signature:

  f(t, inputs=None)

  where:

  t
  [float | np.ndarray] Time sample or vector of time samples

  inputs
  [dict, optional] dictionary of input values, e.g. {'Current function [A]': 0.222, 'Separator porosity': 0.3}

- *np.ndarray* – If called with a numeric (np.ndarray) object, returns a slice of the output corresponding to the target variables.

jax_grad(*t: ndarray = None*, *inputs: dict | None = None*, *output_variables: list[str] | None = None*)

Helper function to compute the gradient of a jaxified expression

Returns a numeric (np.ndarray) object (not a JAX expression). Parameters are inferred from the base object, but can be overridden.

Parameters

- **t** (*float | np.ndarray*) – Time sample or vector of time samples
- **inputs** (*dict*) – dictionary of input values

- **output_variables** (*list of str, optional*) – The variables to be returned. If None, the variables in the model are used.

jax_value(*t: ndarray = None*, *inputs: dict | None = None*, *output_variables: list[str] | None = None*)

Helper function to compute the gradient of a jaxified expression

Returns a numeric (np.ndarray) object (not a JAX expression). Parameters are inferred from the base object, but can be overridden.

Parameters

- **t** (*float | np.ndarray*) – Time sample or vector of time samples
- **inputs** (*dict*) – dictionary of input values
- **output_variables** (*list of str, optional*) – The variables to be returned. If None, the variables in the model are used.

jaxify(*model*, *t_eval*, ***, *output_variables=None*, *calculate_sensitivities=True*, *t_interp=None*)

JAXify the model and solver

Creates a JAX expression representing the IDAKLU-wrapped solver object.

Parameters

- **model** (*pybamm.BaseModel*) – The model to be solved
- **t_eval** (*numeric type, optional*) – The times at which to stop the integration due to a discontinuity in time.
- **output_variables** (*list of str, optional*) – The variables to be returned. If None, the variables in the model are used.
- **calculate_sensitivities** (*bool, optional*) – Whether to calculate sensitivities. Default is True.
- **t_interp** (*None, list or ndarray, optional*) – The times (in seconds) at which to interpolate the solution. Defaults to None. Only valid for solvers that support intra-solve interpolation (*IDAKLUSolver*).

4.7.7 Casadi Solver

class pybamm.**CasadiSolver**(*mode='safe'*, *rtol=1e-06*, *atol=1e-06*, *root_method='casadi'*, *root_tol=1e-06*, *max_step_decrease_count=5*, *dt_max=None*, *extrap_tol=None*, *extra_options_setup=None*, *extra_options_call=None*, *return_solution_if_failed_early=False*, *perturb_algebraic_initial_conditions=None*, *integrators_maxcount=100*)

Solve a discretised model, using CasADi.

Parameters

- **mode** (*str*) – How to solve the model (default is "safe"):
  - "fast": perform direct integration, without accounting for events. Recommended when simulating a drive cycle or other simulation where no events should be triggered.
  - "fast with events": perform direct integration of the whole timespan, then go back and check where events were crossed. Experimental only.
  - "safe": perform step-and-check integration in global steps of size dt_max, checking whether events have been triggered. Recommended for simulations of a full charge or discharge.

  - ”safe without grid”: perform step-and-check integration step-by-step. Takes more steps than “safe” mode, but doesn't require creating the grid each time, so may be faster. Experimental only.

- **rtol** (*float, optional*) – The relative tolerance for the solver (default is 1e-6).
- **atol** (*float, optional*) – The absolute tolerance for the solver (default is 1e-6).
- **root_method** (*str or pybamm algebraic solver class, optional*) – The method to use to find initial conditions (for DAE solvers). If a solver class, must be an algebraic solver class. If “casadi”, the solver uses casadi's Newton rootfinding algorithm to find initial conditions. Otherwise, the solver uses ‘scipy.optimize.root’ with method specified by ‘root_method’ (e.g. “lm”, “hybr”, ...)
- **root_tol** (*float, optional*) – The tolerance for root-finding. Default is 1e-6.
- **max_step_decrease_count** (*float, optional*) – The maximum number of times step size can be decreased before an error is raised. Default is 5.
- **dt_max** (*float, optional*) – The maximum global step size (in seconds) used in “safe” mode. If None the default value is 600 seconds.
- **extrap_tol** (*float, optional*) – The tolerance to assert whether extrapolation occurs or not. Default is 0.
- **extra_options_setup** (*dict, optional*) – Any options to pass to the CasADi integrator when creating the integrator. Please consult CasADi documentation for details. Some useful options:
  - ”max_num_steps”: Maximum number of integrator steps
  - ”print_stats”: Print out statistics after integration
- **extra_options_call** (*dict, optional*) – Any options to pass to the CasADi integrator when calling the integrator. Please consult CasADi documentation for details.
- **return_solution_if_failed_early** (*bool, optional*) – Whether to return a Solution object if the solver fails to reach the end of the simulation, but managed to take some successful steps. Default is False.
- **perturb_algebraic_initial_conditions** (*bool, optional*) – Whether to perturb algebraic initial conditions to avoid a singularity. This can sometimes slow down the solver, but is kept True as default for “safe” mode as it seems to be more robust (False by default for other modes).
- **integrators_maxcount** (*int, optional*) – The maximum number of integrators that the solver will retain before ejecting past integrators using an LRU methodology. A value of 0 or None leaves the number of integrators unbound. Default is 100.

Extends: *pybamm.solvers.base_solver.BaseSolver*

create_integrator(*model*, *inputs*, *t_eval=None*, *use_event_switch=False*)

Method to create a casadi integrator object. If t_eval is provided, the integrator uses t_eval to make the grid. Otherwise, the integrator has grid [0,1].

4.7.8 Algebraic Solvers

class pybamm.**AlgebraicSolver**(*method='lm'*, *tol=1e-06*, *extra_options=None*)

Solve a discretised model which contains only (time independent) algebraic equations using a root finding algorithm. Uses scipy.optimize.root. Note: this solver could be extended for quasi-static models, or models in which the time derivative is manually discretised and results in a (possibly nonlinear) algebaric system at each time level.

Parameters

- **method** (*str, optional*) – The method to use to solve the system (default is "lm"). If it starts with "lsq", least-squares minimization is used. The method for least-squares can be specified in the form "lsq_methodname"
- **tol** (*float, optional*) – The tolerance for the solver (default is 1e-6).
- **extra_options** (*dict, optional*) – Any options to pass to the rootfinder. Vary depending on which method is chosen. Please consult SciPy documentation for details.

Extends: *pybamm.solvers.base_solver.BaseSolver*

class pybamm.**CasadiAlgebraicSolver**(*tol=1e-06*, *extra_options=None*)

Solve a discretised model which contains only (time independent) algebraic equations using CasADi's root finding algorithm. Note: this solver could be extended for quasi-static models, or models in which the time derivative is manually discretised and results in a (possibly nonlinear) algebaric system at each time level.

Parameters

- **tol** (*float, optional*) – The tolerance for the solver (default is 1e-6).
- **extra_options** (*dict, optional*) – Any options to pass to the CasADi rootfinder. Please consult CasADi documentation for details.

Extends: *pybamm.solvers.base_solver.BaseSolver*

4.7.9 Solutions

class pybamm.**Solution**(*all_ts*, *all_ys*, *all_models*, *all_inputs*, *t_event=None*, *y_event=None*, *termination='final time'*, *all_sensitivities=False*, *all_yps=None*, *variables_returned=False*, *check_solution=True*)

Class containing the solution of, and various attributes associated with, a PyBaMM model.

