PyBaMM Documentation

Release 0.2.2

Valentin Sulzer

Mar 12, 2021

Contents

1 Quickstart 3

2 Installation 5

3 API documentation 15

4 Examples 133

5 Contributing 135

Python Module Index 147

Index 149

Python Battery Mathematical Modelling (**PyBAMM**) solves continuum models for batteries, using both numerical methods and asymptotic analysis.

PyBaMM is hosted on GitHub. This page provides the *API*, or *developer documentation* for `pybamm`.

- genindex
- modindex
- search

CHAPTER 1

Quickstart

PyBaMM is available on GNU/Linux, MacOS and Windows.

1.1 Using pip

```
pip install pybamm
```

1.2 Using conda

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

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

1.3 Optional solvers

On GNU/Linux and MacOS, an optional scikits.odes -based solver is available, see *Optional - scikits.odes solver*.

CHAPTER 2

Installation

2.1 GNU-Linux & MacOS

Contents

- *GNU-Linux & MacOS*
  - *Prerequisites*
  - *Install PyBaMM*
    - *User install*
      - *Optional - scikits.odes solver*
    - *Developer install*
  - *Uninstall PyBaMM*
  - *Troubleshooting*

2.1.1 Prerequisites

To use and/or contribute to PyBaMM, you must have Python 3.6 or 3.7 installed (note that 3.8 is not yet supported).

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

```
sudo apt update
sudo apt install python3
```

On Fedora or CentOS, you can use DNF or Yum. For example

```
sudo dnf install python3
```

On Mac OS distributions, you can use `homebrew`. First *install ‘‘brew‘* <https://docs.python-guide.org/starting/install3/osx/>‘__:

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

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

```
brew install python3
```

2.1.2 Install PyBaMM

User install

We recommend to install PyBaMM within a virtual environment, in order not to alter any distribution python files. To create a virtual environment `env` within your current directory type:

```
python3 -m venv env
```

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:

```
pip install pybamm
```

PyBaMM’s dependencies (such as `numpy`, `scipy`, etc) will be installed automatically when you install PyBaMM using `pip`.

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

Optional - scikits.odes solver

Users can install scikits.odes in order to use the wrapped SUNDIALS ODE and DAE solvers.

A pre-requisite is the installation of a BLAS library (such as openblas). On Ubuntu/debian

```
sudo apt install libopenblas-dev
```

and on Mac OS

```
brew install openblas
```

After installing PyBaMM, the following command can be used to automatically install `scikits.odes` and its dependencies

```
$ pybamm_install_odes --install-sundials
```

The `--install-sundials` option is used to activate automatic downloads and installation of the sundials library, which is required by `scikits.odes`.

Developer install

If you wish to contribute to PyBaMM, you should get the latest version from the GitHub repository. To do so, you must have Git and graphviz installed. For instance run

```
sudo apt install git graphviz
```

on Debian-based distributions, or

```
brew install git graphviz
```

on Mac OS.

To install PyBaMM, the first step is to get the code by cloning this repository

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

Then, to install PyBaMM as a developer, type

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

KLU sparse solver If you wish so simulate large systems such as the 2+1D models, we recommend employing a sparse solver. PyBaMM currently offers a direct interface to the sparse KLU solver within Sundials, but it is unlikely to be installed as you may not have all the dependencies available. If you wish to install the KLU from the PyBaMM sources, see *the instructions for compiling the KLU sparse solver*.

To check whether PyBaMM has installed properly, you can run the tests:

```
python3 run-tests.py --unit
```

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

2.1.3 Uninstall PyBaMM

PyBaMM can be uninstalled by running

```
pip uninstall pybamm
```

in your virtual environment.

2.1.4 Troubleshooting

Problem: I've 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.

Problem: When running `python run-tests.py --quick`, gives error `FileNotFoundError: [Errno 2] No such file or directory: 'flake8': 'flake8`.

Solution: make sure you have included the `[dev,docs]` flags when you pip installed PyBaMM, i.e. `pip install -e .[dev,docs]`

Problem: Errors when solving model `ValueError: Integrator name ida does not exsist`, or `ValueError: Integrator name cvode does not exsist`.

Solution: This could mean that you have not installed `scikits.odes` correctly, check the instructions given above and make sure each command was successful.

One possibility is that you have not set your `LD_LIBRARY_PATH` to point to the sundials library, type `echo $LD_LIBRARY_PATH` and make sure one of the directories printed out corresponds to where the sundials libraries are located.

Another common reason is that you forget to install a BLAS library such as OpenBLAS before installing sundials. Check the cmake output when you configured Sundials, it might say:

```
-- A library with BLAS API not found. Please specify library location.
-- LAPACK requires BLAS
```

If this is the case, on a Debian or Ubuntu system you can install OpenBLAS using `sudo apt-get install libopenblas-dev` (or `brew install openblas` for Mac OS) and then re-install sundials using the instructions above.

2.2 Windows

Contents

- *Windows*
  - *Prerequisites*
  - *Install PyBaMM*
    - *User install*
  - *Uninstall PyBaMM*
  - *Installation using WSL*

2.2.1 Prerequisites

To use and/or contribute to PyBaMM, you must have Python 3.6 or 3.7 installed (note that 3.8 is not yet supported).

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.

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

```
python -m venv env
```

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
```

PyBaMM's dependencies (such as `numpy`, `scipy`, etc) will be installed automatically when you install PyBaMM using `pip`.

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

2.2.3 Uninstall PyBaMM

PyBaMM can be uninstalled by running

```
pip uninstall pybamm
```

in your virtual environment.

2.2.4 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 here.

2.3 Windows Subsystem for Linux (WSL)

We recommend the use of Windows Subsystem for Linux (WSL) to install PyBaMM, see the instructions below to get PyBaMM working using Windows, WSL and VSCode.

Contents

- *Windows Subsystem for Linux (WSL)*
  - *Install WSL*
  - *Install PyBaMM*
  - *Use Visual Studio Code to run PyBaMM*

2.3.1 Install WSL

Follow the instructions from Microsoft here. When given the option, choose the Ubuntu 18.04 LTS distribution to install. Don't forget to initialise the Ubuntu installation using the instructions given here.

2.3.2 Install PyBaMM

Open a terminal window in your installed Ubuntu distribution by selecting "Ubuntu" from the start menu. This should give you a bash prompt in your home directory.

To download the PyBaMM source code, you first need to install git, which you can do by typing

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

For easier integration with WSL, we recommend that you install PyBaMM in your *Windows* Documents folder, for example by first navigating to

```
$ cd /mnt/c/Users/USER_NAME/Documents
```

where USER_NAME is your username. Exact path to Windows documents may vary. Now use git to clone the PyBaMM repository:

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

This will create a new directly called `PyBaMM`, you can move to this directory in bash using the `cd` command:

```
cd PyBaMM
```

If you are unfamiliar with the linux command line, you might find it useful to work through this tutorial provided by Ubuntu.

Now head over and follow the installation instructions for PyBaMM for linux here.

2.3.3 Use Visual Studio Code to run PyBaMM

You will probably want to use a native Windows IDE such as Visual Studio Code or the full Microsoft Visual Studio IDE. Both of these packages can connect to WSL so that you can write python code in a native windows environment, while at the same time using WSL to run the code using your installed Ubuntu distribution. The following instructions assume that you are using Visual Studio Code.

First, setup VSCode to run within the `PyBaMM` directory that you created above, using the instructions provided here.

Once you have opened the `PyBaMM` folder in vscode, use the `Extensions` panel to install the `Python` extension from Microsoft. Note that extensions are either installed on the Windows (Local) or on in WSL (WSL:Ubuntu), so even if you have used VSCode previously with the Python extension, you probably haven't installed it in WSL. Make sure to reload after installing the Python extension so that it is available.

If you have installed PyBaMM into the virtual environment `env` as in the PyBaMM linux install guide, then VSCode should automatically start using this environment and you should see something similar to "Python 3.6.8 64-bit ('env': venv)" in the bottom bar.

To test that vscode can run a PyBaMM script, navigate to the `examples/scripts` folder and right click on the `create-model.py` script. Select "Run current file in Python Interactive Window". This should run the script, which sets up and solves a model of SEI thickness using PyBaMM. You should see a plot of SEI thickness versus time pop up in the interactive window.

The Python Interactive Window in VSCode can be used to view plots, but is restricted in functionality and cannot, for example, launch separate windows to show plot. To setup an xserver on windows and use this to launch windows for plotting, follow these instructions:

1. Install VcXsrv from here.
2. Set the display port in the WSL command-line: `echo "export DISPLAY=localhost:0.0" >> ~/.bashrc`

3. Install python3-tk in WSL: `sudo apt-get install python3-tk`
4. Set the matplotlib backend to TKAgg in WSL: `echo "backend : TKAgg" >> ~/.config/matplotlib/matplotlibrc`
5. Before running the code, just launch XLaunch (with the default settings) from within Windows. Then the code works as usual.

2.4 PyBaMM developer install - The KLU sparse solver

If you wish to try a different DAE solver, PyBaMM currently offers a direct interface to the sparse KLU solver within Sundials. This solver comes as a C++ python extension module. Therefore, when installing PyBaMM from source (e.g. from the GitHub repository), the KLU sparse solver module must be compiled. Running `pip install .` or `python setup.py install` in the PyBaMM directory will result in a attempt to compile the KLU module.

Note that if CMake of pybind11 are not found (see below), the installation of PyBaMM will carry on, however skipping the compilation of the `idaklu` module. This allows developers that are not interested in the KLU module to install PyBaMM from source without having to install the required dependencies.

To build the KLU solver, the following dependencies are required:

- A C++ compiler (e.g. `g++`)
- A Fortran compiler (e.g. `gfortran`)
- The python 3 header files
- CMake
- A BLAS implementation (e.g. openblas)
- pybind11
- sundials
- SuiteSparse

The first four should be available through your favourite package manager. On Debian-based GNU/Linux distributions:

```
apt update
apt install python3-dev gcc gfortran cmake libopenblas-dev
```

2.4.1 pybind11

The pybind11 source directory should be located in the PyBaMM project directory at the time of compilation. Simply clone the GitHub repository, for example:

```
# In the PyBaMM project dir (next to setup.py)
git clone https://github.com/pybind/pybind11.git
```

2.4.2 SuiteSparse and sundials

Method 1 - Using the convenience script

The PyBaMM repository contains a script `scripts/setup_KLU_module_build.py` that automatically downloads, extracts, compiles and installs the two libraries.

First install the Python `wget` module

```
pip install wget
```

Then execute the script

```
# In the PyBaMM project dir (next to setup.py)
python scripts/setup_KLU_module_build.py
```

The above will install the required component of SuiteSparse and Sundials in your home directory under `~/.local/`. Note that you can provide the option `--install-dir=<install/path>` to install both libraries to an alternative location. If `<install/path>` is not absolute, it will be interpreted as relative to the PyBaMM project directory.

Finally, reactivate your virtual environment by running

```
source $(VIRTUAL_ENV)/bin/activate
```

Alternatively, you update the `LD_LIBRARY_PATH` environment variable as follows

```
export LD_LIBRARY_PATH=$(HOME)/.local:$LD_LIBRARY_PATH
```

The above export statement will be ran automatically the next time you activate you python virtual environment.

If did not run the convenience script inside a python virtual environment, execute you bash config file

```
source ~/.bashrc
```

(or start a new shell).

Build files are located inside the PyBaMM project directory under `KLU_module_deps/`. Feel free to remove this directory once everything is installed correctly.

Method 2 - Compiling Sundials (advanced)

SuiteSparse

On most current linux distributions and macOS, a recent enough version of the suitesparse source package is available through the package manager. For instance on Fedora

```
yum install libsuitesparse-dev
```

Sundials

The PyBaMM KLU solver requires Sundials >= 4.0. Because most Linux distribution provide older versions through their respective package manager, it is recommended to build and install Sundials manually.

First, download and extract the sundials 5.0.0 source

```
wget https://computing.llnl.gov/projects/sundials/download/sundials-5.0.0.tar.gz .
tar -xvf sundials-5.0.0.tar.gz
```

Then, create a temporary build directory and navigate into it

```
mkdir build_sundials
cd build_sundials
```

You can now configure the build, by running

```
cmake -DLAPACK_ENABLE=ON\
      -DSUNDIALS_INDEX_SIZE=32\
      -DBUILD_ARKODE=OFF\
      -DBUILD_CVODE=OFF\
      -DBUILD_CVODES=OFF\
      -DBUILD_IDAS=OFF\
      -DBUILD_KINSOL=OFF\
      -DEXAMPLES_ENABLE:BOOL=OFF\
      -DKLU_ENABLE=ON\
      -DKLU_INCLUDE_DIR=path/to/suitesparse/headers\
      -DKLU_LIBRARY_DIR=path/to/suitesparse/libraries\
      ../sundials-5.0.0
```

Be careful set the two variables `KLU_INCLUDE_DIR` and `KLU_LIBRARY_DIR` to the correct installation location of the SuiteSparse libary on your system. If you installed SuiteSparse through your package manager, this is likely to be something similar to:

```
-DKLU_INCLUDE_DIR=/usr/include/suitesparse\
-DKLU_LIBRARY_DIR=/usr/lib/x86_64-linux-gnu\
```

By default, Sundials will be installed on your system under `/usr/local` (this varies depending on the distribution). Should you wish to install sundials in a specific location, set the following variable

```
-DCMAKE_INSTALL_PREFIX=install/location\
```

Finally, build the library:

```
make install
```

You may be asked to run this command as a super-user, depending on the installation location.

Alternative installation location

By default, it is assumed that the SuiteSparse and Sundials libraries are installed in your home directory under `~/.local`. If you installed the libraries to (a) different location(s), you must set the options `suitesparse-root` or/and `sundials-root` when installing PyBaMM. Examples:

```
python setup.py install --suitesparse-root=path/to/suitesparse
```

or

```
pip install . --install-option="--sundials-root=path/to/sundials"
```

2.4.3 (re)Install PyBaMM to build the KLU solver

If the above dependencies are correctly installed, any of the following commands will install PyBaMM with the `idaklu` solver module:

```
pip install .
pip install -e .
python setup.py install
python setup.py develop
...
```

Note that it doesn't matter if pybamm is already installed. The above commands will update your exisitng installation by adding the `idaklu` module.

2.4.4 Check that the solver is correctly installed

If you install PyBaMM in editable mode using the `-e` pip switch or if you use the `python setup.py install` command, a log file will be located in the project directory (next to the `setup.py` file).

```
cat setup.log
020-03-24 11:33:50,645 - PyBaMM setup - INFO - Starting PyBaMM setup
2020-03-24 11:33:50,653 - PyBaMM setup - INFO - Not running on windows
2020-03-24 11:33:50,654 - PyBaMM setup - INFO - Could not find CMake. Skipping␣
↪compilation of KLU module.
2020-03-24 11:33:50,655 - PyBaMM setup - INFO - Could not find pybind11 directory (/
↪io/pybind11). Skipping compilation of KLU module.
```

If the KLU sparse solver is correctly installed, then the following command should return `True`.