Parameters

- **all_ts** (numpy.array, size (n,) (or list of these)) – A one-dimensional array containing the times at which the solution is evaluated. A list of times can be provided instead to initialize a solution with sub-solutions.
- **all_ys** (numpy.array, size (m, n) (or list of these)) – A two-dimensional array containing the values of the solution. y[i, :] is the vector of solutions at time t[i]. A list of ys can be provided instead to initialize a solution with sub-solutions.
- **all_models** (*pybamm.BaseModel*) – The model that was used to calculate the solution. A list of models can be provided instead to initialize a solution with sub-solutions that have been calculated using those models.
- **all_inputs** (*dict (or list of these)*) – The inputs that were used to calculate the solution A list of inputs can be provided instead to initialize a solution with sub-solutions.
- **t_event** (numpy.array, size (1,)) – A zero-dimensional array containing the time at which the event happens.
- **y_event** (numpy.array, size (m,)) – A one-dimensional array containing the value of the solution at the time when the event happens.
- **termination** (*str*) – String to indicate why the solution terminated
- **all_sensitivities** (*bool or dict of lists*) – True if sensitivities included as the solution of the explicit forwards equations. False if no sensitivities included/wanted. Dict if sensitivities are provided as a dict of {parameter: [sensitivities]} pairs.

- **variables_returned** (*bool*) – Bool to indicate if *all_ys* contains the full state vector, or is empty because only requested variables have been returned. True if *output_variables* is used with a solver, otherwise False.

property all_models

Model(s) used for solution

property first_state

A Solution object that only contains the first state. This is faster to evaluate than the full solution when only the first state is needed (e.g. to initialize a model with the solution)

get_data_dict(*variables=None*, *short_names=None*, *cycles_and_steps=True*)

Construct a (standard python) dictionary of the solution data containing the variables in *variables*. If *variables* is None then all variables are returned. Any variable names in short_names are replaced with the corresponding short name.

If the solution has cycles, then the cycle numbers and step numbers are also returned in the dictionary.

Parameters

- **variables** (*list, optional*) – List of variables to return. If None, returns all variables in solution.data
- **short_names** (*dict, optional*) – Dictionary of shortened names to use when saving.
- **cycles_and_steps** (*bool, optional*) – Whether to include the cycle numbers and step numbers in the dictionary

Returns

A dictionary of the solution data

Return type

dict

property last_state

A Solution object that only contains the final state. This is faster to evaluate than the full solution when only the final state is needed (e.g. to initialize a model with the solution)

plot(*output_variables=None*, ***kwargs*)

A method to quickly plot the outputs of the solution. Creates a *pybamm.QuickPlot* object (with keyword arguments 'kwargs') and then calls *pybamm.QuickPlot.dynamic_plot()*.

Parameters

- **output_variables** (*list, optional*) – A list of the variables to plot.
- ****kwargs** – Additional keyword arguments passed to *pybamm.QuickPlot.dynamic_plot()*. For a list of all possible keyword arguments see *pybamm.QuickPlot*.

plot_voltage_components(*ax=None*, *show_legend=True*, *split_by_electrode=False*, *show_plot=True*, ***kwargs_fill*)

Generate a plot showing the component overpotentials that make up the voltage

Parameters

- **ax** (*matplotlib Axis, optional*) – The axis on which to put the plot. If None, a new figure and axis is created.
- **show_legend** (*bool, optional*) – Whether to display the legend. Default is True.

- **split_by_electrode** (*bool, optional*) – Whether to show the overpotentials for the negative and positive electrodes separately. Default is False.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.
- **kwargs_fill** – Keyword arguments, passed to ax.fill_between.

save(*filename*)

Save the whole solution using pickle

save_data(*filename=None*, *variables=None*, *to_format='pickle'*, *short_names=None*)

Save solution data only (raw arrays)

Parameters

- **filename** (*str, optional*) – The name of the file to save data to. If None, then a str is returned
- **variables** (*list, optional*) – List of variables to save. If None, saves all of the variables that have been created so far
- **to_format** (*str, optional*) – The format to save to. Options are:
  - 'pickle' (default): creates a pickle file with the data dictionary
  - 'matlab': creates a .mat file, for loading in matlab
  - 'csv': creates a csv file (0D variables only)
  - 'json': creates a json file
- **short_names** (*dict, optional*) – Dictionary of shortened names to use when saving. This may be necessary when saving to MATLAB, since no spaces or special characters are allowed in MATLAB variable names. Note that not all the variables need to be given a short name.

Returns

data – str if 'csv' or 'json' is chosen and filename is None, otherwise None

Return type

str, optional

property sensitivities

np_array

Type

Values of the sensitivities. Returns a dict of param_name

property sub_solutions

List of sub solutions that have been concatenated to form the full solution

property t

Times at which the solution is evaluated

property t_event

Time at which the event happens

property termination

Reason for termination

update(*variables*)

Add ProcessedVariables to the dictionary of variables in the solution

property y

Values of the solution

property y_event

Value of the solution at the time of the event

4.7.10 Post-Process Variables

class pybamm.**ProcessedVariable**(*base_variables*, *base_variables_casadi*, *solution*, *time_integral: ProcessedVariableTimeIntegral | None = None*)

An object that can be evaluated at arbitrary (scalars or vectors) t and x, and returns the (interpolated) value of the base variable at that t and x.

Parameters

- **base_variables** (list of *pybamm.Symbol*) – A list of base variables with a method *evaluate(t,y)*, each entry of which returns the value of that variable for that particular sub-solution. A Solution can be comprised of sub-solutions which are the solutions of different models. Note that this can be any kind of node in the expression tree, not just a *pybamm.Variable*. When evaluated, returns an array of size (m,n)
- **base_variables_casadi** (list of casadi.Function) – A list of casadi functions. When evaluated, returns the same thing as *base_Variable.evaluate* (but more efficiently).
- **solution** (*pybamm.Solution*) – The solution object to be used to create the processed variables
- **time_integral** (pybamm.ProcessedVariableTimeIntegral, optional) – Not none if the variable is to be time-integrated (default is None)

property data

Same as entries, but different name

property entries

Returns the raw data entries of the processed variable. If the processed variable has not been initialized (i.e. the entries have not been calculated), then the processed variable is initialized first.

initialise_sensitivity_explicit_forward()

Set up the sensitivity dictionary

observe_and_interp(*t*, *fill_value*)

Interpolate the variable at the given time points and y values. t must be a sorted array of time points.

observe_raw()

Evaluate the base variable at the given time points and y values.

property sensitivities

Returns a dictionary of sensitivities for each input parameter. The keys are the input parameters, and the value is a matrix of size (n_x * n_t, n_p), where n_x is the number of states, n_t is the number of time points, and n_p is the size of the input parameter

4.7.11 Summary Variables

class pybamm.**SummaryVariables**(*solution:* Solution, *cycle_summary_variables: list[*SummaryVariables*] | None = None*, *esoh_solver:* ElectrodeSOHSolver *| None = None*, *user_inputs: dict[str, Any] | None = None*)

Class for managing and calculating summary variables from a PyBaMM solution. Summary variables are only calculated when simulations are run with PyBaMM Experiments.

Parameters

- **solution** (*pybamm.Solution*) – The solution object to be used for creating the processed variables.
- **cycle_summary_variables** (*list[pybamm.SummaryVariables], optional*) – A list of cycle summary variables.
- **esoh_solver** (*pybamm.lithium_ion.ElectrodeSOHSolver*, optional) – Solver for electrode state-of-health (eSOH) calculations.
- **user_inputs** (*dict, optional*) – Additional user inputs for calculations.

cycle_number

Stores the cycle number for each saved cycle, for use when plotting. Length is equal to the number of cycles in a solution.

Type
array[int]

property all_variables: list[str]

Return names of all possible summary variables, including eSOH variables if appropriate.

property esoh_variables: list[str] | None

Return names of all eSOH variables.

update(*var: str*)

Compute and store a variable and its change.

update_esoh()

Create all aggregated eSOH variables

4.8 Experiments

Classes to help set operating conditions for some standard battery modelling experiments

4.8.1 Base Experiment Class

class pybamm.**Experiment**(*operating_conditions: list[str | tuple[str] | BaseStep], period: str | None = None, temperature: float | None = None, termination: list[str] | None = None*)

Base class for experimental conditions under which to run the model. In general, a list of operating conditions should be passed in. Each operating condition should be either a *pybamm.step.BaseStep* class, which can be created using one of the methods *pybamm.step.current*, *pybamm.step.c_rate*, *pybamm.step.voltage* , *pybamm.step.power*, *pybamm.step.resistance*, or *pybamm.step.string*, or a string, in which case the string is passed to *pybamm.step.string*.

Parameters

- **operating_conditions** (*list[str]*) – List of strings representing the operating conditions.
- **period** (*str, optional*) – Period (1/frequency) at which to record outputs. Default is 1 minute. Can be overwritten by individual operating conditions.
- **temperature** (*float, optional*) – The ambient air temperature in degrees Celsius at which to run the experiment. Default is None whereby the ambient temperature is taken from the parameter set. This value is overwritten if the temperature is specified in a step.

- **termination** (*list[str], optional*) – List of strings representing the conditions to terminate the experiment. Default is None. This is different from the termination for individual steps. Termination for individual steps is specified in the step itself, and the simulation moves to the next step when the termination condition is met (e.g. 2.5V discharge cut-off). Termination for the experiment as a whole is specified here, and the simulation stops when the termination condition is met (e.g. 80% capacity).

static read_termination(*termination*)

Read the termination reason. If this condition is hit, the experiment will stop.

Parameters

termination (*str or list[str], optional*) – A single string, or a list of strings, representing the conditions to terminate the experiment. Only capacity or voltage can be provided as a termination reason. e.g. '4 Ah capacity' or ['80% capacity', '2.5 V']

Returns

A dictionary of the termination conditions. e.g. {'capacity': (4.0, 'Ah')} or {'capacity': (80.0, '%'), 'voltage': (2.5, 'V')}

Return type

dict

search_tag(*tag*)

Search for a tag in the experiment and return the cycles in which it appears.

Parameters

tag (*str*) – The tag to search for

Returns

A list of cycles in which the tag appears

Return type

list

4.8.2 Experiment step functions

The following functions can be used to define steps in an experiment. Note that the drive cycle must start at t=0

pybamm.step.**string**(*text*, ***kwargs*)

Create a step from a string.

Parameters

- **text** (*str*) – The string to parse. Each operating condition should be of the form "Do this for this long" or "Do this until this happens". For example, "Charge at 1 C for 1 hour", or "Charge at 1 C until 4.2 V", or "Charge at 1 C for 1 hour or until 4.2 V". The instructions can be of the form "(Dis)charge at x A/C/W", "Rest", or "Hold at x V until y A". The running time should be a time in seconds, minutes or hours, e.g. "10 seconds", "3 minutes" or "1 hour". The stopping conditions should be a circuit state, e.g. "1 A", "C/50" or "3 V".
- ****kwargs** – Any other keyword arguments are passed to the step class

Returns

A step parsed from the string.

Return type

pybamm.step.BaseStep

pybamm.step.**current**(*value*, ***kwargs*)

Current-controlled step, see pybamm.step.Current.

pybamm.step.**voltage**(**args*, ***kwargs*)

Voltage-controlled step, see pybamm.step.Voltage.

pybamm.step.**power**(*value*, ***kwargs*)

Power-controlled step, see pybamm.step.Power.

pybamm.step.**resistance**(*value*, ***kwargs*)

Resistance-controlled step, see pybamm.step.Resistance.

These functions return the following step class, which is not intended to be used directly:

class pybamm.step.**BaseStep**(*value*, *duration=None*, *termination=None*, *period=None*, *temperature=None*, *tags=None*, *start_time=None*, *description=None*, *direction: str | None = None*)

Class representing one step in an experiment. All experiment steps are functions that return an instance of this class. This class is not intended to be used directly, but can be subtyped to create a custom experiment step.

Parameters

- **value** (*float*) – The value of the step, corresponding to the type of step. Can be a number, a 2-tuple (for cccv_ode), a 2-column array (for drive cycles), or a 1-argument function of t
- **duration** (*float, optional*) – The duration of the step in seconds.
- **termination** (*str or list, optional*) – A string or list of strings indicating the condition(s) that will terminate the step. If a list, the step will terminate when any of the conditions are met.
- **period** (*float or string, optional*) – The period of the step. If a float, the value is in seconds. If a string, the value should be a valid time string, e.g. "1 hour".
- **temperature** (*float or string, optional*) – The temperature of the step. If a float, the value is in Kelvin. If a string, the value should be a valid temperature string, e.g. "25 oC".
- **tags** (*str or list, optional*) – A string or list of strings indicating the tags associated with the step.
- **start_time** (*str or datetime, optional*) – The start time of the step.
- **description** (*str, optional*) – A description of the step.
- **direction** (*str, optional*) – The direction of the step, e.g. "Charge" or "Discharge" or "Rest".

basic_repr()

Return a basic representation of the step, only with type, value, termination and temperature, which are the variables involved in processing the model. Also used for hashing.

copy()

Return a copy of the step.

Returns
A copy of the step.

Return type
pybamm.Step

default_duration(*value*)

Default duration for the step is one day (24 hours) or the duration of the drive cycle

record_tags(*value*, *duration*, *termination*, *period*, *temperature*, *tags*, *start_time*, *description*, *direction*)

Record all the args for repr and hash

setup_timestepping(*solver*, *tf*, *t_interp=None*)

Setup timestepping for the model.

Parameters

- **solver** (*:class`pybamm.BaseSolver`*) – The solver
- **tf** (*float*) – The final time
- **t_interp** (*np.array | None*) – The time points at which to interpolate the solution

to_dict()

Convert the step to a dictionary.

Returns
A dictionary containing the step information.

Return type
dict

value_based_charge_or_discharge()

Determine whether the step is a charge or discharge step based on the value of the step

Custom steps

Custom steps can be defined using either explicit or implicit control:

class pybamm.step.**CustomStepExplicit**(*current_value_function*, ***kwargs*)

Custom step class where the current value is explicitly given as a function of other variables. When using this class, the user must be careful not to create an expression that depends on the current itself, as this will lead to a circular dependency. For example, in some models, the voltage is an explicit function of the current, so the user should not create a step that depends on the voltage. An expression that works for one model may not work for another.

Parameters

- **current_value_function** (*callable*) – A function that takes in a dictionary of variables and returns the current value.
- **duration** (*float, optional*) – The duration of the step in seconds.
- **termination** (*str or list, optional*) – A string or list of strings indicating the condition(s) that will terminate the step. If a list, the step will terminate when any of the conditions are met.
- **period** (*float or string, optional*) – The period of the step. If a float, the value is in seconds. If a string, the value should be a valid time string, e.g. "1 hour".
- **temperature** (*float or string, optional*) – The temperature of the step. If a float, the value is in Kelvin. If a string, the value should be a valid temperature string, e.g. "25 oC".
- **tags** (*str or list, optional*) – A string or list of strings indicating the tags associated with the step.
- **start_time** (*str or datetime, optional*) – The start time of the step.