```
$ python -c "import pybamm; print(pybamm.have_idaklu())"
```

CHAPTER 3

API documentation

3.1 Expression Tree

3.1.1 Symbol

class pybamm.**Symbol**(*name*, *children=None*, *domain=None*, *auxiliary_domains=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" or "tertiary". 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"}*).

__abs__()

return an *AbsoluteValue* object

__add__(*other*)

return an *Addition* object

__ge__(*other*)

return a *EqualHeaviside* object

__getitem__(*key*)

return a *Index* object

__gt__(*other*)
return a *NotEqualHeaviside* object

__init__(*name*, *children=None*, *domain=None*, *auxiliary_domains=None*)
Initialize self. See help(type(self)) for accurate signature.

__le__(*other*)
return a *EqualHeaviside* object

__lt__(*other*)
return a *NotEqualHeaviside* object

__matmul__(*other*)
return a *MatrixMultiplication* object

__mul__(*other*)
return a *Multiplication* object

__neg__()
return a *Negate* object

__pow__(*other*)
return a *Power* object

__radd__(*other*)
return an *Addition* object

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

__rmatmul__(*other*)
return a *MatrixMultiplication* object

__rmul__(*other*)
return a *Multiplication* object

__rpow__(*other*)
return a *Power* object

__rsub__(*other*)
return a *Subtraction* object

__rtruediv__(*other*)
return a *Division* object

__str__()
return a string representation of the node and its children

__sub__(*other*)
return a *Subtraction* object

__truediv__(*other*)
return a *Division* object

auxiliary_domains
Returns domains that are not the primary domain

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*)
Copy the domains from a given symbol, bypassing checks

diff(*variable*)
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

domain
list of applicable domains

Returns

Return type iterable of str

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)
Evaluate expression tree (wrapper to allow using dict of known values). If the dict 'known_evals' is provided, the dict is searched for self.id; if self.id is in the keys, return that value; otherwise, evaluate using `_base_evaluate()` and add that value to known_evals

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)
- **known_evals** (*dict, optional*) – dictionary containing known values (default None)

Returns

- *number or array* – the node evaluated at (t,y)
- **known_evals (if known_evals input is not None)** (*dict*) – the dictionary of known values

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=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()
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.

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_auxiliary_domains(*children*)
Combine auxiliary domains from children, at all levels

has_symbol_of_classes(*symbol_classes*)
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 *u*

See also:

evaluate() evaluate the expression

jac(*variable*, *known_jacs=None*, *clear_domain=True*)
Differentiate a symbol with respect to a (slice of) a StateVector or StateVectorDot. See *pybamm.Jacobian*.

name
name of the node

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

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

pre_order()
returns an iterable that steps through the tree in pre-order fashion

Examples

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

relabel_tree(*symbol*, *counter*)
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)

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).

This is identical to what we'd put in a __hash__ function However, implementing __hash__ requires also implementing __eq__, which would then mess with loop-checking in the anytree module.

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

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

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.

simplify(*simplified_symbols=None*)
Simplify the expression tree. See *pybamm.Simplification*.

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

size_for_testing
Size of an object, based on shape for testing

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=None*, *y=None*, *y_dot=None*, *inputs=None*, *casadi_symbols=None*)
Convert the expression tree to a CasADi expression tree. See *pybamm.CasadiConverter*.

visualise(*filename*)
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"

3.1.2 Parameter

class pybamm.**Parameter**(*name*, *domain=[]*)
A node in the expression tree representing a parameter

This node will be replaced by a *Scalar* node by :class'.Parameter'

Parameters

- **name** (*str*) – name of the node
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list

new_copy()
See *pybamm.Symbol.new_copy()*.

class pybamm.**FunctionParameter**(*name*, *inputs*, *diff_variable=None*)
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.

diff(*variable*)
See *pybamm.Symbol.diff()*.

get_children_domains(*children_list*)
Obtains the unique domain of the children. If the children have different domains then raise an error

new_copy()
See *pybamm.Symbol.new_copy()*.

set_id()
See *pybamm.Symbol.set_id()*

3.1.3 Variable

class pybamm.**Variable**(*name*, *domain=None*, *auxiliary_domains=None*)
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*) – list of domains that this variable is valid over
- **auxiliary_domains** (*dict*) – dictionary of auxiliary domains ({'secondary': ..., 'tertiary': ...}). 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'
- ***Extends** –

diff(*variable*)
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*, *domain=None*, *auxiliary_domains=None*)
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': ...}). 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'
- ***Extends** –

diff(*variable*)
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()
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

class pybamm.**ExternalVariable**(*name*, *size*, *domain=None*, *auxiliary_domains=None*)
A node in the expression tree representing an external variable variable

This node will be discretised by *Discretisation* and converted to a *Vector* 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': ...}). 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'
- ***Extends** –

diff(*variable*)
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

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

3.1.4 Independent Variable

class pybamm.**IndependentVariable**(*name*, *domain=None*, *auxiliary_domains=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

- ***Extends** –

class pybamm.**Time**
A node in the expression tree representing time

Extends: *Symbol*

new_copy()
See *pybamm.Symbol.new_copy()*.

class pybamm.**SpatialVariable**(*name*, *domain=None*, *auxiliary_domains=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")
- ***Extends** –

new_copy()
See *pybamm.Symbol.new_copy()*.

pybamm.**t = the independent variable time**
A node in the expression tree representing time

Extends: *Symbol*

3.1.5 Scalar

class pybamm.**Scalar**(*value*, *name=None*, *domain=[]*)
A node in the expression tree representing a scalar value

Extends: *Symbol*

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
- **domain** (*iterable of str, optional*) – list of domains the parameter is valid over, defaults to empty list

new_copy()
See *pybamm.Symbol.new_copy()*.

set_id()
See *pybamm.Symbol.set_id()*.

value
the value returned by the node when evaluated

3.1.6 Array

class pybamm.**Array**(*entries*, *name=None*, *domain=None*, *auxiliary_domains=None*, *entries_string=None*)
node in the expression tree that holds an tensor type variable (e.g. `numpy.array`)

Parameters

- **entries** (*numpy.array*) – the array associated with the node
- **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_domainds** (*dict, optional*) – dictionary of auxiliary domains, defaults to empty dict
- **entries_string** (*str*) – String representing the entries (slow to recalculate when copying)
- ***Extends** –

ndim
returns the number of dimensions of the tensor

new_copy()
See *pybamm.Symbol.new_copy()*.

set_id()
See *pybamm.Symbol.set_id()*.

shape
returns the number of entries along each dimension

pybamm.**linspace**(*start*, *stop*, *num=50*, ***kwargs*)
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*, *y*, ***kwargs*)
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

3.1.7 Matrix

class pybamm.**Matrix**(*entries*, *name=None*, *domain=None*, *auxiliary_domains=None*, *entries_string=None*)
node in the expression tree that holds a matrix type (e.g. `numpy.array`)

Extends: *Array*

3.1.8 Vector

class pybamm.**Vector**(*entries*, *name=None*, *domain=None*, *auxiliary_domains=None*, *entries_string=None*)
node in the expression tree that holds a vector type (e.g. `numpy.array`)

Extends: *Array*

3.1.9 State Vector

class pybamm.**StateVector**(**y_slices*, *name=None*, *domain=None*, *auxiliary_domains=None*, *evaluation_array=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
- **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** –

diff(*variable*)

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*, *name=None*, *domain=None*, *auxiliary_domains=None*, *evaluation_array=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
- **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** –

diff(*variable*)

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

3.1.10 Binary Operators

class pybamm.**BinaryOperator**(*name*, *left*, *right*)

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

Derived classes will specify the particular operator

Extends: *Symbol*

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)

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)
See *pybamm.Symbol.evaluate()*.

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

format(*left*, *right*)
Format children left and right into compatible form

get_children_domains(*ldomain*, *rdomain*)
Combine domains from children in appropriate way

new_copy()
See *pybamm.Symbol.new_copy()*.

class pybamm.**Power**(*left*, *right*)
A node in the expression tree representing a ** power operator

Extends: *BinaryOperator*

class pybamm.**Addition**(*left*, *right*)
A node in the expression tree representing an addition operator

Extends: *BinaryOperator*

class pybamm.**Subtraction**(*left*, *right*)
A node in the expression tree representing a subtraction operator

Extends: *BinaryOperator*

class pybamm.**Multiplication**(*left*, *right*)
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: *BinaryOperator*

class pybamm.**MatrixMultiplication**(*left*, *right*)
A node in the expression tree representing a matrix multiplication operator

Extends: *BinaryOperator*

diff(*variable*)
See *pybamm.Symbol.diff()*.

class pybamm.**Division**(*left*, *right*)
A node in the expression tree representing a division operator

Extends: *BinaryOperator*

class pybamm.**Inner**(*left*, *right*)
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: *BinaryOperator*

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**Heaviside**(*name*, *left*, *right*)
A node in the expression tree representing a heaviside step function.

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: *BinaryOperator*

diff(*variable*)
See *pybamm.Symbol.diff()*.

class pybamm.**EqualHeaviside**(*left*, *right*)
A heaviside function with equality (return 1 when left = right)

class pybamm.**NotEqualHeaviside**(*left*, *right*)
A heaviside function without equality (return 0 when left = right)

class pybamm.**Minimum**(*left*, *right*)
Returns the smaller of two objects

class pybamm.**Maximum**(*left*, *right*)
Returns the smaller of two objects

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

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

pybamm.**source**(*left*, *right*, *boundary=False*)
A convinience 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*) – 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.

3.1.11 Unary Operators

class pybamm.**UnaryOperator**(*name*, *child*, *domain=None*, *auxiliary_domains=None*)
A node in the expression tree representing a unary operator (e.g. '-', grad, div)

Derived classes will specify the particular operator

Extends: *Symbol*

Parameters

- **name** (*str*) – name of the node
- **child** (*Symbol*) – child node

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)
See *pybamm.Symbol.evaluate()*.

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

new_copy()
See *pybamm.Symbol.new_copy()*.

class pybamm.**Negate**(*child*)
A node in the expression tree representing a - negation operator

Extends: *UnaryOperator*

class pybamm.**AbsoluteValue**(*child*)
A node in the expression tree representing an *abs* operator

Extends: *UnaryOperator*

diff(*variable*)
See *pybamm.Symbol.diff()*.

class pybamm.**Sign**(*child*)
A node in the expression tree representing a *sign* operator

Extends: *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.

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

set_id()
See *pybamm.Symbol.set_id()*

class pybamm.**SpatialOperator**(*name*, *child*, *domain=None*, *auxiliary_domains=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*

Extends: *UnaryOperator*

Parameters

- **name** (*str*) – name of the node
- **child** (*Symbol*) – child node

diff(*variable*)
See *pybamm.Symbol.diff()*.

class pybamm.**Gradient**(*child*)
A node in the expression tree representing a grad operator

Extends: *SpatialOperator*

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**Divergence**(*child*)
A node in the expression tree representing a div operator

Extends: *SpatialOperator*

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

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: *SpatialOperator*

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**Gradient_Squared**(*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:** *SpatialOperator*

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

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:** *SpatialOperator*

class pybamm.**Integral**(*child*, *integration_variable*)
A node in the expression tree representing an integral operator

$$I = \int_a^b f(u)\, du,$$

where a and b are the left-hand and right-hand boundaries of the domain respectively, and $u \in$ domain. Can be integration with respect to time or space.

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** (** *SpatialOperator*) –

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

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** (** 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
- **vector_type** (*str, optional*) – Whether to return a row or column vector (default is row)
- ****Extends** (** *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** (** *SpatialOperator*) –

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

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** (** *SpatialOperator*) –

evaluate_for_shape()
See pybamm.Symbol.evaluate_for_shape_using_domain()

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

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.

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** (** *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.

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** (** *BoundaryOperator*) –

class pybamm.**BoundaryGradient**(*child*, *side*)
A node in the expression tree which gets the boundary flux of a variable.

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** (** *BoundaryOperator*) –

pybamm.**grad**(*expression*)
convenience function for creating a *Gradient*

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

Returns the gradient of `expression`

Return type *Gradient*

pybamm.**div**(*expression*)
convenience function for creating a *Divergence*

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

Returns the divergence of `expression`

Return type *Divergence*

pybamm.**laplacian**(*expression*)
convenience function for creating a *Laplacian*

Parameters **expression** (*Symbol*) – the laplacian will be performed on this sub-expression

Returns the laplacian of `expression`

Return type *Laplacian*

pybamm.**grad_squared**(*expression*)
convenience function for creating a *Gradient_Squared*

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

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

Return type *Gradient_Squared*

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*)
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*)
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.**z_average**(*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*)
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.**sign**(*symbol*)
Returns a *Sign* object.

3.1.12 Concatenations

class pybamm.**Concatenation**(**children*, *name=None*, *check_domain=True*, *concat_fun=None*)
A node in the expression tree representing a concatenation of symbols

Extends: *pybamm.Symbol*

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

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)
See *pybamm.Symbol.evaluate()*.

new_copy()
See *pybamm.Symbol.new_copy()*.

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

Extends: *Concatenation*

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

class pybamm.**DomainConcatenation**(*children*, *full_mesh*, *copy_this=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

Extends: *pybamm.Concatenation*

Parameters

- **children** (iterable of *pybamm.Symbol*) – The symbols to concatenate
- **full_mesh** (pybamm.BaseMesh) – 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

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.

Extends: *Concatenation*

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

3.1.13 Broadcasting Operators

class pybamm.**Broadcast**(*child*, *broadcast_domain*, *broadcast_auxiliary_domains=None*, *broadcast_type='full to nodes'*, *name=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
- **broadcast_domain** (*iterable of str*) – Primary domain for broadcast. This will become the domain of the symbol
- **broadcast_auxiliary_domains** (*dict of str*) – Auxiliary domains for broadcast.
- **broadcast_type** (*str, optional*) – Whether to broadcast to the full domain (primary and secondary) or only in the primary direction. Default is "full".
- **name** (*str*) – name of the node
- ****Extends** (** *SpatialOperator*) –

class pybamm.**FullBroadcast**(*child*, *broadcast_domain*, *auxiliary_domains*, *name=None*)

A class for full broadcasts

check_and_set_domains(*child*, *broadcast_type*, *broadcast_domain*, *broadcast_auxiliary_domains*)

See Broadcast.check_and_set_domains()

class pybamm.**PrimaryBroadcast**(*child*, *broadcast_domain*, *name=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*) – 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** (** *SpatialOperator*) –

check_and_set_domains(*child*, *broadcast_type*, *broadcast_domain*, *broadcast_auxiliary_domains*)
See `Broadcast.check_and_set_domains()`

class pybamm.**SecondaryBroadcast**(*child*, *broadcast_domain*, *name=None*)
A node in the expression tree representing a primary 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*) – Primary domain for broadcast. This will become the domain of the symbol
- **name** (*str*) – name of the node
- ****Extends** (** *SpatialOperator*) –

check_and_set_domains(*child*, *broadcast_type*, *broadcast_domain*, *broadcast_auxiliary_domains*)
See `Broadcast.check_and_set_domains()`

class pybamm.**FullBroadcastToEdges**(*child*, *broadcast_domain*, *auxiliary_domains*, *name=None*)
A full broadcast onto the edges of a domain (edges of primary dimension, nodes of other dimensions)

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**PrimaryBroadcastToEdges**(*child*, *broadcast_domain*, *name=None*)
A primary broadcast onto the edges of the domain

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

class pybamm.**SecondaryBroadcastToEdges**(*child*, *broadcast_domain*, *name=None*)
A secondary broadcast onto the edges of a domain

evaluates_on_edges()
See *pybamm.Symbol.evaluates_on_edges()*.

pybamm.**ones_like**(**symbols*)
Create a symbol with the same shape as the input symbol and with constant value '1', using *FullBroadcast*.

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

3.1.14 Functions

class pybamm.**Function**(*function*, **children*, *name=None*, *derivative='autograd'*, *differentiated_function=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
- **derivative** (*str, optional*) – Which derivative to use when differentiating ("autograd" or "derivative"). Default is "autograd".
- **differentiated_function** (*method, optional*) – The function which was differentiated to obtain this one. Default is None.
- ****Extends** (** *pybamm.Symbol*) –

diff(*variable*)

See *pybamm.Symbol.diff()*.

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)

See *pybamm.Symbol.evaluate()*.

evaluates_on_edges()

See *pybamm.Symbol.evaluates_on_edges()*.

get_children_domains(*children_list*)

Obtains the unique domain of the children. If the children have different domains then raise an error

new_copy()

See *pybamm.Symbol.new_copy()*.

class pybamm.**SpecificFunction**(*function*, *child*)

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

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

Cosine function

pybamm.**cos**(*child*)

Returns cosine function of child.

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

Hyberbolic cosine function

pybamm.**cosh**(*child*)

Returns hyperbolic cosine function of child.

class pybamm.**Exponential**(*child*)

Exponential function

pybamm.**exp**(*child*)

Returns exponential function of child.

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

pybamm.**log**(*child*, *base='e'*)
Returns logarithmic function of child (any base, default 'e').

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

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

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

pybamm.**sin**(*child*)
Returns sine function of child.

class pybamm.**Sinh**(*child*)
Hyperbolic sine function

pybamm.**sinh**(*child*)
Returns hyperbolic sine function of child.

3.1.15 Input Parameter

class pybamm.**InputParameter**(*name*, *domain=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)

new_copy()
See *pybamm.Symbol.new_copy()*.

set_expected_size(*size*)
Specify the size that the input parameter should be

3.1.16 Interpolant

class pybamm.**Interpolant**(*data*, *child*, *name=None*, *interpolator='cubic spline'*, *extrapolate=True*, *entries_string=None*)
Interpolate data in 1D.

Parameters

- **data** (numpy.ndarray) – Numpy array of data to use for interpolation. Must have exactly two columns (x and y data)
- **child** (*pybamm.Symbol*) – Node to use when evaluating the interpolant
- **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 ("pchip" or "cubic spline"). Note that whichever interpolator is used must be differentiable (for `Interpolator._diff`). Default is "cubic spline". Note that "pchip" may give slow results.
- **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.
- ****Extends**** (*pybamm.Function*) –

set_id()
See *pybamm.Symbol.set_id()*.

3.1.17 Operations on expression trees

Classes and functions that operate on the expression tree

Simplify

class pybamm.**Simplification**(*simplified_symbols=None*)

simplify(*symbol*, *clear_domains=True*)
This function recurses down the tree, applying any simplifications defined in classes derived from pybamm.Symbol. E.g. any expression multiplied by a pybamm.Scalar(0) will be simplified to a pybamm.Scalar(0). If a symbol has already been simplified, the stored value is returned.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to simplify
- **clear_domains** (*bool*) – Whether to remove a symbol's domain when simplifying. Default is True.

Returns

- *pybamm.Symbol*
- *Simplified symbol*

pybamm.**simplify_if_constant**(*symbol*, *keep_domains=False*)
Utility function to simplify an expression tree if it evalutes to a constant scalar, vector or matrix

pybamm.**simplify_addition_subtraction**(*myclass*, *left*, *right*)
if children are associative (addition, subtraction, etc) then try to find groups of constant children (that produce a value) and simplify them to a single term

The purpose of this function is to simplify expressions like (1 + (1 + p)), which should be simplified to (2 + p). The former expression consists of an Addition, with a left child of Scalar type, and a right child of another Addition containing a Scalar and a Parameter. For this case, this function will first flatten the expression to a list of the bottom level children (i.e. [Scalar(1), Scalar(2), Parameter(p)]), and their operators (i.e. [None, Addition, Addition]), and then combine all the constant children (i.e. Scalar(1) and Scalar(1)) to a single child (i.e. Scalar(2))

Note that this function will flatten the expression tree until a symbol is found that is not either an Addition or a Subtraction, so this function would simplify (3 - (2 + a*b*c)) to (1 + a*b*c)

This function is useful if different children expressions contain non-constant terms that prevent them from being simplified, so for example (1 + a) + (b - 2) - (6 + c) will be simplified to (-7 + a + b - c)

Parameters

- **myclass** (*class*) – the binary operator class (pybamm.Addition or pybamm.Subtraction) operating on children left and right
- **left** (*derived from pybamm.Symbol*) – the left child of the binary operator
- **right** (*derived from pybamm.Symbol*) – the right child of the binary operator

pybamm.**simplify_multiplication_division**(*myclass*, *left*, *right*)

if children are associative (multiply, division, etc) then try to find groups of constant children (that produce a value) and simplify them

The purpose of this function is to simplify expressions of the type (1 * c / 2), which should simplify to (0.5 * c). The former expression consists of a Division, with a left child of a Multiplication containing a Scalar and a Parameter, and a right child consisting of a Scalar. For this case, this function will first flatten the expression to a list of the bottom level children on the numerator (i.e. [Scalar(1), Parameter(c)]) and their operators (i.e. [None, Multiplication]), as well as those children on the denominator (i.e. [Scalar(2)]. After this, all the constant children on the numerator and denominator (i.e. Scalar(1) and Scalar(2)) will be combined appropriately, in this case to Scalar(0.5), and combined with the nonconstant children (i.e. Parameter(c))

Note that this function will flatten the expression tree until a symbol is found that is not either an Multiplication, Division or MatrixMultiplication, so this function would simplify (3*(1 + d)*2) to (6 * (1 + d))

As well as Multiplication and Division, this function can handle MatrixMultiplication. If any MatrixMultiplications are found on the numerator/denominator, no reordering of children is done to find groups of constant children. In this case only neighbouring constant children on the numerator are simplified

Parameters

- **myclass** (*class*) – the binary operator class (pybamm.Addition or pybamm.Subtraction) operating on children left and right
- **left** (*derived from pybamm.Symbol*) – the left child of the binary operator
- **right** (*derived from pybamm.Symbol*) – the right child of the binary operator

EvaluatorPython

class pybamm.**EvaluatorPython**(*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

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)

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

Jacobian

class pybamm.**Jacobian**(*known_jacs=None*, *clear_domain=True*)

Helper class to calculate the jacobian of an expression.

Parameters

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

jac(*symbol*, *variable*)
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*, *t*, *y*, *y_dot*, *inputs*)
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

Symbol Unpacker

class pybamm.**SymbolUnpacker**(*classes_to_find*, *unpacked_symbols=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** (dict {variable ids -> *pybamm.Symbol*}) – cached unpacked equations

unpack_list_of_symbols(*list_of_symbols*)
Unpack a list of symbols. See SymbolUnpacker.unpack()

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

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

Return type list of *pybamm.Symbol*

unpack_symbol(*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*

3.2 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.

3.2.1 Base Models

Base Model

class pybamm.**BaseModel**(*name='Unnamed model'*)
Base model class for other models to extend.

name
A string giving the name of the model

Type str

options
A dictionary of options to be passed to the model

Type dict

rhs
A dictionary that maps expressions (variables) to expressions that represent the rhs

Type dict

algebraic
A dictionary that maps expressions (variables) to expressions that represent the algebraic equations. The algebraic expressions are assumed to equate to zero. Note that all the variables in the model must exist in the keys of *rhs* or *algebraic*.

Type dict

initial_conditions
A dictionary that maps expressions (variables) to expressions that represent the initial conditions for the state variables y. The initial conditions for algebraic variables are provided as initial guesses to a root finding algorithm that calculates consistent initial conditions.

Type dict

boundary_conditions
A dictionary that maps expressions (variables) to expressions that represent the boundary conditions

Type dict

variables
A dictionary that maps strings to expressions that represent the useful variables

Type dict

events
A list of events. Each event can either cause the solver to terminate (e.g. concentration goes negative), or be used to inform the solver of the existance of a discontinuity (e.g. discontinuity in the input current)

Type list of *pybamm.Event*

concatenated_rhs
After discretisation, contains the expressions representing the rhs equations concatenated into a single expression

Type *pybamm.Concatenation*

concatenated_algebraic
After discretisation, contains the expressions representing the algebraic equations concatenated into a single expression

Type *pybamm.Concatenation*

concatenated_initial_conditions
After discretisation, contains the vector of initial conditions

Type `numpy.array`

mass_matrix
After discretisation, contains the mass matrix for the model. This is computed automatically

Type *pybamm.Matrix*

mass_matrix_inv
After discretisation, contains the inverse mass matrix for the differential (rhs) part of model. This is computed automatically

Type *pybamm.Matrix*

jacobian
Contains the Jacobian for the model. If model.use_jacobian is True, the Jacobian is computed automatically during solver set up

Type *pybamm.Concatenation*

jacobian_rhs
Contains the Jacobian for the part of the model which contains time derivatives. If model.use_jacobian is True, the Jacobian is computed automatically during solver set up

Type *pybamm.Concatenation*

jacobian_algebraic
Contains the Jacobian for the algebraic part of the model. This may be used by the solver when calculating consistent initial conditions. If model.use_jacobian is True, the Jacobian is computed automatically during solver set up

Type *pybamm.Concatenation*

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

Type bool

use_simplify
Whether to simplify the expression tress representing the rhs and algebraic equations, Jacobain (if using) and events, before solving the model (default is True)

Type bool

convert_to_format
Whether to convert the expression trees representing the rhs and algebraic equations, Jacobain (if using) and events into a different format:

- None: keep PyBaMM expression tree structure.
- "python": convert into pure python code that will calculate the result of calling *evaluate(t, y)* on the given expression treeself.
- "casadi": convert into CasADi expression tree, which then uses CasADi's algorithm to calculate the Jacobian.

Default is "casadi".

Type str

check_algebraic_equations(*post_discretisation*)
Check that the algebraic equations are well-posed. Before discretisation, each algebraic equation key must appear in the equation After discretisation, there must be at least one StateVector in each algebraic equation

check_default_variables_dictionaries()
Chec that the right variables are provided.

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

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

default_solver
Return default solver based on whether model is ODE model or DAE model

info(*symbol_name*)
Provides helpful summary information for a symbol.

Parameters **parameter_name** (*str*) –

input_parameters
Returns all the input parameters in the model

new_copy(*options=None*)
Create an empty copy with identical options, or new options if specified

parameters
Returns all the parameters in the model

timescale
Timescale of model, to be used for non-dimensionalising time when solving

update(**submodels*)
Update model to add new physics from submodels

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

Base Battery Model

class pybamm.**BaseBatteryModel**(*options=None*, *name='Unnamed battery model'*)
Base model class with some default settings and required variables

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.

- **"dimensionality"** [int, optional] Sets the dimension of the current collector problem. Can be 0 (default), 1 or 2.
- **"surface form"** [bool or str, optional] Whether to use the surface formulation of the problem. Can be False (default), "differential" or "algebraic".
- **"convection"** [bool or str, optional] Whether to include the effects of convection in the model. Can be False (default), "differential" or "algebraic". Must be 'False' for lithium-ion models.
- **"side reactions"** [list, optional] Contains a list of any side reactions to include. Default is []. If this list is not empty (i.e. side reactions are included in the model), then "surface form" cannot be 'False'.
- **"interfacial surface area"** [str, optional] Sets the model for the interfacial surface area. Can be "constant" (default) or "varying". Not currently implemented in any of the models.
- **"current collector"** [str, optional] Sets the current collector model to use. Can be "uniform" (default), "potential pair" or "potential pair quite conductive".
- **"particle"** [str, optional] Sets the submodel to use to describe behaviour within the particle. Can be "Fickian diffusion" (default) or "fast diffusion".
- **"thermal"** [str, optional] Sets the thermal model to use. Can be "isothermal" (default), "lumped", "x-lumped", or "x-full".
- **"external submodels"** [list] A list of the submodels that you would like to supply an external variable for instead of solving in PyBaMM. The entries of the lists are strings that correspond to the submodel names in the keys of *self.submodels*.
- **"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": *pybamm.sei.ReactionLimited*
  - "solvent-diffusion limited": *pybamm.sei.SolventDiffusionLimited*
  - "electron-migration limited": *pybamm.sei.ElectronMigrationLimited*
  - "interstitial-diffusion limited": *pybamm.sei.InterstitialDiffusionLimited*
- **"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 sin**

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

Type dict

Extends: *pybamm.BaseModel*

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*

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

Event

class pybamm.**Event**(*name*, *expression*, *event_type=<EventType.TERMINATION: 0>*)