- **description** (*str, optional*) – A description of the step.
- **direction** (*str, optional*) – The direction of the step, e.g. "Charge" or "Discharge" or "Rest".

Examples

Control the current to always be equal to a target power divided by voltage (this is one way to implement a power control step):

```
>>> def current_function(variables):
...     P = 4
...     V = variables["Voltage [V]"]
...     return P / V
```

Create the step with a 2.5 V termination condition:

```
>>> step = pybamm.step.CustomStepExplicit(current_function, termination="2.5V")
```

Extends: `pybamm.experiment.step.base_step.BaseStepExplicit`

copy()

Return a copy of the step.

Returns

A copy of the step.

Return type

`pybamm.Step`

class pybamm.step.**CustomStepImplicit**(*current_rhs_function*, *control='algebraic'*, ***kwargs*)

Custom step, see *pybamm.step.BaseStep* for arguments.

Parameters

- **current_rhs_function** (*callable*) – A function that takes in a dictionary of variables and returns the equation controlling the current.
- **control** (*str, optional*) – Whether the control is algebraic or differential. Default is algebraic, in which case the equation is

$$0 = f(\text{variables})$$

where f is the current_rhs_function.

If control is "differential", the equation is

$$\frac{dI}{dt} = f(\text{variables})$$

- **duration** (*float, optional*) – The duration of the step in seconds.
- **termination** (*str or list, optional*) – A string or list of strings indicating the condition(s) that will terminate the step. If a list, the step will terminate when any of the conditions are met.
- **period** (*float or string, optional*) – The period of the step. If a float, the value is in seconds. If a string, the value should be a valid time string, e.g. "1 hour".
- **temperature** (*float or string, optional*) – The temperature of the step. If a float, the value is in Kelvin. If a string, the value should be a valid temperature string, e.g. "25 oC".

- **tags** (*str or list, optional*) – A string or list of strings indicating the tags associated with the step.
- **start_time** (*str or datetime, optional*) – The start time of the step.
- **description** (*str, optional*) – A description of the step.
- **direction** (*str, optional*) – The direction of the step, e.g. "Charge" or "Discharge" or "Rest".

Examples

Control the current so that the voltage is constant (without using the built-in voltage control):

```
>>> def voltage_control(variables):
...     V = variables["Voltage [V]"]
...     return V - 4.2
```

Create the step with a duration of 1h. In this case we don't need to specify that the control is algebraic, as this is the default.

```
>>> step = pybamm.step.CustomStepImplicit(voltage_control, duration=3600)
```

Alternatively, control the current by a differential equation to achieve a target power:

```
>>> def power_control(variables):
...     V = variables["Voltage [V]"]
...     # Large time constant to avoid large overshoot. The user should be careful
...     # to choose a time constant that is appropriate for the model being used,
...     # as well as choosing the appropriate sign for the time constant.
...     K_V = 100
...     return K_V * (V - 4.2)
```

Create the step with a 2.5 V termination condition. Now we need to specify that the control is differential.

```
>>> step = pybamm.step.CustomStepImplicit(
...     power_control, termination="2.5V", control="differential"
... )
```

Extends: `pybamm.experiment.step.base_step.BaseStepImplicit`

copy()

Return a copy of the step.

Returns
A copy of the step.

Return type
`pybamm.Step`

Step terminations

Standard step termination events are implemented by the following classes, which are called when the termination is specified by a specific string. These classes can be either be called directly or via the string format specified in the class docstring

class pybamm.step.**CrateTermination**(*value*, *operator=None*)

Termination based on C-rate, created when a string termination of the C-rate type (e.g. "C/10") is provided

Extends: *pybamm.experiment.step.step_termination.BaseTermination*

get_event(*variables*, *step*)

See *BaseTermination.get_event()*

class pybamm.step.**CurrentTermination**(*value*, *operator=None*)

Termination based on current, created when a string termination of the current type (e.g. "1A") is provided

Extends: *pybamm.experiment.step.step_termination.BaseTermination*

get_event(*variables*, *step*)

See *BaseTermination.get_event()*

class pybamm.step.**VoltageTermination**(*value*, *operator=None*)

Termination based on voltage, created when a string termination of the voltage type (e.g. "4.2V") is provided

Extends: *pybamm.experiment.step.step_termination.BaseTermination*

get_event(*variables*, *step*)

See *BaseTermination.get_event()*

The following classes can be used to define custom terminations for an experiment step:

class pybamm.step.**BaseTermination**(*value*, *operator=None*)

Base class for a termination event for an experiment step. To create a custom termination, a class must implement *get_event* to return a *pybamm.Event* corresponding to the desired termination. In most cases the class *pybamm.step.CustomTermination* can be used to assist with this.

Parameters

value (*float*) – The value at which the event is triggered

get_event(*variables*, *step*)

Return a *pybamm.Event* object corresponding to the termination event

Parameters

- **variables** (*dict*) – Dictionary of model variables, to be used for selecting the variable(s) that determine the event
- **step** (*pybamm.step.BaseStep*) – Step for which this is a termination event, to be used in some cases to determine the sign of the event.

class pybamm.step.**CustomTermination**(*name*, *event_function*)

Define a custom termination event using a function. This can be used to create an event based on any variable in the model.

Parameters

- **name** (*str*) – Name of the event
- **event_function** (*callable*) – A function that takes in a dictionary of variables and evaluates the event value. Must be positive before the event is triggered and zero when the event is triggered.

Example

Add a cut-off based on negative electrode stoichiometry. The event will trigger when the negative electrode stoichiometry reaches 10%.

```
>>> def neg_stoich_cutoff(variables):
...     return variables["Negative electrode stoichiometry"] - 0.1
```

```
>>> neg_stoich_termination = pybamm.step.CustomTermination(
...     name="Negative stoichiometry cut-off", event_function=neg_stoich_cutoff
... )
```

Extends: *pybamm.experiment.step.step_termination.BaseTermination*

get_event(*variables*, *step*)

See *BaseTermination.get_event()*

4.9 Simulation

class pybamm.**Simulation**(*model*, *experiment=None*, *geometry=None*, *parameter_values=None*, *submesh_types=None*, *var_pts=None*, *spatial_methods=None*, *solver=None*, *output_variables=None*, *C_rate=None*, *discretisation_kwargs=None*)

A Simulation class for easy building and running of PyBaMM simulations.

Parameters

- **model** (*pybamm.BaseModel*) – The model to be simulated
- **experiment** (*pybamm.Experiment* or string or list (optional)) – The experimental conditions under which to solve the model. If a string is passed, the experiment is constructed as *pybamm.Experiment([experiment])*. If a list is passed, the experiment is constructed as *pybamm.Experiment(experiment)*.
- **geometry** (*pybamm.Geometry* (optional)) – The geometry upon which to solve the model
- **parameter_values** (*pybamm.ParameterValues* (optional)) – Parameters and their corresponding numerical values.
- **submesh_types** (*dict (optional)*) – A dictionary of the types of submesh to use on each subdomain
- **var_pts** (*dict (optional)*) – A dictionary of the number of points used by each spatial variable
- **spatial_methods** (*dict (optional)*) – A dictionary of the types of spatial method to use on each domain (e.g. pybamm.FiniteVolume)
- **solver** (*pybamm.BaseSolver* (optional)) – The solver to use to solve the model.
- **output_variables** (*list (optional)*) – A list of variables to plot automatically
- **C_rate** (*float (optional)*) – The C-rate at which you would like to run a constant current (dis)charge.
- **discretisation_kwargs** (*dict (optional)*) – Any keyword arguments to pass to the Discretisation class. See *pybamm.Discretisation* for details.

build(*initial_soc=None*, *inputs=None*)

A method to build the model into a system of matrices and vectors suitable for performing numerical computations. If the model has already been built or solved then this function will have no effect. This method will automatically set the parameters if they have not already been set.

Parameters

- **initial_soc** (*float, optional*) – Initial State of Charge (SOC) for the simulation. Must be between 0 and 1. If given, overwrites the initial concentrations provided in the parameter set.
- **inputs** (*dict, optional*) – A dictionary of input parameters to pass to the model when solving.

build_for_experiment(*initial_soc=None*, *inputs=None*, *solve_kwargs=None*)

Similar to *Simulation.build()*, but for the case of simulating an experiment, where there may be several models and solvers to build.

create_gif(*number_of_images=80*, *duration=0.1*, *output_filename='plot.gif'*)

Generates x plots over a time span of t_eval and compiles them to create a GIF. For more information see *pybamm.QuickPlot.create_gif()*

Parameters

- **number_of_images** (*int (optional)*) – Number of images/plots to be compiled for a GIF.
- **duration** (*float (optional)*) – Duration of visibility of a single image/plot in the created GIF.
- **output_filename** (*str (optional)*) – Name of the generated GIF file.

plot(*output_variables=None*, ***kwargs*)

A method to quickly plot the outputs of the simulation. Creates a *pybamm.QuickPlot* object (with keyword arguments 'kwargs') and then calls *pybamm.QuickPlot.dynamic_plot()*.

Parameters

- **output_variables** (*list, optional*) – A list of the variables to plot.
- ****kwargs** – Additional keyword arguments passed to *pybamm.QuickPlot.dynamic_plot()*. For a list of all possible keyword arguments see *pybamm.QuickPlot*.

plot_voltage_components(*ax=None*, *show_legend=True*, *split_by_electrode=False*, *show_plot=True*, ***kwargs_fill*)

Generate a plot showing the component overpotentials that make up the voltage

Parameters

- **ax** (*matplotlib Axis, optional*) – The axis on which to put the plot. If None, a new figure and axis is created.
- **show_legend** (*bool, optional*) – Whether to display the legend. Default is True.
- **split_by_electrode** (*bool, optional*) – Whether to show the overpotentials for the negative and positive electrodes separately. Default is False.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.
- **kwargs_fill** – Keyword arguments, passed to ax.fill_between.

save(*filename*)

Save simulation using pickle module.

Parameters

filename (*str*) – The file extension can be arbitrary, but it is common to use ".pkl" or ".pickle"

save_model(*filename: str | None = None*, *mesh: bool = False*, *variables: bool = False*)

Write out a discretised model to a JSON file

Parameters

- **mesh** (*bool*) – The mesh used to discretise the model. If false, plotting tools will not be available when the model is read back in and solved.
- **variables** (*bool*) – The discretised variables. Not required to solve a model, but if false tools will not be available. Will automatically save meshes as well, required for plotting tools.
- **filename** (*str, optional*) – The desired name of the JSON file. If no name is provided, one will be created based on the model name, and the current datetime.

solve(*t_eval=None*, *solver=None*, *save_at_cycles=None*, *calc_esoh=True*, *starting_solution=None*, *initial_soc=None*, *callbacks=None*, *showprogress=False*, *inputs=None*, *t_interp=None*, ***kwargs*)

A method to solve the model. This method will automatically build and set the model parameters if not already done so.

Parameters

- **t_eval** (*numeric type, optional*) – The times at which to stop the integration due to a discontinuity in time. Can be provided as an array of times at which to return the solution, or as a list *[t0, tf]* where *t0* is the initial time and *tf* is the final time. If the solver does not support intra-solve interpolation, providing *t_eval* as a list returns the solution at 100 points within the interval *[t0, tf]*. Otherwise, the solution is returned at the times specified in *t_interp* or as a result of the adaptive time-stepping solution. See the *t_interp* argument for more details.

  If not using an experiment or running a drive cycle simulation (current provided as data) *t_eval must* be provided.

  If running an experiment the values in *t_eval* are ignored, and the solution times are specified by the experiment.

  If None and the parameter "Current function [A]" is read from data (i.e. drive cycle simulation) the model will be solved at the times provided in the data.
- **solver** (*pybamm.BaseSolver*, optional) – The solver to use to solve the model. If None, Simulation.solver is used
- **save_at_cycles** (*int or list of ints, optional*) – Which cycles to save the full sub-solutions for. If None, all cycles are saved. If int, every multiple of save_at_cycles is saved. If list, every cycle in the list is saved. The first cycle (cycle 1) is always saved.
- **calc_esoh** (*bool, optional*) – Whether to include eSOH variables in the summary variables. If *False* then only summary variables that do not require the eSOH calculation are calculated. Default is True.
- **starting_solution** (*pybamm.Solution*) – The solution to start stepping from. If None (default), then self._solution is used. Must be None if not using an experiment.

- **initial_soc** (*float, optional*) – Initial State of Charge (SOC) for the simulation. Must be between 0 and 1. If given, overwrites the initial concentrations provided in the parameter set.
- **callbacks** (*list of callbacks, optional*) – A list of callbacks to be called at each time step. Each callback must implement all the methods defined in `pybamm.callbacks.BaseCallback`.
- **showprogress** (*bool, optional*) – Whether to show a progress bar for cycling. If true, shows a progress bar for cycles. Has no effect when not used with an experiment. Default is False.
- **t_interp** (*None, list or ndarray, optional*) – The times (in seconds) at which to interpolate the solution. Defaults to None. Only valid for solvers that support intra-solve interpolation (*IDAKLUSolver*).
- ****kwargs** – Additional key-word arguments passed to *solver.solve*. See *pybamm.BaseSolver.solve()*.

step(*dt*, *solver=None*, *t_eval=None*, *save=True*, *starting_solution=None*, *inputs=None*, ***kwargs*)

A method to step the model forward one timestep. This method will automatically build and set the model parameters if not already done so.

Parameters

- **dt** (*numeric type*) – The timestep over which to step the solution
- **solver** (*pybamm.BaseSolver*) – The solver to use to solve the model.
- **t_eval** (*list or numpy.ndarray, optional*) – An array of times at which to return the solution during the step (Note: t_eval is the time measured from the start of the step, so should start at 0 and end at dt). By default, the solution is returned at t0 and t0 + dt.
- **save** (*bool*) – Turn on to store the solution of all previous timesteps
- **starting_solution** (*pybamm.Solution*) – The solution to start stepping from. If None (default), then self._solution is used
- ****kwargs** – Additional key-word arguments passed to *solver.solve*. See *pybamm.BaseSolver.step()*.

4.10 Plotting

4.10.1 Quick Plot

class pybamm.**QuickPlot**(*solutions*, *output_variables=None*, *labels=None*, *colors=None*, *linestyles=None*, *shading='auto'*, *figsize=None*, *n_rows=None*, *time_unit=None*, *spatial_unit='um'*, *variable_limits='fixed'*, *n_t_linear=100*)

Generates a quick plot of a subset of key outputs of the model so that the model outputs can be easily assessed.