Defines an event for use within a pybamm model

name

A string giving the name of the event

Type str

event_type

An enum defining the type of event

Type *pybamm.EventType*

expression

An expression that defines when the event occurs

Type *pybamm.Symbol*

evaluate(*t=None*, *y=None*, *y_dot=None*, *inputs=None*, *known_evals=None*)
Acts as a drop-in replacement for *pybamm.Symbol.evaluate()*

class pybamm.**EventType**
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.

3.2.2 Lithium-ion Models

Base Lithium-ion Model

class pybamm.lithium_ion.**BaseModel**(*options=None*, *name='Unnamed lithium-ion model'*)
Overwrites default parameters from Base Model with default parameters for lithium-ion models

Extends: *pybamm.BaseBatteryModel*

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[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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).

References

Extends: *pybamm.lithium_ion.BaseModel*

class pybamm.lithium_ion.**BasicSPM**(*name='Single Particle Model'*)
Single Particle Model (SPM) model of a lithium-ion battery, from[2].

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.

[1] SG Marquis, V Sulzer, R Timms, CP Please and SJ Chapman. "An asymptotic derivation of a single particle model with electrolyte". Journal of The Electrochemical Society, 166(15):A3693–A3706, 2019

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

References

Extends: *pybamm.lithium_ion.BaseModel*

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[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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).

References

Extends: *pybamm.lithium_ion.BaseModel*

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[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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).

References

Extends: *pybamm.lithium_ion.BaseModel*

class pybamm.lithium_ion.**BasicDFN**(*name='Doyle-Fuller-Newman model'*)

Doyle-Fuller-Newman (DFN) model of a lithium-ion battery, from[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.

[1] SG Marquis, V Sulzer, R Timms, CP Please and SJ Chapman. "An asymptotic derivation of a single particle model with electrolyte". Journal of The Electrochemical Society, 166(15):A3693–A3706, 2019

[1] SG Marquis, V Sulzer, R Timms, CP Please and SJ Chapman. "An asymptotic derivation of a single particle model with electrolyte". Journal of The Electrochemical Society, 166(15):A3693–A3706, 2019

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

References

Extends: *pybamm.lithium_ion.BaseModel*

3.2.3 Lead Acid Models

Base Model

class pybamm.lead_acid.**BaseModel**(*options=None*, *name='Unnamed lead-acid model'*)
Overwrites default parameters from Base Model with default parameters for lead-acid models

Extends: *pybamm.BaseBatteryModel*

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[1].

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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).

References

Extends: *pybamm.lead_acid.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

Higher-Order Models

class pybamm.lead_acid.**BaseHigherOrderModel**(*options=None*, *name='Composite model'*, *build=True*)
Base model for higher-order models for lead-acid, from[1]. Uses leading-order model from *pybamm.lead_acid.LOQS*

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the model.

[1] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part II. Asymptotic analysis. Journal of The Electrochemical Society 166.12 (2019), A2372–A2382.
[1] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part II. Asymptotic analysis. Journal of The Electrochemical Society 166.12 (2019), A2372–A2382.

- **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).

References

Extends: *pybamm.lead_acid.BaseModel*

set_full_convection_submodel()
Update convection submodel, now that we have the spatially heterogeneous interfacial current densities

set_full_interface_submodel()
Set full interface submodel, to get spatially heterogeneous interfacial current densities

set_full_porosity_submodel()
Update porosity submodel, now that we have the spatially heterogeneous interfacial current densities

class pybamm.lead_acid.**FOQS**(*options=None*, *name='FOQS model'*, *build=True*)
First-order quasi-static model for lead-acid, from[1]. Uses leading-order model from *pybamm.lead_acid.LOQS*

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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.lead_acid.BaseHigherOrderModel*) –

set_full_porosity_submodel()
Update porosity submodel, now that we have the spatially heterogeneous interfacial current densities

class pybamm.lead_acid.**Composite**(*options=None*, *name='Composite model'*, *build=True*)
Composite model for lead-acid, from[1]. Uses leading-order model from *pybamm.lead_acid.LOQS*

Extends: *pybamm.lead_acid.BaseHigherOrderModel*

set_full_porosity_submodel()
Update porosity submodel, now that we have the spatially heterogeneous interfacial current densities

class pybamm.lead_acid.**CompositeExtended**(*options=None*, *name='Extended composite model (distributed)'*, *build=True*)
Extended composite model for lead-acid. Uses leading-order model from *pybamm.lead_acid.LOQS*

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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.lead_acid.BaseHigherOrderModel*

Full Model

class pybamm.lead_acid.**Full**(*options=None*, *name='Full model'*, *build=True*)
Porous electrode model for lead-acid, from[1], based on the Newman-Tiedemann model.

Parameters

- **options** (*dict, optional*) – A dictionary of options to be passed to the model.
- **name** (*str, optional*) – The name of the 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).

References

Extends: *pybamm.lead_acid.BaseModel*

class pybamm.lead_acid.**BasicFull**(*name='Basic full model'*)
Porous electrode model for lead-acid, from[2].

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.

References

Extends: *pybamm.lead_acid.BaseModel*

3.2.4 Submodels

Base Submodel

class pybamm.**BaseSubModel**(*param*, *domain=None*, *name='Unnamed submodel'*, *external=False*)
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

param
The model parameter symbols

Type parameter class

rhs
A dictionary that maps expressions (variables) to expressions that represent the rhs

Type dict

[1] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part II. Asymptotic analysis. Journal of The Electrochemical Society 166.12 (2019), A2372–A2382.
[2] V Sulzer, SJ Chapman, CP Please, DA Howey, and CW Monroe. Faster lead-acid battery simulations from porous-electrode theory: Part II. Asymptotic analysis. Journal of The Electrochemical Society 166.12 (2019), A2372–A2382..

algebraic
A dictionary that maps expressions (variables) to expressions that represent the algebraic equations. The algebraic expressions are assumed to equate to zero. Note that all the variables in the model must exist in the keys of *rhs* or *algebraic*.

Type dict

initial_conditions
A dictionary that maps expressions (variables) to expressions that represent the initial conditions for the state variables y. The initial conditions for algebraic variables are provided as initial guesses to a root finding algorithm that calculates consistent initial conditions.

Type dict

boundary_conditions
A dictionary that maps expressions (variables) to expressions that represent the boundary conditions

Type dict

variables
A dictionary that maps strings to expressions that represent the useful variables

Type dict

events
A list of events. Each event can either cause the solver to terminate (e.g. concentration goes negative), or be used to inform the solver of the existance of a discontinuity (e.g. discontinuity in the input current)

Type list

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_external_variables()
A public method that returns the variables in a submodel which are supplied by an external source.

Returns A list of the external variables in the model.

Return type list

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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

set_events (*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 a 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 a 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.

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.BaseSubModel*

Composite Potential Pair models

class pybamm.current_collector.**BaseCompositePotentialPair** (*param*)
Composite potential pair model for the current collectors. This is identical to the BasePotentialPair model, except the name of the fundamental variables are changed to avoid clashes with leading order.

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.current_collector.BasePotentialPair*

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

class pybamm.current_collector.**CompositePotentialPair2plus1D**(*param*)

class pybamm.current_collector.**CompositePotentialPair1plus1D**(*param*)

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[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.

References

Extends: *pybamm.BaseModel*

default_solver
Return default solver based on whether model is ODE model or DAE model

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.

Extends: *pybamm.BaseModel*

default_solver
Return default solver based on whether model is ODE model or DAE model

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.

[1] R Timms, SG Marquis, V Sulzer, CP Please and SJ Chapman. "Asymptotic Reduction of a Lithium-ion Pouch Cell Model". Submitted, 2020.

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.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

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[1] and[2].

Parameters **param** (*parameter class*) – The parameters to use for this submodel

References

Extends: *pybamm.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

[1] R Timms, SG Marquis, V Sulzer, CP Please and SJ Chapman. "Asymptotic Reduction of a Lithium-ion Pouch Cell Model". Submitted, 2020.

[2] SG Marquis, R Timms, V Sulzer, CP Please and SJ Chapman. "A Suite of Reduced-Order Models of a Single-Layer Lithium-ion Pouch Cell". In preparation, 2020.

self.initial_conditions. Unless overwritten by a submodel, the default behaviour of 'pass' is used a 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

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 a 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.

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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Quite Conductive Potential Pair models

class pybamm.current_collector.**BaseQuiteConductivePotentialPair**(*param*)
A submodel for Ohm's law plus conservation of current in the current collectors, in the limit of quite conductive electrodes.

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

class pybamm.current_collector.**QuiteConductivePotentialPair1plus1D**(*param*)

class pybamm.current_collector.**QuiteConductivePotentialPair2plus1D**(*param*)

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*)
Base class for convection submodels.

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.BaseSubModel*

Through-cell Convection

Base Model

class pybamm.convection.through_cell.**BaseThroughCellModel**(*param*)
Base class for convection submodels in the through-cell direction.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- ****Extends** (** *pybamm.convection.BaseModel*) –

No Convection

class pybamm.convection.through_cell.**NoConvection**(*param*)
A submodel for case where there is no convection.

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.convection.through_cell.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.convection.through_cell.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.convection.through_cell.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 a 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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Transverse Convection

Base Model

class pybamm.convection.transverse.**BaseTransverseModel**(*param*)
Base class for convection submodels in transverse directions.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- ****Extends** (** *pybamm.convection.BaseModel*) –

No Transverse Convection

class pybamm.convection.transverse.**NoConvection**(*param*)
Submodel for no convection in transverse directions

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- ****Extends** (** pybamm.convection.through_cell.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.convection.through_cell.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.convection.through_cell.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 a 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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Electrode

Electrode Base Model

class pybamm.electrode.**BaseElectrode**(*param*, *domain*, *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'
- **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.BaseSubModel*) –

Ohmic

Base Model

class pybamm.electrode.ohm.**BaseModel**(*param*, *domain*, *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'

Extends: *pybamm.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Leading Order Model

class pybamm.electrode.ohm.**LeadingOrder**(*param*, *domain*, *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'
- **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.electrode.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Composite Model

class pybamm.electrode.ohm.**Composite**(*param*, *domain*)

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'
- ****Extends** (** pybamm.BaseOhm) –

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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Full Model

class pybamm.electrode.ohm.**Full**(*param*, *domain*)

Full model of electrode employing Ohm's law.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – Either 'Negative' or 'Positive'

Extends: *pybamm.electrode.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 a 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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Surface Form

class pybamm.electrode.ohm.**SurfaceForm**(*param*, *domain*)

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'

Extends: *pybamm.electrode.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

Electrolyte Conductivity

Base Electrolyte Conductivity Submodel

class pybamm.electrolyte_conductivity.**BaseElectrolyteConductivity**(*param*, *domain=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
- **reactions** (*dict, optional*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Leading Order Model

class pybamm.electrolyte_conductivity.**LeadingOrder**(*param*, *domain=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
- **reactions** (*dict, optional*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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*, *higher_order_terms='composite'*)
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
- **higher_order_terms** (*str*) – What kind of higher-order terms to use ('composite' or 'first-order')
- **domain** (*str, optional*) – The domain in which the model holds
- ****Extends** (** *pybamm.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

Full Model

class pybamm.electrolyte_conductivity.**Full**(*param*)
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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.electrolyte_conductivity.BaseElectrolyteConductivity*) –

check_algebraic_equations(*post_discretisation*)
Check that the algebraic equations are well-posed. Before discretisation, each algebraic equation key must appear in the equation After discretisation, there must be at least one StateVector in each algebraic equation

check_default_variables_dictionaries()
Chec that the right variables are provided.

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

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

default_solver
Return default solver based on whether model is ODE model or DAE 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_external_variables()
A public method that returns the variables in a submodel which are supplied by an external source.

Returns A list of the external variables in the model.

Return type list

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

info(*symbol_name*)
Provides helpful summary information for a symbol.

Parameters **parameter_name** (*str*) –

input_parameters
Returns all the input parameters in the model

new_copy(*options=None*)
Create an empty copy with identical options, or new options if specified

parameters
Returns all the parameters in the model

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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

set_events(*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 a 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 a 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.

timescale
Timescale of model, to be used for non-dimensionalising time when solving

update(**submodels*)
Update model to add new physics from submodels

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

Surface Form

Full Model

class pybamm.electrolyte_conductivity.surface_potential_form.**FullDifferential**(*param*, *domain*)
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

Extends: pybamm.electrolyte_conductivity.surface_potential_form.BaseFull

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*)
Full model for conservation of charge in the electrolyte employing the Stefan-Maxwell constitutive equations. (Full refers to unreduced by asymptotic methods)

Parameters **param** – The parameters to use for this submodel

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*)
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

Extends: 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*)
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

Extends: 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.

Electrolyte Diffusion

Base Electrolyte Diffusion Submodel

class pybamm.electrolyte_diffusion.**BaseElectrolyteDiffusion**(*param*)
Base class for conservation of mass in the electrolyte.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **reactions** (*dict, optional*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.BaseSubModel*) –

set_events(*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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Constant Concentration

class pybamm.electrolyte_diffusion.**ConstantConcentration**(*param*)
Class for constant concentration of electrolyte

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.electrolyte_diffusion.BaseElectrolyteDiffusion*

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 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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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 a 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.

Composite Model

class pybamm.electrolyte_diffusion.**Composite**(*param*, *extended=False*)
Class for conservation of mass in the electrolyte employing the Stefan-Maxwell constitutive equations. (Composite refers to composite model by asymptotic methods)

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **reactions** (*dict*) – Dictionary of reaction terms
- **extended** (*bool*) – Whether to include feedback from the first-order terms
- ****Extends** (** *pybamm.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 a 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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

set_rhs(*variables*)
Composite reaction-diffusion with source terms from leading order

Full Model

class pybamm.electrolyte_diffusion.**Full**(*param*)
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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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 a 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 a 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.

Current control external circuit

class pybamm.external_circuit.**CurrentControl**(*param*)
External circuit with current control.

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

Function control external circuit

class pybamm.external_circuit.**FunctionControl**(*param*, *external_circuit_function*)
External circuit with an arbitrary function.

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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

class pybamm.external_circuit.**VoltageFunctionControl**(*param*)
External circuit with voltage control, implemented as an extra algebraic equation.

class pybamm.external_circuit.**PowerFunctionControl**(*param*)
External circuit with power control.

Interface

Interface Base Model

class pybamm.interface.**BaseInterface**(*param*, *domain*, *reaction*)
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
- ****Extends** (** *pybamm.BaseSubModel*) –

Kinetics

class pybamm.interface.**BaseKinetics**(*param*, *domain*, *reaction*, *options=None*)
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. In this case "sei film resistance" is the important option. See *pybamm.BaseBatteryModel*
- ****Extends** (** *pybamm.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

class pybamm.interface.**ButlerVolmer**(*param*, *domain*, *reaction*, *options=None*)
Base submodel which implements the forward Butler-Volmer equation:

$$j = 2 * j_0(c) * \sinh((ne/(2 * (1 + \Theta T)) * \eta_r(c))$$

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. In this case "sei film resistance" is the important option. See *pybamm.BaseBatteryModel*
- ****Extends** (** pybamm.interface.kinetics.BaseKinetics) –

class pybamm.interface.**NoReaction**(*param*, *domain*, *reaction*)
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
- ****Extends** (** pybamm.interface.kinetics.BaseKinetics) –

class pybamm.interface.**ForwardTafel**(*param*, *domain*, *reaction*, *options=None*)

Base submodel which implements the forward Tafel equation:

$$j = j_0(c) * \exp((ne/(2 * (1 + \Theta T)) * \eta_r(c))$$

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. In this case "sei film resistance" is the important option. See *pybamm.BaseBatteryModel*
- ****Extends** (** pybamm.interface.kinetics.BaseKinetics) –

class pybamm.interface.**BackwardTafel**(*param*, *domain*, *reaction*)

Base submodel which implements the backward Tafel equation:

$$j = -j_0(c) * \exp(-\eta_r(c))$$

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.

Extends: pybamm.interface.kinetics.BaseKinetics

Inverse Kinetics

class pybamm.interface.inverse_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. Default is None, in which case j.domain is used.
- **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.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

First-order Kinetics

class pybamm.interface.**FirstOrderKinetics**(*param*, *domain*, *leading_order_model*)
First-order kinetics

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **leading_order_model** (pybamm.interface.kinetics.BaseKinetics) – The leading-order model with respect to which this is first-order
- ****Extends** (** *pybamm.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

class pybamm.interface.**InverseFirstOrderKinetics**(*param*, *domain*, *leading_order_models*)
Base inverse first-order kinetics. This class needs to consider *all* of the leading-order submodels simultaneously in order to find the first-order correction to the potentials

Parameters

- **param** – model parameters
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- **leading_order_models** (pybamm.interface.kinetics.BaseKinetics) – The leading-order models with respect to which this is first-order
- ****Extends** (** *pybamm.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

Diffusion-limited

class pybamm.interface.**DiffusionLimited**(*param*, *domain*, *reaction*, *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
- **order** (*str*) – The order of the model ("leading" or "full")
- ****Extends** (** *pybamm.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

SEI models

class pybamm.sei.**BaseModel**(*param*, *domain*)
Base class for SEI models.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain to implement the model, either: 'Negative' or 'Positive'.
- ****Extends** (** *pybamm.interface.BaseInterface*) –

class pybamm.sei.**ConstantSEI**(*param*, *domain*)
Base 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
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

class pybamm.sei.**ElectronMigrationLimited**(*param*, *domain*)
Base class for electron-migration limited SEI growth.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

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 a 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.sei.**InterstitialDiffusionLimited**(*param*, *domain*)
Base class for interstitial-diffusion limited SEI growth.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

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 a 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.sei.**NoSEI**(*param*, *domain*)
Base class for no SEI.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

class pybamm.sei.**ReactionLimited**(*param*, *domain*)
Base class for reaction limited SEI growth.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

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 a 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.sei.**SolventDiffusionLimited**(*param*, *domain*)
Base class for solvent-diffusion limited SEI growth.

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'
- ****Extends** (** *pybamm.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

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 a 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.

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
- **reactions** (*dict, optional*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.BaseSubModel*) –

Composite Model

class pybamm.oxygen_diffusion.**Composite**(*param*, *extended=False*)

Class for conservation of mass of oxygen. (Composite refers to composite expansion in 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
- **reactions** (*dict*) – Dictionary of reaction terms
- **extended** (*bool*) – Whether to include feedback from the first-order terms
- ****Extends** (** *pybamm.oxygen_diffusion.Full*) –

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_rhs(*variables*)

Composite reaction-diffusion with source terms from leading order

First-Order Model

class pybamm.oxygen_diffusion.**FirstOrder**(*param*)

Class for conservation of mass of oxygen. (First-order refers to first-order expansion in 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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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

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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** *pybamm.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 a 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 a 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
- **reactions** (*dict*) – Dictionary of reaction terms
- ****Extends** (** pybamm.oxgen_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 a 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.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*)
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'

Extends: *pybamm.BaseSubModel*

set_events(*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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Fickian Single Particle

class pybamm.particle.**FickianSingleParticle**(*param*, *domain*)
Base class for molar conservation in a single x-averaged particle which employs 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'

Extends: *pybamm.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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

set_initial_conditions(*variables*)
For single particle models, initial conditions can't depend on x so we arbitrarily set the initial values of the single particles to be given by the values at x=0 in the negative electrode and x=1 in the positive electrode. Typically, supplied initial conditions are uniform x.

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.

Fickian Many Particles

class pybamm.particle.**FickianManyParticles**(*param*, *domain*)
Base class for molar conservation in many particles which employs 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'

Extends: *pybamm.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 a 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 a 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.

Fast Single Particle

class pybamm.particle.**FastSingleParticle**(*param*, *domain*)
Base class for molar conservation in a single x-averaged particle with uniform concentration in r (i.e. infinitely fast diffusion within particles).

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'

Extends: *pybamm.particle.BaseParticle*

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*)
For single particle models, initial conditions can't depend on x so we arbitrarily evaluate them at x=0 in the negative electrode and x=1 in the positive electrode (they will usually be constant)

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.

Fast Many Particles

class pybamm.particle.**FastManyParticles** (*param*, *domain*)
Base class for molar conservation in many particles with uniform concentration in r (i.e. infinitely fast diffusion within particles).

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **domain** (*str*) – The domain of the model either 'Negative' or 'Positive'

Extends: *pybamm.particle.BaseParticle*

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 a 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.

Porosity

Base Model

class pybamm.porosity.**BaseModel** (*param*)
Base class for porosity

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.BaseSubModel*

set_events (*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 a implemented in *pybamm.BaseSubModel*.

Parameters **variables** (*dict*) – The variables in the whole model.

Constant Porosity

class pybamm.porosity.**Constant**(*param*)
Submodel for constant porosity

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.porosity.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

Leading-Order Model

class pybamm.porosity.**LeadingOrder**(*param*)
Leading-order model for reaction-driven porosity changes

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.porosity.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 a 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.porosity.**Full**(*param*)
Full model for reaction-driven porosity changes

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.porosity.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 a 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*, *cc_dimension=0*)
Base class for thermal effects

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **cc_dimension** (*int, optional*) – The dimension of the current collectors. Can be 0 (default), 1 or 2.
- ****Extends** (** *pybamm.BaseSubModel*) –

Isothermal Model

class pybamm.thermal.isothermal.**Isothermal**(*param*)
Class for isothermal submodel.

Parameters **param** (*parameter class*) – The parameters to use for this submodel

Extends: *pybamm.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*, *cc_dimension=0*, *geometry='arbitrary'*)
Class for lumped thermal submodel

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **cc_dimension** (*int, optional*) – The dimension of the current collectors. Can be 0 (default), 1 or 2.

- **geometry** (*string, optional*) – The geometry for the lumped thermal submodel. Can be "arbitrary" (default) or pouch.
- ****Extends** (** *pybamm.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 a 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.

One Dimensional Model

class pybamm.thermal.x_full.**OneDimensionalX**(*param*)

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).

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- ****Extends** (** *pybamm.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 a 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 a 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.

Pouch Cell

Thermal Model for "1+1D" Pouch Cell

class pybamm.thermal.pouch_cell.**CurrentCollector1D**(*param*)
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[1] and[2].

Parameters **param** (*parameter class*) – The parameters to use for this submodel

References

Extends: *pybamm.thermal.BaseThermal*

get_coupled_variables(*variables*)
A public method that creates and returns the variables in a submodel which require variables in other

[1] R Timms, SG Marquis, V Sulzer, CP Please and SJ Chapman. "Asymptotic Reduction of a Lithium-ion Pouch Cell Model". In preparation, 2020.

[2] SG Marquis, R Timms, V Sulzer, CP Please and SJ Chapman. "A Suite of Reduced-Order Models of a Single-Layer Lithium-ion Pouch Cell". In preparation, 2020.

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 a 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 a 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 Model for "2+1D" Pouch Cell

class pybamm.thermal.pouch_cell.**CurrentCollector2D**(*param*)
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[1] and[2].

Parameters **param** (*parameter class*) – The parameters to use for this submodel

References

Extends: *pybamm.thermal.BaseThermal*

[1] R Timms, SG Marquis, V Sulzer, CP Please and SJ Chapman. "Asymptotic Reduction of a Lithium-ion Pouch Cell Model". In preparation, 2020.
[2] SG Marquis, R Timms, V Sulzer, CP Please and SJ Chapman. "A Suite of Reduced-Order Models of a Single-Layer Lithium-ion Pouch Cell". In preparation, 2020.

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 a 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 a 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.

Tortuosity

Base Model

class pybamm.tortuosity.**BaseModel**(*param*, *phase*)
Base class for tortuosity

Parameters

- **param** (*parameter class*) – The parameters to use for this submodel
- **phase** (*str*) – The material for the model ('electrolyte' or 'electrode').
- ****Extends** (** *pybamm.BaseSubModel*) –

Bruggeman Model

class pybamm.tortuosity.**Bruggeman**(*param*, *phase*, *set_leading_order=False*)
Submodel for Bruggeman tortuosity

Extends: *pybamm.tortuosity.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

3.3 Parameters

3.3.1 Base Parameter Values

class pybamm.**ParameterValues**(*values=None*, *chemistry=None*)
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 a file of parameters If string, gets passed to read_parameters_csv to read a file.
- **chemistry** (*dict*) – Dict of strings for default chemistries. Must be of the form: {"base chemistry": base_chemistry, "cell": cell_properties_authorYear, "anode": anode_chemistry_authorYear, "separator": separator_chemistry_authorYear, "cathode": cathode_chemistry_authorYear, "electrolyte": electrolyte_chemistry_authorYear, "experiment": experimental_conditions_authorYear}. Then the anode chemistry is loaded from the file inputs/parameters/base_chemistry/anodes/anode_chemistry_authorYear, etc. Parameters in "cell" should include geometry and current collector properties. Parameters in "experiment" should include parameters relating to experimental conditions, such as initial conditions and currents.

Examples

```
>>> import pybamm
>>> values = {"some parameter": 1, "another parameter": 2}
>>> param = pybamm.ParameterValues(values)
>>> param["some parameter"]
1
>>> file = "input/parameters/lithium-ion/cells/kokam_Marquis2019/parameters.csv"
>>> values_path = pybamm.get_parameters_filepath(file)
>>> param = pybamm.ParameterValues(values=values_path)
>>> param["Negative current collector thickness [m]"]
```

(continues on next page)

(continued from previous page)

```
2.5e-05
>>> param = pybamm.ParameterValues(chemistry=pybamm.parameter_sets.Marquis2019)
>>> param["Reference temperature [K]"]
298.15
```

copy()
Returns a copy of the parameter values. Makes sure to copy the internal dictionary.

evaluate(*symbol*)
Process and evaluate a symbol.

Parameters **symbol** (*pybamm.Symbol*) – Symbol or Expression tree to evaluate

Returns The evaluated symbol

Return type number of array

static find_parameter(*path*)
Look for parameter file in the different locations in PARAMETER_PATH

get(*key*, *default=None*)
Return item correspoonding 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. For dimensionless parameters that depend on the C-rate, the value is given as a function of the C-rate (either x * Crate or x / Crate depending on the dependence)

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*

read_parameters_csv(*filename*)
Reads parameters from csv file into dict.

Parameters **filename** (*str*) – The name of the csv file containing the parameters.

Returns {name: value} pairs for the parameters.

Return type dict

search(*key*, *print_values=True*)
Search dictionary for keys containing 'key'.

See pybamm.FuzzyDict.search().

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

update_from_chemistry(*chemistry*)
Load standard set of components from a 'chemistry' dictionary

values()
Get the values of the dictionary

3.3.2 Geometric Parameters

Standard geometric parameters

3.3.3 Electrical Parameters

3.3.4 Thermal Parameters

3.3.5 Standard Lithium-ion Parameters

Standard parameters for lithium-ion battery models

3.3.6 Standard Lead-Acid Parameters

Standard Parameters for lead-acid battery models

3.3.7 Parameters Sets

Parameter sets from papers. The 'citation' entry provides a reference to the appropriate paper in the file "pybamm/CITATIONS.txt". To see which parameter sets have been used in your simulation, add the line "pybamm.print_citations()" to your script.

3.4 Geometry

3.4.1 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}}}
```