Parameters

- **solutions** ((iter of) *pybamm.Solution* or *pybamm.Simulation*) – The numerical solution(s) for the model(s), or the simulation object(s) containing the solution(s).
- **output_variables** (*list of str, optional*) – List of variables to plot
- **labels** (*list of str, optional*) – Labels for the different models. Defaults to model names

- **colors** (*list of str, optional*) – The colors to loop over when plotting. Defaults to None, in which case the default color loop defined by matplotlib style sheet or rcParams is used.
- **linestyles** (*list of str, optional*) – The linestyles to loop over when plotting. Defaults to ["-", ":", "–", "-."]
- **shading** (*str, optional*) – The shading to use for 2D plots. Defaults to "auto".
- **figsize** (*tuple of floats, optional*) – The size of the figure to make
- **n_rows** (*int, optional*) – The number of rows to use. If None (default), floor(n // sqrt(n)) is used where n = len(output_variables) so that the plot is as square as possible
- **time_unit** (*str, optional*) – Format for the time output ("hours", "minutes", or "seconds")
- **spatial_unit** (*str, optional*) – Format for the spatial axes ("m", "mm", or "um")
- **variable_limits** (*str or dict of str, optional*) – How to set the axis limits (for 0D or 1D variables) or colorbar limits (for 2D variables). Options are:
- **n_t_linear** (*int, optional*) – The number of linearly spaced time points added to the t axis for each sub-solution. Note: this is only used if the solution has hermite interpolation enabled.
  - "fixed" (default): keep all axes fixes so that all data is visible
  - "tight": make axes tight to plot at each time
  - dictionary: fine-grain control for each variable, can be either "fixed" or "tight" or a specific tuple (lower, upper).

create_gif(*number_of_images=80*, *duration=0.1*, *output_filename='plot.gif'*)

Generates x plots over a time span of max_t - min_t and compiles them to create a GIF.

Parameters

- **number_of_images** (*int, optional*) – Number of images/plots to be compiled for a GIF.
- **duration** (*float, optional*) – Duration of visibility of a single image/plot in the created GIF.
- **output_filename** (*str, optional*) – Name of the generated GIF file.

dynamic_plot(*show_plot=True*, *step=None*)

Generate a dynamic plot with a slider to control the time.

Parameters

- **step** (*float, optional*) – For notebook mode, size of steps to allow in the slider. Defaults to 1/100th of the total time.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.

plot(*t*, *dynamic=False*)

Produces a quick plot with the internal states at time t.

Parameters

- **t** (*float*) – Dimensional time (in 'time_units') at which to plot.

- **dynamic** (*bool, optional*) – Determine whether to allocate space for a slider at the bottom of the plot when generating a dynamic plot. If True, creates a dynamic plot with a slider.

reset_axis()

Reset the axis limits to the default values. These are calculated to fit around the minimum and maximum values of all the variables in each subplot

slider_update(*t*)

Update the plot in self.plot() with values at new time

pybamm.**dynamic_plot**(**args*, ***kwargs*)

Creates a *pybamm.QuickPlot* object (with arguments 'args' and keyword arguments 'kwargs') and then calls *pybamm.QuickPlot.dynamic_plot()*. The key-word argument 'show_plot' is passed to the 'dynamic_plot' method, not the *QuickPlot* class.

Returns

plot – The 'QuickPlot' object that was created

Return type

pybamm.QuickPlot

class pybamm.**QuickPlotAxes**

Class to store axes for the QuickPlot

add(*keys*, *axis*)

Add axis

Parameters

- **keys** (*iter*) – Iterable of keys of variables being plotted on the axis
- **axis** (*matplotlib Axis object*) – The axis object

by_variable(*key*)

Get axis by variable name

4.10.2 Plot

pybamm.**plot**(*x*, *y*, *ax=None*, *show_plot=True*, ***kwargs*)

Generate a simple 1D plot. Calls *matplotlib.pyplot.plot* with keyword arguments 'kwargs'. For a list of 'kwargs' see the matplotlib plot documentation

Parameters

- **x** (*pybamm.Array*) – The array to plot on the x axis
- **y** (*pybamm.Array*) – The array to plot on the y axis
- **ax** (*matplotlib Axis, optional*) – The axis on which to put the plot. If None, a new figure and axis is created.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.
- **kwargs** – Keyword arguments, passed to plt.plot

4.10.3 Plot 2D

pybamm.**plot2D**(*x*, *y*, *z*, *ax=None*, *show_plot=True*, ***kwargs*)

Generate a simple 2D plot. Calls *matplotlib.pyplot.contourf* with keyword arguments 'kwargs'. For a list of 'kwargs' see the matplotlib contourf documentation

Parameters

- **x** (*pybamm.Array*) – The array to plot on the x axis. Can be of shape (M, N) or (N, 1)
- **y** (*pybamm.Array*) – The array to plot on the y axis. Can be of shape (M, N) or (M, 1)
- **z** (*pybamm.Array*) – The array to plot on the z axis. Is of shape (M, N)
- **ax** (*matplotlib Axis, optional*) – The axis on which to put the plot. If None, a new figure and axis is created.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.

4.10.4 Plot Voltage Components

pybamm.**plot_voltage_components**(*input_data*, *ax=None*, *show_legend=True*, *split_by_electrode=False*, *show_plot=True*, ***kwargs_fill*)

Generate a plot showing the component overpotentials that make up the voltage

Parameters

- **input_data** (*pybamm.Solution* or *pybamm.Simulation*) – Solution or Simulation object from which to extract voltage components.
- **ax** (*matplotlib Axis, optional*) – The axis on which to put the plot. If None, a new figure and axis is created.
- **show_legend** (*bool, optional*) – Whether to display the legend. Default is True
- **split_by_electrode** (*bool, optional*) – Whether to show the overpotentials for the negative and positive electrodes separately. Default is False.
- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.
- **kwargs_fill** – Keyword arguments: matplotlib.axes.Axes.fill_between

4.10.5 Plot Summary Variables

pybamm.**plot_summary_variables**(*solutions*, *output_variables=None*, *labels=None*, *show_plot=True*, ***kwargs_fig*)

Generate a plot showing/comparing the summary variables.

Parameters

- **solutions** ((iter of) *pybamm.Solution*) – The solution(s) for the model(s) from which to extract summary variables.
- **output_variables** (*list (optional)*) – A list of variables to plot automatically. If None, the default ones are used.
- **labels** (*list (optional)*) – A list of labels to be added to the legend. No labels are added by default.

- **show_plot** (*bool, optional*) – Whether to show the plots. Default is True. Set to False if you want to only display the plot after plt.show() has been called.
- **kwargs_fig** – Keyword arguments, passed to plt.subplots.

4.11 Utility functions

pybamm.**root_dir**()

return the root directory of the PyBaMM install directory

class pybamm.**Timer**

Provides accurate timing.

Example

timer = pybamm.Timer() print(timer.time())

reset()

Resets this timer's start time.

time()

Returns the time (float, in seconds) since this timer was created, or since meth:*reset()* was last called.

class pybamm.**TimerTime**(*value*)

class pybamm.**FuzzyDict**

copy() → a shallow copy of D

get_best_matches(*key*)

Get best matches from keys

search(*keys: str | list[str]*, *print_values: bool = False*)

Search dictionary for keys containing all terms in 'keys'. If print_values is True, both the keys and values will be printed. Otherwise, just the keys will be printed. If no results are found, the best matches are printed.