Extends: dict

Parameters **geometries** (*dict*) – The dictionary to create the geometry with

parameters
Returns all the parameters in the geometry

3.4.2 Battery Geometry

pybamm.**battery_geometry**(*include_particles=True*, *current_collector_dimension=0*)
A convenience function to create battery geometries.

Parameters

- **include_particles** (*bool*) – Whether to include particle domains
- **current_collector_dimensions** (*int, default*) – The dimensions of the current collector. Should be 0 (default), 1 or 2

Returns A geometry class for the battery

Return type *pybamm.Geometry*

3.5 Meshes

3.5.1 Meshes

class pybamm.**Mesh**(*geometry*, *submesh_types*, *var_pts*)
Mesh contains a list of submeshes on each subdomain.

Extends: dict

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

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.

3.5.2 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.SubMesh*) –

3.5.3 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.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.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

+ a,for k = 1, . . . , N, where N is the number of nodes.

Is side is “right”, the gridpoints are given by

+ a,for k = 1, . . . , N.

If side is “symmetric”, the first half of the interval is meshed using the gridpoints

+ 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.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.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.SubMesh1D*) –

3.5.4 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.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.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

+ 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.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.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.ScikitSubMesh2D*) –

3.6 Discretisation and spatial methods

3.6.1 Discretisation

class pybamm.**Discretisation**(*mesh=None*, *spatial_methods=None*)

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_initial_conditions(*model*)

Check initial conditions are a numpy array

check_initial_conditions_rhs(*model*)
Check initial conditions and rhs have the same shape

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

check_variables(*model*)
Check variables in variable list against rhs Be lenient with size check if the variable in model.variables is broadcasted, or a concatenation (if broadcasted, variable is a multiplication with a vector of ones)

create_jacobian(*model*)
Creates Jacobian 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. The Jacobian is df/dy.

Note: At present, calculation of the Jacobian is deferred until after simplification, since it is much faster to compute the Jacobian of the simplified model. However, in some use cases (e.g. running the same model multiple times but with different parameters) it may be more efficient to compute the Jacobian once, before simplification, so that parameters in the Jacobian can be updated (see PR #670).

Parameters **model** (*pybamm.BaseModel*) – Discretised model. Must have attributes rhs, initial_conditions and boundary_conditions (all dicts of {variable: equation})

Returns The expression trees corresponding to the Jacobian of the model

Return type *pybamm.Concatenation*

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*)
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)

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*, *check_model=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.
- **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 recommened to leave this option as True as it may help to identify any errors. 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_external_variables(*model*)
Add external variables to the list of variables to account for, being careful about concatenations

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

3.6.2 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.

Parameters **mesh** – Contains all the submeshes for discretisation

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 approriate 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*, *domain*, *auxiliary_domains*, *broadcast_type*)
Broadcast symbol to a specified domain.

Parameters

- **symbol** (*pybamm.Symbol*) – The symbol to be broadcasted
- **domain** (*iterable of strings*) – The domain to broadcast to
- **auxiliary_domains** (*dict of strings*) – The auxiliary domains for broadcasting
- **broadcast_type** (*str*) – The type of broadcast: 'primary to node', 'primary to edges', 'secondary to nodes', 'secondary 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.id: {"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*

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.id: {"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.id: {"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*)
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

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.id: {"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.id: {"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*

3.6.3 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

- **mesh** (*pybamm.Mesh*) – Contains all the submeshes for discretisation
- ****Extends** (“”: *pybamm.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(*domains*, *vector_type='row'*)
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 **domains** (*dict*) – The domain(s) and auxiliary domains of integration

Returns

- *pybamm.Matrix* – The finite volume integral matrix for the domain
- **vector_type** (*str, optional*) – Whether to return a row or column vector in the primary dimension (default is row)

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()*

gradient (*symbol*, *discretised_symbol*, *boundary_conditions*)
Matrix-vector multiplication to implement the gradient operator. See *pybamm.SpatialMethod.gradient()*

gradient_matrix (*domain*, *auxiliary_domains*)
Gradient matrix for finite volumes in the appropriate domain. Equivalent to grad(y) = (y[1:] - y[:-1])/dx

Parameters

- **domains** (*list*) – The domain(s) in which to compute the gradient matrix, including ghost nodes
- **auxiliary_domains** (*dict*) – The auxiliary domains in which to compute the gradient matrix

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

$$F(x) = \int_x^e ndf(u)\,du$$

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*)
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()*

preprocess_external_variables(*var*)
For finite volumes, we need the boundary fluxes for discretising properly. Here, we extrapolate and then add them to the boundary conditions.

Parameters **var** (*pybamm.Variable* or *pybamm.Concatenation*) – The external variable that is to be processed

Returns **new_bcs** – A dictionary containing the new boundary conditions

Return type dict

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*

3.6.4 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.

Parameters

- **mesh** (*pybamm.Mesh*) – Contains all the submeshes for discretisation
- ****Extends** (“”: *pybamm.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.id: {“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 assemled finite element matrices to account for boundary conditons.

Parameters

- **M** (scipy.sparse.coo_matrix) – The assemled finite element matrix to adjust.
- **boundary** (numpy.array) – Array of the indicies 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.id: {"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(*domains*, *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

- **domains** (*dict*) – The domain(s) of integration
- **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.id: {"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.id: {"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*)
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*)
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.id: {"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.id: {"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.id: {"negative tab": neg. tab bc, "positive tab": pos. tab bc}})

Returns The (sparse) finite element stiffness matrix for the domain

Return type *pybamm.Matrix*

3.6.5 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.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*)
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.

3.7 Solvers

3.7.1 Base Solver

class pybamm.**BaseSolver**(*method=None*, *rtol=1e-06*, *atol=1e-06*, *root_method=None*, *root_tol=1e-06*, *max_steps='deprecated'*)
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).

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 states
- **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)

Return type array-like, same shape as y0_guess

copy()

Returns a copy of the solver

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. 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*)

Unpack model, perform checks, simplify 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

solve(*model*, *t_eval=None*, *external_variables=None*, *inputs=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

- **t_eval** (*numeric type*) – The times (in seconds) at which to compute the solution
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving

Raises pybamm.ModelError – If an empty model is passed (*model.rhs = {}* and *model.algebraic={}* and *model.variables = {}*)

step(*old_solution*, *model*, *dt*, *npts=2*, *external_variables=None*, *inputs=None*, *save=True*)

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
- **npts** (*int, optional*) – The number of points at which the solution will be returned during the step dt. default is 2 (returns the solution at t0 and t0 + dt).
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **save** (*bool*) – Turn on to store the solution of all previous timesteps

Raises pybamm.ModelError – If an empty model is passed (*model.rhs = {}* and *model.algebraic = {}* and *model.variables = {}*)

3.7.2 Dummy Solver

class pybamm.**DummySolver**

Dummy solver class for empty models.

3.7.3 Scipy Solver

class pybamm.**ScipySolver**(*method='BDF'*, *rtol=1e-06*, *atol=1e-06*, *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).
- **extra_options** (*dict, optional*) – Any options to pass to the solver. Please consult SciPy documentation for details.

3.7.4 Scikits.odes Solvers

class pybamm.**ScikitsOdeSolver**(*method='cvode', rtol=1e-06, atol=1e-06, linsolver='deprecated', extra_options=None*)

Solve a discretised model, using scikits.odes.

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).
- **extra_options** (*dict, optional*) – Any options to pass to the solver. Please consult scikits.odes documentation for details. Some common keys:
  - 'linsolver': can be 'dense' (= default), 'lapackdense', 'spgmr', 'spbcgs', 'sptfqmr'

class pybamm.**ScikitsDaeSolver**(*method='ida', rtol=1e-06, atol=1e-06, root_method='casadi', root_tol=1e-06, extra_options=None, max_steps='deprecated'*)

Solve a discretised model, using scikits.odes.

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).
- **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).
- **extra_options** (*dict, optional*) – Any options to pass to the solver. Please consult scikits.odes documentation for details. Some common keys:
  - 'max_steps': maximum (int) number of steps the solver can take

3.7.5 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, extra_options_setup=None, extra_options_call=None*)

Solve a discretised model, using CasADi.

Extends: *pybamm.BaseSolver*

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.
  - "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.

- ”old safe”: perform step-and-check integration in steps of size dt for each dt in t_eval, checking whether events have been triggered.

- **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_counts** (*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 corresponds to a non-dimensional time of 0.01 (i.e. `0.01 * model.timescale_eval`).
- **extra_options_setup** (*dict, optional*) – Any options to pass to the CasADi integrator when creating the integrator. Please consult CasADi documentation for details. Some typical options:

  - ”max_num_steps”: Maximum number of integrator steps

- **extra_options_call** (*dict, optional*) – Any options to pass to the CasADi integrator when calling the integrator. Please consult CasADi documentation for details.

3.7.6 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”)
- **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.

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.

3.7.7 Solutions

class pybamm.**_BaseSolution**(*t*, *y*, *t_event=None*, *y_event=None*, *termination='final time'*, *copy_this=None*)

(Semi-private) class containing the solution of, and various attributes associated with, a PyBaMM model. This class is automatically created by the *Solution* class, and should never be called from outside the *Solution* class.

Parameters

- **t** (numpy.array, size (n,)) – A one-dimensional array containing the times at which the solution is evaluated
- **y** (numpy.array, size (m, n)) – A two-dimensional array containing the values of the solution. y[i, :] is the vector of solutions at time t[i].
- **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
- **copy_this** (*pybamm.Solution*, optional) – A solution to copy, if provided. Default is None.

inputs
Values of the inputs

model
Model used for solution

save(*filename*)
Save the whole solution using pickle

save_data(*filename*, *variables=None*, *to_format='pickle'*)
Save solution data only (raw arrays)

Parameters

- **filename** (*str*) – The name of the file to save data to
- **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 (1D variables only)

t
Times at which the solution is evaluated

t_event
Time at which the event happens

termination
Reason for termination

update(*variables*)
Add ProcessedVariables to the dictionary of variables in the solution

y
Values of the solution

y_event
Value of the solution at the time of the event

class pybamm.**Solution**(*t*, *y*, *t_event=None*, *y_event=None*, *termination='final time'*)
Class extending the base solution, with additional functionality for concatenating different solutions together

Extends: *_BaseSolution*

append(*solution*, *start_index=1*, *create_sub_solutions=False*)
Appends solution.t and solution.y onto self.t and self.y.

Note: by default this process removes the initial time and state of solution to avoid duplicate times and states being stored (self.t[-1] is equal to solution.t[0], and self.y[:, -1] is equal to solution.y[:, 0]). Set the optional argument `start_index` to override this behavior

sub_solutions
List of sub solutions that have been concatenated to form the full solution

3.7.8 Post-Process Variables

class pybamm.**ProcessedVariable**(*base_variable*, *solution*, *known_evals=None*, *warn=True*)
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_variable** (*pybamm.Symbol*) – A base variable with a method *evaluate(t,y)* that returns the value of that variable. 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)
- **solution** (*pybamm.Solution*) – The solution object to be used to create the processed variables
- **known_evals** (*dict*) – Dictionary of known evaluations, to be used to speed up finding the solution
- **warn** (*bool, optional*) – Whether to raise warnings when trying to evaluate time and length scales. Default is True.

call_1D(*t*, *x*, *r*, *z*)
Evaluate a 1D variable

call_2D(*t*, *x*, *r*, *y*, *z*)
Evaluate a 2D variable

data
Same as entries, but different name

get_spatial_scale(*name*, *domain=None*)
Returns the spatial scale for a named spatial variable

initialise_2D()
Initialise a 2D object that depends on x and r, or x and z.

class pybamm.**ProcessedSymbolicVariable**(*base_variable*, *solution*)
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_variable** (*pybamm.Symbol*) – A base variable with a method *evaluate(t,y)* that returns the value of that variable. 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)
- **solution** (*pybamm.Solution*) – The solution object to be used to create the processed variables

data
Same as entries, but different name

initialise_0D()
Create a 0D variable

initialise_1D()
Create a 1D variable

sensitivity(*inputs=None*, *check_inputs=True*)
Returns the sensitivity of the variable to the symbolic inputs at the specified input values

Parameters **inputs** (*dict*) – The inputs at which to evaluate the variable.

value(*inputs=None*, *check_inputs=True*)
Returns the value of the variable at the specified input values

Parameters **inputs** (*dict*) – The inputs at which to evaluate the variable.

value_and_sensitivity(*inputs=None*)
Returns the value of the variable and its sensitivity to the symbolic inputs at the specified input values

Parameters **inputs** (*dict*) – The inputs at which to evaluate the variable.

3.8 Experiments

Classes to help set operating conditions for some standard battery modelling experiments

3.8.1 Base Experiment Class

class pybamm.**Experiment**(*operating_conditions*, *parameters=None*, *period='1 minute'*)
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 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". 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".

Parameters

- **operating_conditions** (*list*) – List of operating conditions
- **parameters** (*dict*) – Dictionary of parameters to use for this experiment, replacing default parameters as appropriate
- **period** (*string, optional*) – Period (1/frequency) at which to record outputs. Default is 1 minute. Can be overwritten by individual operating conditions.

convert_electric(*electric*)
Convert electrical instructions to consistent output

convert_time_to_seconds(*time_and_units*)
Convert a time in seconds, minutes or hours to a time in seconds

read_operating_conditions(*operating_conditions*)
Convert operating conditions to the appropriate format

Parameters **operating_conditions** (*list*) – List of operating conditions

Returns **operating_conditions** – Operating conditions in the tuple format

Return type list

read_string(*cond*)
Convert a string to a tuple of the right format

Parameters **cond** (*str*) – String of appropriate form for example "Charge at x C for y hours". x and y must be numbers, 'C' denotes the unit of the external circuit (can be A for current, C for C-rate, V for voltage or W for power), and 'hours' denotes the unit of time (can be second(s), minute(s) or hour(s))

3.9 Simulation

class pybamm.**Simulation**(*model*, *experiment=None*, *geometry=None*, *parameter_values=None*, *submesh_types=None*, *var_pts=None*, *spatial_methods=None*, *solver=None*, *quick_plot_vars=None*, *C_rate=None*)
A Simulation class for easy building and running of PyBaMM simulations.

Parameters

- **model** (*pybamm.BaseModel*) – The model to be simulated
- **experiment** (*pybamm.Experiment* (optional)) – The experimental conditions under which to solve the model
- **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.
- **quick_plot_vars** (*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 experiment at.

build(*check_model=True*)
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. If you want to rebuild, first use "reset()". This method will automatically set the parameters if they have not already been set.

Parameters **check_model** (*bool, optional*) – If True, model checks are performed after discretisation (see *pybamm.Discretisation.process_model()*). Default is True.

get_variable_array (**variables*)
A helper function to easily obtain a dictionary of arrays of values for a list of variables at the latest timestep.

Parameters **variable** (*str*) – The name of the variable/variables you wish to obtain the arrays for.

Returns **variable_arrays** – A dictionary of the variable names and their corresponding arrays.

Return type dict

plot (*quick_plot_vars=None*, *testing=False*)
A method to quickly plot the outputs of the simulation.

Parameters

- **quick_plot_vars** (*list, optional*) – A list of the variables to plot.
- **bool, optional** (*testing,*) – If False the plot will not be displayed

reset (*update_model=True*)
A method to reset a simulation back to its unprocessed state.

save (*filename*)
Save simulation using pickle

set_defaults ()
A method to set all the simulation specs to default values for the supplied model.

set_parameters ()
A method to set the parameters in the model and the associated geometry. If the model has already been built or solved then this will first reset to the unprocessed state and then set the parameter values.

set_up_experiment (*model*, *experiment*)
Set up a simulation to run with an experiment. This creates a dictionary of inputs (current/voltage/power, running time, stopping condition) for each operating condition in the experiment. The model will then be solved by integrating the model successively with each group of inputs, one group at a time.

solve (*t_eval=None*, *solver=None*, *external_variables=None*, *inputs=None*, *check_model=True*)
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 compute the solution. If None and the parameter "Current function [A]" is not read from data the model will be solved for a full discharge (1 hour / C_rate). If None and the parameter "Current function [A]" is read from data the model will be solved at the times provided in the data.
- **solver** (*pybamm.BaseSolver*) – The solver to use to solve the model.
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time. The variables must correspond to the variables that would normally be found by solving the submodels that have been made external.
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **check_model** (*bool, optional*) – If True, model checks are performed after discretisation (see *pybamm.Discretisation.process_model()*). Default is True.

specs(*model_options=None*, *geometry=None*, *parameter_values=None*, *submesh_types=None*, *var_pts=None*, *spatial_methods=None*, *solver=None*, *quick_plot_vars=None*, *C_rate=None*)
A method to set the various specs of the simulation. This method automatically resets the model after the new specs have been set.

Parameters

- **model_options** (*dict, optional*) – A dictionary of options to tweak the model you are using
- **geometry** (*pybamm.Geometry*, optional) – The geometry upon which to solve the model
- **parameter_values** (*dict, optional*) – A dictionary of 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.
- **quick_plot_vars** (*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 experiment at.

step(*dt*, *solver=None*, *npts=2*, *external_variables=None*, *inputs=None*, *save=True*)
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.
- **npts** (*int, optional*) – The number of points at which the solution will be returned during the step dt. default is 2 (returns the solution at t0 and t0 + dt).
- **external_variables** (*dict*) – A dictionary of external variables and their corresponding values at the current time. The variables must correspond to the variables that would normally be found by solving the submodels that have been made external.
- **inputs** (*dict, optional*) – Any input parameters to pass to the model when solving
- **save** (*bool*) – Turn on to store the solution of all previous timesteps

3.10 Plotting

3.10.1 Quick Plot

class pybamm.**QuickPlot**(*solutions*, *output_variables=None*, *labels=None*, *colors=None*, *linestyles=None*, *figsize=None*, *time_unit=None*, *spatial_unit='um'*, *variable_limits='fixed'*)
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 ["r", "b", "k", "g", "m", "c"]
- **linestyles** (*list of str, optional*) – The linestyles to loop over when plotting. Defaults to ["-", ":", "–", "-."]
- **figsize** (*tuple of floats, optional*) – The size of the figure to make
- **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:
  - "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).

dynamic_plot(*testing=False*, *step=None*)

Generate a dynamic plot with a slider to control the time.

Parameters

- **step** (*float*) – For notebook mode, size of steps to allow in the slider. Defaults to 1/100th of the total time.
- **testing** (*bool*) – Whether to actually make the plot (turned off for unit tests)

get_spatial_var(*key*, *variable*, *dimension*)

Return the appropriate spatial variable(s)

plot(*t*)

Produces a quick plot with the internal states at time t.

Parameters **t** (*float*) – Dimensional time (in hours) at which to plot.

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

3.10.2 Dynamic Plot

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 'testing' is passed to the 'dynamic_plot' method, not the *QuickPlot* class.

Returns **plot** – The 'QuickPlot' object that was created

Return type *pybamm.QuickPlot*

3.10.3 Plot

pybamm.**plot**(*x*, *y*, *xlabel=None*, *ylabel=None*, *title=None*, *testing=False*, ***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
- **xlabel** (*str, optional*) – The label for the x axis
- **ylabel** (*str, optional*) – The label for the y axis
- **testing** (*bool, optional*) – Whether to actually make the plot (turned off for unit tests)

3.10.4 Plot 2D

pybamm.**plot2D**(*x*, *y*, *z*, *xlabel=None*, *ylabel=None*, *title=None*, *testing=False*, ***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)
- **xlabel** (*str, optional*) – The label for the x axis
- **ylabel** (*str, optional*) – The label for the y axis
- **title** (*str, optional*) – The title for the plot
- **testing** (*bool, optional*) – Whether to actually make the plot (turned off for unit tests)

3.11 Utility functions

pybamm.**get_infinite_nested_dict**()

Return a dictionary that allows infinite nesting without having to define level by level.

See: https://stackoverflow.com/questions/651794/whats-the-best-way-to-initialize-a-dict-of-dicts-in-python/652226#652226

Example

```
>>> import pybamm
>>> d = pybamm.get_infinite_nested_dict()
>>> d["a"] = 1
>>> d["a"]
1
>>> d["b"]["c"]["d"] = 2
>>> d["b"]["c"] == {"d": 2}
True
```

pybamm.**load_function**(*filename*)
Load a python function from a file "function_name.py" called "function_name". The filename might either be an absolute path, in which case that specific file will be used, or the file will be searched for relative to PyBaMM root.

Parameters **filename** (*str*) – The name of the file containing the function of the same name.

Returns The python function loaded from the file.

Return type function

pybamm.**rmse**(*x*, *y*)
Calculate the root-mean-square-error between two vectors x and y, ignoring NaNs

pybamm.**root_dir**()
return the root directory of the PyBaMM install directory

class pybamm.**Timer**
Provides accurate timing.

Example

timer = pybamm.Timer() print(timer.format(timer.time()))

format(*time=None*)
Formats a (non-integer) number of seconds, returns a string like "5 weeks, 3 days, 1 hour, 4 minutes, 9 seconds", or "0.0019 seconds".

Parameters **time** (*float, optional*) – The time to be formatted.

Returns The string representation of `time` in human-readable form.

Return type string

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.

3.12 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. For a list of all possible citations, see *pybamm/CITATIONS.txt*

Examples

```
>>> import pybamm
>>> pybamm.citations.register("sulzer2020python")
>>> pybamm.print_citations("citations.txt")
```

print(*filename=None*)
Print all citations that were used for running simulations.

Parameters **filename** (*str, optional*) – Filename to which to print citations. If None, citations are printed to the terminal.

read_citations()
Read the citations text file

register(*key*)
Register a paper to be cited. The intended use is that *register()* should be called only when the referenced functionality is actually being used.

Parameters **key** (*str*) – The key for the paper to be cited

pybamm.**print_citations**(*filename=None*)
See *Citations.print()*

3.13 Parameters command line interface

PyBaMM comes with a small command line interface that can be used to manage parameter sets. By default, PyBaMM provides parameters in the "input" directory located in the pybamm package directory. If you wish to add new parameters, you can first pull a given parameter directory into the current working directory using the command `pybamm_edit_parameter` for manual editing. By default, PyBaMM first looks for parameter defined in the current working directory before falling back the package directory if nothing is found locally. If you wish to access a newly defined parameter set from anywhere in your system, you can use `pybamm_add_parameter` to copy a given parameter directory to the package directory. To get a list of currently available parameter sets, use `pybamm_list_parameters`.

pybamm.parameters_cli.**add_parameter**(*arguments=None*)
Add a parameter directory to package input directory. This allows the parameters to be used from anywhere in the system.

Example: "add_parameter foo lithium-ion anodes" will copy directory foo in "pybamm/input/parameters/lithium-ion/anodes".

pybamm.parameters_cli.**remove_parameter**(*arguments=None*)
Remove a parameter directory from package input directory.

Example: "rm_parameter foo lithium-ion anodes" will remove directory foo in "pybamm/input/parameters/lithium-ion/anodes".

pybamm.parameters_cli.**edit_parameter**(*arguments=None*)
Copy a given default parameter directory to the current working directory for editing. For example

```
edit_param(["lithium-ion"])
```

will create the directory structure:

```
lithium-ion/
  anodes/
    graphite_Chen2020
    ...
  cathodes/
  ...
```

in the current working directory.

CHAPTER 4

Examples

Detailed examples can be viewed on the GitHub examples page, and run locally using `jupyter notebook`, or online through Binder.

CHAPTER 5

Contributing

There are many ways to contribute to PyBaMM:

5.1 Adding Parameter Values

As with any contribution to PyBaMM, please follow the workflow in CONTRIBUTING.md. In particular, start by creating an issue to discuss what you want to do - this is a good way to avoid wasted coding hours!

5.1.1 The role of parameter values

All models in PyBaMM are implemented as expression trees. At the stage of creating a model, we use *pybamm.Parameter* and *pybamm.FunctionParameter* objects to represent parameters and functions respectively.

We then create a `ParameterValues` class, using a specific set of parameters, to iterate through the model and replace any *pybamm.Parameter* objects with a *pybamm.Scalar* and any *pybamm.FunctionParameter* objects with a *pybamm.Function*.

For an example of how the parameter values work, see the parameter values notebook.

5.1.2 Adding a set of parameters values

Parameter sets are split by material into `anodes`, `separators`, `cathodes`, `electrolytes`, `cells` (for cell geometries and thermal properties) and `experiments` (for initial conditions and charge/discharge rates). To add a new parameter set in one of these subcategories, first create a new folder in the appropriate chemistry folder: for example, to add a new anode chemistry for lithium-ion, add a subfolder `input/parameters/lithium-ion/anodes/new_anode_chemistry_AuthorYear`. This subfolder should then contain:

- a csv file `parameters.csv` with all the relevant scalar parameters. The expected structure of the csv file is:

Name [Units]	Value	Reference	Notes
Example [m]	13	AuthorYear	an example

Empty lines, and lines starting with #, will be ignored.

- a `README.md` file with information on where these parameters came from
- python files for any functions, which should be referenced from the `parameters.csv` file (see `Adding a Function` below)
- csv files for any data to be interpolated, which should be referenced from the `parameters.csv` file (see `Adding data for interpolation` below)

The easiest way to start is to copy an existing file (e.g. ``input/parameters/lithium-ion/anodes/graphite_mcmb2528_Marquis2019``) and replace all entries in all files as appropriate

5.1.3 Adding a function

Functions should be added as Python functions under a file with the same name in the appropriate chemistry folder in `input/parameters/`. These Python functions should be documented with references explaining where they were obtained. For example, we would put the following Python function in a file `input/parameters/lithium_ion/anodes/new_anode_chemistry_AuthorYear/diffusivity_AuthorYear.py`

```
def diffusivity_AuthorYear(c_e):
    """
    Dimensional Fickian diffusivity in the electrolyte [m2.s-1], from [1]_, as a
    function of the electrolyte concentration c_e [mol.m-3].

    References
    ----------
    .. [1] J Bloggs, AN Other. A set of parameters. A Chemistry Journal,
           123(4):567-573, 2019.

    """
    return (1.75 + 260e-6 * c_e) * 1e-9
```

Then, these functions should be added to the parameter file from which they will be called (must be in the same folder), with the tag `[function]`, for example:

Name [Units]	Value	Reference	Notes
Example [m2.s-1]	[function]diffusivity_AuthorYear	AuthorYear	a function

5.1.4 Adding data for interpolation

Data should be added as as csv file in the appropriate chemistry folder in `input/parameters/`. For example, we would put the following data in a file `input/parameters/lithium_ion/anodes/new_anode_chemistry_AuthorYear/diffusivity_AuthorYear.csv`

# concentration [mol/m3]	Diffusivity [m2/s]
0.0000000000000000e+00	4.714135898019971016e+00
2.040816326530612082e-02	4.708899441575220557e+00
4.081632653061224164e-02	4.702448345762175741e+00
6.122448979591836593e-02	4.694558534379876136e+00
8.163265306122448328e-02	4.684994372928071193e+00
1.020408163265306006e-01	4.673523893805322516e+00
1.224489795918367319e-01	4.659941254449398329e+00
1.428571428571428492e-01	4.644096031712390271e+00

Empty lines, and lines starting with #, will be ignored.