Parameters

- **keys** (*str or list of str*) – Search term(s)
- **print_values** (*bool, optional*) – If True, print both keys and values. Otherwise, print only keys. Default is False.

pybamm.**load**(*filename*)

Load a saved object

pybamm.**has_jax**()

Check if jax and jaxlib are installed with the correct versions

Returns

True if jax and jaxlib are installed with the correct versions, False if otherwise

Return type

bool

pybamm.**is_jax_compatible**()

Check if the available versions of jax and jaxlib are compatible with PyBaMM

Returns
True if jax and jaxlib are compatible with PyBaMM, False if otherwise

Return type
bool

pybamm.**set_logging_level**(*level*)

Set the logging level for PyBaMM

Parameters
level (str) – The logging level to set. Should be one of 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'

4.12 Callbacks

class pybamm.callbacks.**Callback**

Base class for callbacks, for documenting callback methods.

Callbacks are used to perform actions (e.g. logging, saving) at certain points in the simulation. Each callback method is named *on_<event>*, where *<event>* describes the point at which the callback is called. For example, the callback *on_experiment_start* is called at the start of an experiment simulation. In general, callbacks take a single argument, *logs*, which is a dictionary of information about the simulation. Each callback method should return *None* (the output of the method is ignored).

EXPERIMENTAL - this class is experimental and the callback interface may change in future releases.

on_cycle_end(*logs*)

Called at the end of each cycle in an experiment simulation.

on_cycle_start(*logs*)

Called at the start of each cycle in an experiment simulation.

on_experiment_end(*logs*)

Called at the end of an experiment simulation.

on_experiment_error(*logs*)

Called when a SolverError occurs during an experiment simulation.

For example, this could be used to send an error alert with a bug report when running batch simulations in the cloud.

on_experiment_infeasible_event(*logs*)

Called when an experiment simulation is infeasible due to an event.

on_experiment_infeasible_time(*logs*)

Called when an experiment simulation is infeasible due to reaching maximum time.

on_experiment_start(*logs*)

Called at the start of an experiment simulation.

on_step_end(*logs*)

Called at the end of each step in an experiment simulation.

on_step_start(*logs*)

Called at the start of each step in an experiment simulation.

class pybamm.callbacks.**CallbackList**(*callbacks*)

Container abstracting a list of callbacks, so that they can be called in a single step e.g. *callbacks.on_simulation_end(...)*.

This is done without having to redefine the method each time by using the *callback_loop_decorator* decorator, which is applied to every method that starts with *on_*, using the *inspect* module.

If better control over how the callbacks are called is required, it might be better to be more explicit with the for loop.

Extends: *pybamm.callbacks.Callback*

on_cycle_end(**args*, ***kwargs*)

Called at the end of each cycle in an experiment simulation.

on_cycle_start(**args*, ***kwargs*)

Called at the start of each cycle in an experiment simulation.

on_experiment_end(**args*, ***kwargs*)

Called at the end of an experiment simulation.

on_experiment_error(**args*, ***kwargs*)

Called when a SolverError occurs during an experiment simulation.

For example, this could be used to send an error alert with a bug report when running batch simulations in the cloud.

on_experiment_infeasible_event(**args*, ***kwargs*)

Called when an experiment simulation is infeasible due to an event.

on_experiment_infeasible_time(**args*, ***kwargs*)

Called when an experiment simulation is infeasible due to reaching maximum time.

on_experiment_start(**args*, ***kwargs*)

Called at the start of an experiment simulation.

on_step_end(**args*, ***kwargs*)

Called at the end of each step in an experiment simulation.

on_step_start(**args*, ***kwargs*)

Called at the start of each step in an experiment simulation.

class pybamm.callbacks.**LoggingCallback**(*logfile=None*)

Logging callback, implements methods to log progress of the simulation.

Parameters

logfile (*str, optional*) – Where to send the log output. If None, uses pybamm's logger.

Extends: *pybamm.callbacks.Callback*

on_cycle_end(*logs*)

Called at the end of each cycle in an experiment simulation.

on_cycle_start(*logs*)

Called at the start of each cycle in an experiment simulation.

on_experiment_end(*logs*)

Called at the end of an experiment simulation.

on_experiment_error(*logs*)

Called when a SolverError occurs during an experiment simulation.

For example, this could be used to send an error alert with a bug report when running batch simulations in the cloud.

on_experiment_infeasible_event(*logs*)

Called when an experiment simulation is infeasible due to an event.

on_experiment_infeasible_time(*logs*)

Called when an experiment simulation is infeasible due to reaching maximum time.

on_experiment_start(*logs*)

Called at the start of an experiment simulation.

on_step_end(*logs*)

Called at the end of each step in an experiment simulation.

on_step_start(*logs*)

Called at the start of each step in an experiment simulation.

pybamm.callbacks.**setup_callbacks**(*callbacks*)

4.13 Citations

class pybamm.**Citations**

Entry point to citations management. This object may be used to record BibTeX citation information and then register that a particular citation is relevant for a particular simulation.

Citations listed in *pybamm/CITATIONS.bib* can be registered with their citation key. For all other works provide a BibTeX Citation to *register()*.

Examples

```
>>> pybamm.citations.register("Sulzer2021")
>>> pybamm.citations.register("@misc{Newton1687, title={Mathematical...}}")
>>> pybamm.print_citations("citations.txt")
```

print(*filename=None*, *output_format='text'*, *verbose=False*)

Print all citations that were used for running simulations. The verbose option is provided to print tags for citations in the output such that it can be seen where the citations were registered due to the use of PyBaMM models and solvers in the code.

Note

If a citation is registered manually, it will not be tagged.

Warning

This function will notify the user if a citation that has been previously registered is invalid or cannot be parsed.

Parameters

- **filename** (*str, optional*) – Filename to which to print citations. If None, citations are printed to the terminal.
- **verbose** (*bool, optional*) – If True, prints the citation tags for the citations that have been registered. An example of the output is shown below.

Examples

```
pybamm.lithium_ion.SPM()
pybamm.Citations.print(verbose=True) or pybamm.print_citations(verbose=True)
```

will append the following at the end of the list of citations:

```
Citations registered:

Marquis2019 was cited due to the use of SPM
```

read_citations()

Reads the citations in *pybamm.CITATIONS.bib*. Other works can be cited by passing a BibTeX citation to *register()*.

register(*key*)

Register a paper to be cited, one at a time. The intended use is that *register()* should be called only when the referenced functionality is actually being used.

> **Warning**
>
> Registering a BibTeX citation, with the same key as an existing citation, will overwrite the current citation.

Parameters

key (*str*) –

- The citation key for an entry in *pybamm/CITATIONS.bib* or
- A BibTeX formatted citation

pybamm.**print_citations**(*filename=None*, *output_format='text'*, *verbose=False*)

See *Citations.print()*

4.14 Batch Study

class pybamm.**BatchStudy**(*models*, *experiments=None*, *geometries=None*, *parameter_values=None*, *submesh_types=None*, *var_pts=None*, *spatial_methods=None*, *solvers=None*, *output_variables=None*, *C_rates=None*, *repeats=1*, *permutations=False*)

A BatchStudy class for comparison of different PyBaMM simulations.