Then, this data should be added to the parameter file from which it will be called (must be in the same folder), with the tag `[data]`, for example:

Name [Units]	Value	Reference	Notes
Example [m2.s-1]	[data]diffusivity_AuthorYear	AuthorYear	some data

5.1.5 Using new parameters

If you have added a whole new set of parameters, then you can create a new parameter set in `pybamm/parameters/parameter_sets.py`, by just adding a new dictionary to that file, for example

```
AuthorYear = {
    "chemistry": "lithium-ion",
    "cell": "new_cell_AuthorYear",
    "anode": "new_anode_AuthorYear",
    "separator": "new_separator_AuthorYear",
    "cathode": "new_cathode_AuthorYear",
    "electrolyte": "new_electrolyte_AuthorYear",
    "experiment": "new_experiment_AuthorYear",
}
```

Then, to use these new parameters, use:

```
param = pybamm.ParameterValues(chemistry=pybamm.parameter_sets.AuthorYear)
```

Note that you can re-use existing parameter subsets instead of creating new ones (for example, you could just replace "experiment": "new_experiment_AuthorYear" with "experiment": "1C_discharge_from_full_Marquis2019" in the above dictionary).

It's also possible to add parameters for a single material (e.g. anode) and then re-use existing parameters for the other materials, without adding a parameter set to `pybamm/parameters/parameter_sets.py`.

```
param = pybamm.ParameterValues(
    chemistry={
        "chemistry": "lithium-ion",
        "cell": "kokam_Marquis2019",
        "anode": "new_anode_chemistry_AuthorYear",
        "separator": "separator_Marquis2019",
        "cathode": "lico2_Marquis2019",
        "electrolyte": "lipf6_Marquis2019",
        "experiment": "1C_discharge_from_full_Marquis2019",
    }
)
```

or, equivalently in this case (since the only difference from the standard parameters from Marquis et al. is the set of anode parameters),

```
param = pybamm.ParameterValues(
    chemistry={
        **pybamm.parameter_sets.Marquis2019,
        "anode": "new_anode_chemistry_AuthorYear",
    }
)
```

See the "Getting Started" tutorial for examples of setting parameters in action.

5.1.6 Unit tests for the new class

You might want to add some unit tests to show that the parameters combine as expected (see e.g. lithium-ion parameter tests), but this is not crucial.

5.1.7 Test on the models

In theory, any existing model can now be solved using the new parameters instead of their default parameters, with no extra work from here. To test this, add something like the following test to one of the model test files (e.g. DFN):

```
def test_my_new_parameters(self):
    model = pybamm.lithium_ion.DFN()
    parameter_values = pybamm.ParameterValues(chemistry=pybamm.parameter_sets.
↪AuthorYear)
    modeltest = tests.StandardModelTest(model, parameter_values=parameter_values)
    modeltest.test_all()
```

This will check that the model can run with the new parameters (but not that it gives a sensible answer!).

Once you have performed the above checks, you are almost ready to merge your code into the core PyBaMM - see CONTRIBUTING.md workflow for how to do this.

5.2 Adding a Model

As with any contribution to PyBaMM, please follow the workflow in CONTRIBUTING.md. In particular, start by creating an issue to discuss what you want to do - this is a good way to avoid wasted coding hours!

We aim here to provide an overview of how a new model is entered into PyBaMM in a form which can be eventually merged into the master branch of the PyBaMM project. However, we recommend that you first read through the notebook: create a model, which goes step-by-step through the procedure for creating a model. Once you understand that procedure, you can then formalise your model following the outline provided here.

5.2.1 The role of models

One of the main motivations for PyBaMM is to allow for new models of batteries to be easily be added, solved, tested, and compared without requiring a detailed knowledge of sophisticated numerical methods. It has therefore been our focus to make the process of adding a new model as simple as possible. To achieve this, all models in PyBaMM are implemented as expression trees, which abstract away the details of computation.

The fundamental building blocks of a PyBaMM expression tree are *pybamm.Symbol*. There are different types of *pybamm.Symbol*: *pybamm.Variable*, *pybamm.Parameter*, *pybamm.Addition*, *pybamm.Multiplication*, *pybamm.Gradient* etc which have been created so that each component of a model written out in PyBaMM mirrors exactly the written mathematics. For example, the expression:

$$\nabla \cdot (D(c)\nabla c) + aFj$$

is simply written as

```
div(D(c) * grad(c)) + a * F * j
```

within PyBaMM. A model in PyBaMM is essentially an organised collection of expression trees.

5.2.2 Implementing a new model

To add a new model (e.g. My New Model), first create a new file (`my_new_model.py`) in `pybamm/models` (or the relevant subdirectory). In this file create a new class which inherits from *pybamm.BaseModel* (or `pybamm.LithiumIonBaseModel` if you are modelling a full lithium-ion battery or `pybamm.LeadAcidBaseModel` if you are modelling a full lead acid battery):

```
class MyNewModel(pybamm.BaseModel):
    def
```

and add the class to `pybamm/__init__.py`:

```
from .models.my_new_model import MyNewModel
```

(this line will be slightly different if you created your model in a subdirectory of models). Within your new class `MyNewModel`, first create an initialisation function which calls the initialisation function of the parent class

```
def __init__(self):
    super().__init__()
```

Within the initialisation function of `MyNewModel` you must then define the following attributes:

- `self.rhs`
- `self.algebraic`
- `self.boundary_conditions`
- `self.initial_conditions`
- `self.variables`

You may also optionally also provide:

- `self.events`
- `self.default_geometry`
- `self.default_solver`
- `self.default_spatial_methods`
- `self.default_submesh_types`
- `self.default_var_pts`
- `self.default_parameter_values`

We will go through each of these attributes in turn here for completeness but refer the user to the API documentation or example notebooks (create-model.ipnb) if further details are required.

Governing equations

The governing equations which can either be parabolic or elliptic are entered into the `self.rhs` and `self.algebraic` dictionaries, respectively. We associate each governing equation with a subject variable, which is the variable that is found when the equation is solved. We use this subject variable as the key of the dictionary. For parabolic equations, we rearrange the equation so that the time derivative of the subject variable is the only term on the left hand side of the equation. We then simply write the resulting right hand side into the `self.rhs` dictionary with the subject variable as the key. For elliptic equations, we rearrange so that the left hand side of the equation if zero and then write the right hand side into the `self.algebraic` dictionary in the same way. The resulting dictionary should look like:

```
self.rhs = {parabolic_var1: parabolic_rhs1, parabolic_var2: parabolic_rhs2, ...}
self.algebraic = {elliptic_var1: elliptic_rhs1, elliptic_var2: elliptic_rhs2, ...}
```

Boundary conditions

Boundary conditions on a variable can either be Dirichlet or Neumann (support for mixed boundary conditions will be added at a later date). For a variable c on a one dimensional domain with a Dirichlet condition of $c = 1$ on the left boundary and a Neumann condition of $\nabla c = 2$ on the right boundary, we then have:

```
self.boundary_conditions = {c: {"left": (1, "Dirichlet"), "right": (2, "Neumann")}}
```

Initial conditions

For a variable c that is initially at a value of $c = 1$, the initial condition is included written into the model as

```
self.initial_conditions = {c: 1}
```

Output variables

PyBaMM allows users to create combinations of symbols to output from their model. For example, we might wish to output the terminal voltage which is given by $V = \phi_{s,p}|_{x=1} - \phi_{s,n}|_{x=0}$. We would first define the voltage symbol V and then include it into the output variables dictionary in the form:

```
self.variables = {"Terminal voltage [V]": V}
```

Note that we indicate that the quanitity is dimensional by including the dimensions, Volts in square brackets. We do this to distinguish between dimensional and dimensionless outputs which may otherwise share the same name.

Note that if your model inherits from `pybamm.StandardBatteryBaseModel`, then there is a standard set of output parameters which is enforced to ensure consistency across models so that they can be easily tested and compared.

Events

Events can be added to stop computation when the event occurs. For example, we may wish to terminate our computation when the terminal voltage V reaches some minimum voltage during a discharge V_{min}. We do this by adding the following to the events dictionary:

```
self.events["Minimum voltage cut-off"] = V - V_min
```

Events will stop the solver whenever they return 0.

Setting defaults

It can be useful for testing, and quickly running a model to have a default setup. Each of the defaults listed above should adhere to the API requirements but in short, we require `self.default_geometry` to be a dictionary of the right format (see *pybamm.battery_geometry()*), `self.default_solver` to be an instance of *pybamm.BaseSolver*, and `self.default_parameter_values` to be an instance of *pybamm.ParameterValues*. We also require that `self.default_submesh_types` is a dictionary with keys which are strings corresponding to the regions of the battery (e.g. "negative electrode") and values which are an instance

of *pybamm.SubMesh1D*. The `self.default_spatial_methods` attribute is also required to be a dictionary with keys corresponding to the regions of the battery but with values which are an instance of *pybamm.SpatialMethod*. Finally, `self.default_var_pts` is required to be a dictionary with keys which are an instance of *pybamm.SpatialVariable* and values which are integers.

Using submodels

The inbuilt models in PyBaMM do not add all the model attributes within their own file. Instead, they make use of inbuilt submodel (a particle model, an electrolyte model, etc). There are two main reasons for this. First, the code in the submodels can then be used by multiple models cutting down on repeated code. This makes it easier to maintain the codebase because fixing an issue in a submodel fixes that issue everywhere the submodel is called (instead of having to track down the issue in every model). Secondly, it allows for the user to easily switch a submodel out for another and study the effect. For example, we may be using standard diffusion in the particles but decide that we would like to switch in particles which are phase separating. With submodels all we need to do is switch the submodel instead of re-writing the whole sections of the model. Submodel contributions are highly encouraged so where possible, try to divide your model into submodels.

In addition to calling submodels, common sets of variables and parameters found in lithium-ion and lead acid batteries are provided in *standard_variables.py*, *standard_parameters_lithium_ion.py*, *standard_parameters_lead_acid.py*, *electrical_parameters.py*, *geometric_parameters.py*, and *standard_spatial_vars.py* which we encourage use of to save redefining the same parameters and variables in every model and submodel.

5.2.3 Unit tests for a MyNewModel

We strongly recommend testing your model to ensure that it is behaving correctly. To do this, first create a new file `test_my_new_model.py` within `tests/integration/test_models` (or the appropriate subdirectory). Within this file, add the following code

```
import pybamm
import unittest

class TestMyNewModel(unittest.TestCase):
    def my_first_test(self):
        # add test here

if __name__ == "__main__":
    print("Add -v for more debug output")
    import sys

    if "-v" in sys.argv:
        debug = True
    unittest.main()
```

We can now add functions such as `my_first_test()` to `TestMyNewModel` which run specific tests. As a first test, we recommend you make use of `tests.StandardModelTest` which runs a suite of basic tests. If your new model is a full model of a battery and therefore inherits from `pybamm.StandardBatteryBaseModel` then `tests.StandardBatteryTest` will also check the set of outputs are producing reasonable behaviour.

Please see the tests of the inbuilt models to get a further idea of how to test the your model.

5.3 Adding a Spatial Method

As with any contribution to PyBaMM, please follow the workflow in CONTRIBUTING.md. In particular, start by creating an issue to discuss what you want to do - this is a good way to avoid wasted coding hours!

5.3.1 The role of spatial methods

All models in PyBaMM are implemented as expression trees. After it has been created and parameters have been set, the model is passed to the *pybamm.Discretisation* class, which converts it into a linear algebra form. For example, the object:

```
grad(u)
```

might get converted to a Matrix-Vector multiplication:

```
Matrix(100,100) @ y[0:100]
```

(in Python 3.5+, @ means matrix multiplication, while * is elementwise product). The *pybamm.Discretisation* class is a wrapper that iterates through the different parts of the model, performing the trivial conversions (e.g. Addition –> Addition), and calls upon spatial methods to perform the harder conversions (e.g. grad(u) –> Matrix * StateVector, SpatialVariable –> Vector, etc).

Hence SpatialMethod classes only need to worry about the specific conversions, and *pybamm.Discretisation* deals with the rest.

5.3.2 Implementing a new spatial method

To add a new spatial method (e.g. My Fast Method), first create a new file (`my_fast_method.py`) in `pybamm/spatial_methods/`, with a single class that inherits from *pybamm.SpatialMethod*, such as:

```
class MyFastMethod(pybamm.SpatialMethod):
```

and add the class to `pybamm/__init__.py`:

```
from .spatial_methods.my_fast_method import MyFastMethod
```

You can then start implementing the spatial method by adding functions to the class. In particular, any spatial method *must* have the following functions (from the base class *pybamm.SpatialMethod*):

- *pybamm.SpatialMethod.gradient()*
- *pybamm.SpatialMethod.divergence()*
- *pybamm.SpatialMethod.integral()*
- `pybamm.SpatialMethod.indefinite integral()`
- *pybamm.SpatialMethod.boundary_value_or_flux()*

Optionally, a new spatial method can also overwrite the default behaviour for the following functions:

- *pybamm.SpatialMethod.spatial_variable()*
- *pybamm.SpatialMethod.broadcast()*
- *pybamm.SpatialMethod.mass_matrix()*
- *pybamm.SpatialMethod.process_binary_operators()*

- *pybamm.SpatialMethod.concatenation()*

For an example of an existing spatial method implementation, see the Finite Volume API docs and notebook.

5.3.3 Unit tests for the new class

For the new spatial method to be added to PyBaMM, you must add unit tests to demonstrate that it behaves as expected (see, for example, the Finite Volume unit tests). The best way to get started would be to create a file `test_my_fast_method.py` in `tests/unit/test_spatial_methods/` that performs at least the following checks:

- Operations return objects that have the expected shape
- Standard operations behave as expected, e.g. (in 1D) grad(x^2) = 2*x, integral(sin(x), 0, pi) = 2
- (more advanced) make sure that the operations converge at the correct rate to known analytical solutions as you decrease the grid size

5.3.4 Test on the models

In theory, any existing model can now be discretised using `MyFastMethod` instead of their default spatial methods, with no extra work from here. To test this, add something like the following test to one of the model test files (e.g. DFN):

```
def test_my_fast_method(self):
    model = pybamm.lithium_ion.DFN()
    spatial_methods = {
        "macroscale": pybamm.MyFastMethod,
        "negative particle": pybamm.MyFastMethod,
        "positive particle": pybamm.MyFastMethod,
    }
    modeltest = tests.StandardModelTest(model, spatial_methods=spatial_methods)
    modeltest.test_all()
```

This will check that the model can run with the new spatial method (but not that it gives a sensible answer!).

Once you have performed the above checks, you are almost ready to merge your code into the core PyBaMM - see CONTRIBUTING.md workflow for how to do this.

5.4 Adding a Solver

As with any contribution to PyBaMM, please follow the workflow in CONTRIBUTING.md. In particular, start by creating an issue to discuss what you want to do - this is a good way to avoid wasted coding hours!

5.4.1 The role of solvers

All models in PyBaMM are implemented as expression trees. After the model has been created, parameters have been set, and the model has been discretised, the model is now a linear algebra object with the following attributes:

model.concatenated_rhs A *pybamm.Symbol* node that can be evaluated at a state (`t`, `y`) and returns the value of all the differential equations at that state, concatenated into a single vector

model.concatenated_algebraic A *pybamm.Symbol* node that can be evaluated at a state (`t`, `y`) and returns the value of all the algebraic equations at that state, concatenated into a single vector

model.concatenated_initial_conditions A numpy array of initial conditions for all the differential and algebraic equations, concatenated into a single vector

model.events A dictionary of *pybamm.Symbol* nodes representing events at which the solver should terminate. Specifically, the solver should terminate when any of the events in `model.events.values()` evaluate to zero

The role of solvers is to solve a model at a given set of time points, returning a vector of times `t` and a matrix of states `y`.

5.4.2 Base solver classes vs specific solver classes

There is one general base solver class, *pybamm.BaseSolver*, which sets up some useful solver properties such as tolerances and implement a method `self.solve()` that solves a model at a given set of time points.

The `solve` method unpacks the model, simplifies it by removing extraneous operations, (optionally) creates or calls the mass matrix and/or jacobian, and passes the appropriate attributes to another method, called `integrate`, which does the time-stepping. The role of specific solver classes is simply to implement this `integrate` method for an arbitrary set of derivative function, initial conditions etc.

The base solver class also computes a consistent set of initial conditions for the algebraic equations, using `model.concatenated_initial_conditions` as an initial guess.

5.4.3 Implementing a new solver

To add a new solver (e.g. My Fast DAE Solver), first create a new file (`my_fast_dae_solver.py`) in `pybamm/solvers/`, with a single class that inherits from *pybamm.BaseSolver*. For example:

```
def MyFastDaeSolver(pybamm.BaseSolver):
```

Also add the class to `pybamm/__init__.py`:

```
from .solvers.my_fast_dae_solver import MyFastDaeSolver
```

You can then start implementing the solver by adding the `integrate` function to the class.