Parameters

- **models** (*dict*) – A dictionary of models to be simulated
- **experiments** (*dict (optional)*) – A dictionary of experimental conditions under which to solve the model. Default is None
- **geometries** (*dict (optional)*) – A dictionary of geometries upon which to solve the model

- **parameter_values** (*dict (optional)*) – A dictionary of parameters and their corresponding numerical values. Default is None
- **submesh_types** (*dict (optional)*) – A dictionary of the types of submesh to use on each subdomain. Default is None
- **var_pts** (*dict (optional)*) – A dictionary of the number of points used by each spatial variable. Default is None
- **spatial_methods** (*dict (optional)*) – A dictionary of the types of spatial method to use on each domain. Default is None
- **solvers** (*dict (optional)*) – A dictionary of solvers to use to solve the model. Default is None
- **output_variables** (*dict (optional)*) – A dictionary of variables to plot automatically. Default is None
- **C_rates** (*dict (optional)*) – A dictionary of C-rates at which you would like to run a constant current (dis)charge. Default is None
- **repeats** (*int (optional)*) – The number of times *solve* should be called. Default is 1
- **permutations** (*bool (optional)*) – If False runs first model with first solver, first experiment and second model with second solver, second experiment etc. If True runs a cartesian product of models, solvers and experiments. Default is False

create_gif(*number_of_images=80*, *duration=0.1*, *output_filename='plot.gif'*)

Generates x plots over a time span of t_eval and compiles them to create a GIF. For more information see *pybamm.QuickPlot.create_gif()*

Parameters

- **number_of_images** (*int, optional*) – Number of images/plots to be compiled for a GIF.
- **duration** (*float, optional*) – Duration of visibility of a single image/plot in the created GIF.
- **output_filename** (*str, optional*) – Name of the generated GIF file.

plot(*output_variables=None*, ***kwargs*)

For more information on the parameters used in the plot, See *pybamm.Simulation.plot()*

solve(*t_eval=None*, *solver=None*, *save_at_cycles=None*, *calc_esoh=True*, *starting_solution=None*, *initial_soc=None*, *t_interp=None*, ***kwargs*)

For more information on the parameters used in the solve, See *pybamm.Simulation.solve()*

4.15 PyBaMM Data

class pybamm.**DataLoader**

Data Loader class for downloading and loading data files upstream at https://github.com/pybamm-team/pybamm-data/

The following files are listed in the registry -

4.15.1 COMSOL Results

Andersson *et al.*[1] Doyle *et al.*[2] Harris *et al.*[3] Marquis *et al.*[4] Marquis[5]

- comsol_01C.json
- comsol_05C.json
- comsol_1C.json
- comsol_1plus1D_3C.json
- comsol_2C.json
- comsol_3C.json

4.15.2 Kokam SLPB 75106100 discharge data from Ecker et al (2015)

Ecker *et al.*[6] Ecker *et al.*[7]

- Ecker_1C.csv
- Ecker_5C.csv

4.15.3 Enertech cells - discharge results for beginning of life

Andersson *et al.*[1] Doyle *et al.*[2] Harris *et al.*[3] Marquis *et al.*[4] Ai *et al.*[8] Deshpande *et al.*[9] Timms *et al.*[10]

- 0.1C_discharge_U.txt
- 0.1C_discharge_displacement.txt
- 0.5C_discharge_T.txt
- 0.5C_discharge_U.txt
- 0.5C_discharge_displacement.txt
- 1C_discharge_T.txt
- 1C_discharge_U.txt
- 1C_discharge_displacement.txt
- 2C_discharge_T.txt

[1] Joel A. E. Andersson, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019. doi:10.1007/s12532-018-0139-4.

[2] Marc Doyle, Thomas F. Fuller, and John 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.

[3] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, and others. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.

[4] Scott G. Marquis, Valentin Sulzer, Robert Timms, Colin P. Please, and S. Jon Chapman. An asymptotic derivation of a single particle model with electrolyte. *Journal of The Electrochemical Society*, 166(15):A3693–A3706, 2019. doi:10.1149/2.0341915jes.

[5] Scott G. Marquis. *Long-term degradation of lithium-ion batteries*. PhD thesis, University of Oxford, 2020.

[6] Madeleine Ecker, Thi Kim Dung Tran, Philipp Dechent, Stefan Käbitz, Alexander Warnecke, and Dirk Uwe Sauer. Parameterization of a Physico-Chemical Model of a Lithium-Ion Battery: I. Determination of Parameters. *Journal of the Electrochemical Society*, 162(9):A1836–A1848, 2015. doi:10.1149/2.0551509jes.

[7] Madeleine Ecker, Stefan Käbitz, Izaro Laresgoiti, and Dirk Uwe Sauer. Parameterization of a Physico-Chemical Model of a Lithium-Ion Battery: II. Model Validation. *Journal of The Electrochemical Society*, 162(9):A1849–A1857, 2015. doi:10.1149/2.0541509jes.

[8] Weilong Ai, Ludwig Kraft, Johannes Sturm, Andreas Jossen, and Billy Wu. Electrochemical thermal-mechanical modelling of stress inhomogeneity in lithium-ion pouch cells. *Journal of The Electrochemical Society*, 167(1):013512, 2019. doi:10.1149/2.0122001JES.

[9] Rutooj Deshpande, Mark Verbrugge, Yang-Tse Cheng, John Wang, and Ping Liu. Battery cycle life prediction with coupled chemical degradation and fatigue mechanics. *Journal of the Electrochemical Society*, 159(10):A1730, 2012. doi:10.1149/2.049210jes.

[10] Robert Timms, Scott G Marquis, Valentin Sulzer, Colin P. Please, and S Jonathan Chapman. Asymptotic Reduction of a Lithium-ion Pouch Cell Model. *SIAM Journal on Applied Mathematics*, 81(3):765–788, 2021. doi:10.1137/20M1336898.

- 2C_discharge_U.txt
- 2C_discharge_displacement.txt
- stn_2C.txt
- stp_2C.txt

4.15.4 Drive cycles

Andersson *et al.*[Page 235, 1] Doyle *et al.*[Page 235, 2] Harris *et al.*[Page 235, 3] Marquis *et al.*[Page 235, 4] Marquis[Page 235, 5]

- UDDS.csv
- US06.csv
- WLTC.csv
- car_current.csv

`get_data`(*filename: str*)

Fetches the data file from upstream and stores it in the local cache directory under pybamm directory.

Parameters
filename (*str*) – Name of the data file to be fetched from the registry.

Return type
pathlib.PurePath

`show_registry`()

Prints the name of all the files present in the registry.

Return type
list

References

Version: 25.1.1

Useful links: Project Home Page | *Installation* | Source Repository | Issue Tracker | Discussions

PyBaMM (Python Battery Mathematical Modelling) is an open-source battery simulation package written in Python. Our mission is to accelerate battery modelling research by providing open-source tools for multi-institutional, interdisciplinary collaboration. Broadly, PyBaMM consists of

1. a framework for writing and solving systems of differential equations,
2. a library of battery models and parameters, and
3. specialized tools for simulating battery-specific experiments and visualizing the results.

Together, these enable flexible model definitions and fast battery simulations, allowing users to explore the effect of different battery designs and modeling assumptions under a variety of operating scenarios.

User Guide

The user guide is the best place to start learning PyBaMM. It contains an installation guide, an introduction to the main concepts and links to additional tutorials.

To the user guide

Examples

Examples and tutorials can be viewed on the GitHub examples page, which also provides a link to run them online through Google Colab.

To the examples

API Documentation

The reference guide contains a detailed description of the functions, modules, and objects included in PyBaMM. The reference describes how the methods work and which parameters can be used.

To the API documentation

Contributor's Guide

Contributions to PyBaMM and its development are welcome! If you have ideas for features, bug fixes, models, spatial methods, or solvers, we would love to hear from you.

To the contributor's guide

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z