For an example of an existing solver implementation, see the Scikits DAE solver API docs and notebook.

5.4.4 Unit tests for the new class

For the new solver to be added to PyBaMM, you must add unit tests to demonstrate that it behaves as expected (see, for example, the Scikits solver tests). The best way to get started would be to create a file `test_my_fast_solver.py` in `tests/unit/test_solvers/` that performs at least the following checks:

- The `integrate` method works on a simple ODE/DAE model with/without jacobian, mass matrix and/or events as appropriate
- The `solve` method works on a simple model (in theory, if the `integrate` method works then the `solve` method should always work)

If the solver is expected to converge in a certain way as the time step is changed, you could also add a convergence test in `tests/convergence/solvers/`.

5.4.5 Test on the models

In theory, any existing model can now be solved using *MyFastDaeSolver* instead of their default solvers, with no extra work from here. To test this, add something like the following test to one of the model test files (e.g. DFN):

```
def test_my_fast_solver(self):
    model = pybamm.lithium_ion.DFN()
    solver = pybamm.MyFastDaeSolver()
    modeltest = tests.StandardModelTest(model, solver=solver)
    modeltest.test_all()
```

This will check that the model can run with the new solver (but not that it gives a sensible answer!).

Once you have performed the above checks, you are almost ready to merge your code into the core PyBaMM - see CONTRIBUTING.md workflow for how to do this.

Before contributing, please read the Contribution Guidelines.

Python Module Index

p

pybamm, 15
pybamm.parameters.electrical_parameters, 97
pybamm.parameters.geometric_parameters, 97
pybamm.parameters.parameter_sets, 97
pybamm.parameters.standard_parameters_lead_acid, 97
pybamm.parameters.standard_parameters_lithium_ion, 97
pybamm.parameters.thermal_parameters, 97

Index

Symbols

_BaseSolution (*class in pybamm*), 122
__abs__() (*pybamm.Symbol method*), 15
__add__() (*pybamm.Symbol method*), 15
__ge__() (*pybamm.Symbol method*), 15
__getitem__() (*pybamm.Symbol method*), 15
__gt__() (*pybamm.Symbol method*), 15
__init__() (*pybamm.Symbol method*), 16
__le__() (*pybamm.Symbol method*), 16
__lt__() (*pybamm.Symbol method*), 16
__matmul__() (*pybamm.Symbol method*), 16
__mul__() (*pybamm.Symbol method*), 16
__neg__() (*pybamm.Symbol method*), 16
__pow__() (*pybamm.Symbol method*), 16
__radd__() (*pybamm.Symbol method*), 16
__repr__() (*pybamm.Symbol method*), 16
__rmatmul__() (*pybamm.Symbol method*), 16
__rmul__() (*pybamm.Symbol method*), 16
__rpow__() (*pybamm.Symbol method*), 16
__rsub__() (*pybamm.Symbol method*), 16
__rtruediv__() (*pybamm.Symbol method*), 16
__str__() (*pybamm.Symbol method*), 16
__sub__() (*pybamm.Symbol method*), 16
__truediv__() (*pybamm.Symbol method*), 16

A

AbsoluteValue (*class in pybamm*), 27
add_ghost_meshes() (*pybamm.Mesh method*), 98
add_ghost_nodes() (*pybamm.FiniteVolume method*), 109
add_neumann_values() (*pybamm.FiniteVolume method*), 109
add_parameter() (*in module pybamm.parameters_cli*), 131
Addition (*class in pybamm*), 25
algebraic (*pybamm.BaseModel attribute*), 40
algebraic (*pybamm.BaseSubModel attribute*), 49
AlgebraicSolver (*class in pybamm*), 121
AlternativeEffectiveResistance2D (*class in pybamm.current_collector*), 52
append() (*pybamm.Solution method*), 123
Array (*class in pybamm*), 22
assemble_mass_form() (*pybamm.ScikitFiniteElement method*), 114
auxiliary_domains (*pybamm.Symbol attribute*), 16

B

BackwardTafel (*class in pybamm.interface*), 73
BaseBatteryModel (*class in pybamm*), 43
BaseCompositePotentialPair (*class in pybamm.current_collector*), 51
BaseElectrode (*class in pybamm.electrode*), 59
BaseElectrolyteConductivity (*class in pybamm.electrolyte_conductivity*), 62
BaseElectrolyteDiffusion (*class in pybamm.electrolyte_diffusion*), 67
BaseHigherOrderModel (*class in pybamm.lead_acid*), 47
BaseInterface (*class in pybamm.interface*), 71
BaseKinetics (*class in pybamm.interface*), 71
BaseModel (*class in pybamm*), 40
BaseModel (*class in pybamm.convection*), 55
BaseModel (*class in pybamm.current_collector*), 51
BaseModel (*class in pybamm.electrode.ohm*), 59
BaseModel (*class in pybamm.lead_acid*), 47
BaseModel (*class in pybamm.lithium_ion*), 45
BaseModel (*class in pybamm.oxygen_diffusion*), 79
BaseModel (*class in pybamm.porosity*), 86
BaseModel (*class in pybamm.sei*), 75
BaseModel (*class in pybamm.tortuosity*), 93
BaseParticle (*class in pybamm.particle*), 83
BasePotentialPair (*class in pybamm.current_collector*), 53
BaseQuiteConductivePotentialPair (*class in pybamm.current_collector*), 54
BaseSolver (*class in pybamm*), 117
BaseSubModel (*class in pybamm*), 49
BaseThermal (*class in pybamm.thermal*), 89

BaseThroughCellModel (*class in pybamm.convection.through_cell*), 55
BaseTransverseModel (*class in pybamm.convection.transverse*), 57
BasicDFN (*class in pybamm.lithium_ion*), 46
BasicFull (*class in pybamm.lead_acid*), 49
BasicSPM (*class in pybamm.lithium_ion*), 45
battery_geometry() (*in module pybamm*), 98
bc_apply() (*pybamm.ScikitFiniteElement method*), 114
BinaryOperator (*class in pybamm*), 24
boundary_conditions (*pybamm.BaseModel attribute*), 40
boundary_conditions (*pybamm.BaseSubModel attribute*), 50
boundary_integral() (*pybamm.ScikitFiniteElement method*), 114
boundary_integral() (*pybamm.SpatialMethod method*), 105
boundary_integral_vector() (*pybamm.ScikitFiniteElement method*), 114
boundary_mass_matrix() (*pybamm.ScikitFiniteElement method*), 115
boundary_value() (*in module pybamm*), 32
boundary_value_or_flux() (*pybamm.FiniteVolume method*), 110
boundary_value_or_flux() (*pybamm.ScikitFiniteElement method*), 115
boundary_value_or_flux() (*pybamm.SpatialMethod method*), 105
boundary_value_or_flux() (*pybamm.ZeroDimensionalSpatialMethod method*), 117
BoundaryGradient (*class in pybamm*), 30
BoundaryIntegral (*class in pybamm*), 29
BoundaryOperator (*class in pybamm*), 30
BoundaryValue (*class in pybamm*), 30
Broadcast (*class in pybamm*), 33
broadcast() (*pybamm.SpatialMethod method*), 106
Bruggeman (*class in pybamm.tortuosity*), 94
build() (*pybamm.Simulation method*), 125
ButlerVolmer (*class in pybamm.interface*), 72

C

calculate_consistent_state() (*pybamm.BaseSolver method*), 118
call_1D() (*pybamm.ProcessedVariable method*), 123
call_2D() (*pybamm.ProcessedVariable method*), 123
CasadiAlgebraicSolver (*class in pybamm*), 121
CasadiConverter (*class in pybamm*), 39
CasadiSolver (*class in pybamm*), 120
Chebyshev1DSubMesh (*class in pybamm*), 100
check_algebraic_equations() (*pybamm.BaseModel method*), 42
check_algebraic_equations() (*pybamm.electrolyte_conductivity.Full method*), 63
check_and_set_domains() (*pybamm.FullBroadcast method*), 33
check_and_set_domains() (*pybamm.PrimaryBroadcast method*), 34
check_and_set_domains() (*pybamm.SecondaryBroadcast method*), 34
check_default_variables_dictionaries() (*pybamm.BaseModel method*), 42
check_default_variables_dictionaries() (*pybamm.electrolyte_conductivity.Full method*), 63
check_ics_bcs() (*pybamm.BaseModel method*), 42
check_ics_bcs() (*pybamm.electrolyte_conductivity.Full method*), 64
check_initial_conditions() (*pybamm.Discretisation method*), 102
check_initial_conditions_rhs() (*pybamm.Discretisation method*), 103
check_model() (*pybamm.Discretisation method*), 103
check_tab_conditions() (*pybamm.Discretisation method*), 103
check_variables() (*pybamm.Discretisation method*), 103
check_well_determined() (*pybamm.BaseModel method*), 42
check_well_determined() (*pybamm.electrolyte_conductivity.Full method*), 64
check_well_posedness() (*pybamm.BaseModel method*), 42
check_well_posedness() (*pybamm.electrolyte_conductivity.Full method*), 64
children (*pybamm.Symbol attribute*), 16
Citations (*class in pybamm*), 130
clear_domains() (*pybamm.Symbol method*), 16
combine_submeshes() (*pybamm.Mesh method*), 98
Composite (*class in pybamm.electrode.ohm*), 60
Composite (*class in pybamm.electrolyte_conductivity*), 63
Composite (*class in pybamm.electrolyte_diffusion*), 68
Composite (*class in pybamm.lead_acid*), 48
Composite (*class in pybamm.oxygen_diffusion*), 80
CompositeExtended (*class in pybamm.lead_acid*), 48
CompositePotentialPair1plus1D (*class in pybamm.current_collector*), 52
CompositePotentialPair2plus1D (*class in pybamm.current_collector*), 52

concatenated_algebraic (*pybamm.BaseModel attribute*), 41
concatenated_initial_conditions (*pybamm.BaseModel attribute*), 41
concatenated_rhs (*pybamm.BaseModel attribute*), 41
Concatenation (*class in pybamm*), 32
concatenation() (*pybamm.FiniteVolume method*), 110
concatenation() (*pybamm.SpatialMethod method*), 106
Constant (*class in pybamm.porosity*), 87
ConstantConcentration (*class in pybamm.electrolyte_diffusion*), 67
ConstantSEI (*class in pybamm.sei*), 75
convert() (*pybamm.CasadiConverter method*), 39
convert_electric() (*pybamm.Experiment method*), 124
convert_time_to_seconds() (*pybamm.Experiment method*), 124
convert_to_format (*pybamm.BaseModel attribute*), 42
copy() (*pybamm.BaseSolver method*), 118
copy() (*pybamm.ParameterValues method*), 95
copy_domains() (*pybamm.Symbol method*), 16
Cos (*class in pybamm*), 35
cos() (*in module pybamm*), 35
Cosh (*class in pybamm*), 35
cosh() (*in module pybamm*), 35
create_jacobian() (*pybamm.Discretisation method*), 103
create_mass_matrix() (*pybamm.Discretisation method*), 103
CurrentCollector1D (*class in pybamm.thermal.pouch_cell*), 91
CurrentCollector2D (*class in pybamm.thermal.pouch_cell*), 92
CurrentControl (*class in pybamm.external_circuit*), 70

D

data (*pybamm.ProcessedSymbolicVariable attribute*), 124
data (*pybamm.ProcessedVariable attribute*), 123
default_solver (*pybamm.BaseModel attribute*), 42
default_solver (*pybamm.current_collector.AlternativeEffectiveResistance2D attribute*), 52
default_solver (*pybamm.current_collector.EffectiveResistance attribute*), 52
default_solver (*pybamm.electrolyte_conductivity.Full attribute*), 64
definite_integral_matrix() (*pybamm.FiniteVolume method*), 110
definite_integral_matrix() (*pybamm.ScikitFiniteElement method*), 115
DefiniteIntegralVector (*class in pybamm*), 29
delta_function() (*pybamm.FiniteVolume method*), 110
delta_function() (*pybamm.SpatialMethod method*), 106
DeltaFunction (*class in pybamm*), 30
DFN (*class in pybamm.lithium_ion*), 46
diff() (*pybamm.AbsoluteValue method*), 27
diff() (*pybamm.ExternalVariable method*), 21
diff() (*pybamm.Function method*), 35
diff() (*pybamm.FunctionParameter method*), 20
diff() (*pybamm.Heaviside method*), 26
diff() (*pybamm.MatrixMultiplication method*), 25
diff() (*pybamm.Sign method*), 27
diff() (*pybamm.SpatialOperator method*), 28
diff() (*pybamm.StateVector method*), 24
diff() (*pybamm.StateVectorDot method*), 24
diff() (*pybamm.Symbol method*), 17
diff() (*pybamm.Variable method*), 20
diff() (*pybamm.VariableDot method*), 21
DiffusionLimited (*class in pybamm.interface*), 75
Discretisation (*class in pybamm*), 102
div() (*in module pybamm*), 31
Divergence (*class in pybamm*), 28
divergence() (*pybamm.FiniteVolume method*), 110
divergence() (*pybamm.ScikitFiniteElement method*), 115
divergence() (*pybamm.SpatialMethod method*), 106
divergence_matrix() (*pybamm.FiniteVolume method*), 110
Division (*class in pybamm*), 25
domain (*pybamm.Symbol attribute*), 17
DomainConcatenation (*class in pybamm*), 32
DummySolver (*class in pybamm*), 119
dynamic_plot() (*in module pybamm*), 128
dynamic_plot() (*pybamm.QuickPlot method*), 128

E

edge_to_node() (*pybamm.FiniteVolume method*), 110
edit_parameter() (*in module pybamm.parameters_cli*), 131
EffectiveResistance (*class in pybamm.current_collector*), 52
ElectronMigrationLimited (*class in pybamm.sei*), 76
EqualHeaviside (*class in pybamm*), 26
evaluate() (*pybamm.BinaryOperator method*), 25
evaluate() (*pybamm.Concatenation method*), 32
evaluate() (*pybamm.EvaluatorPython method*), 38

evaluate() (*pybamm.Event method*), 45
evaluate() (*pybamm.Function method*), 35
evaluate() (*pybamm.ParameterValues method*), 95
evaluate() (*pybamm.Symbol method*), 17
evaluate() (*pybamm.UnaryOperator method*), 27
evaluate_for_shape() (*pybamm.DeltaFunction method*), 30
evaluate_for_shape() (*pybamm.Symbol method*), 17
evaluate_ignoring_errors() (*pybamm.Symbol method*), 17
evaluates_on_edges() (*pybamm.BinaryOperator method*), 25
evaluates_on_edges() (*pybamm.BoundaryIntegral method*), 30
evaluates_on_edges() (*pybamm.DeltaFunction method*), 30
evaluates_on_edges() (*pybamm.Divergence method*), 28
evaluates_on_edges() (*pybamm.FullBroadcastToEdges method*), 34
evaluates_on_edges() (*pybamm.Function method*), 35
evaluates_on_edges() (*pybamm.Gradient method*), 28
evaluates_on_edges() (*pybamm.Gradient_Squared method*), 28
evaluates_on_edges() (*pybamm.Index method*), 27
evaluates_on_edges() (*pybamm.Inner method*), 26
evaluates_on_edges() (*pybamm.Integral method*), 29
evaluates_on_edges() (*pybamm.Laplacian method*), 28
evaluates_on_edges() (*pybamm.PrimaryBroadcastToEdges method*), 34
evaluates_on_edges() (*pybamm.SecondaryBroadcastToEdges method*), 34
evaluates_on_edges() (*pybamm.Symbol method*), 17
evaluates_on_edges() (*pybamm.UnaryOperator method*), 27
evaluates_to_number() (*pybamm.Symbol method*), 17
EvaluatorPython (*class in pybamm*), 38
Event (*class in pybamm*), 44
event_type (*pybamm.Event attribute*), 44
events (*pybamm.BaseModel attribute*), 41
events (*pybamm.BaseSubModel attribute*), 50
EventType (*class in pybamm*), 45
exp() (*in module pybamm*), 35
Experiment (*class in pybamm*), 124
Explicit (*class in pybamm.convection.through_cell*), 56
Exponential (*class in pybamm*), 35
Exponential1DSubMesh (*class in pybamm*), 99
expression (*pybamm.Event attribute*), 44
ExternalVariable (*class in pybamm*), 21

F

FastManyParticles (*class in pybamm.particle*), 86
FastSingleParticle (*class in pybamm.particle*), 85
FickianManyParticles (*class in pybamm.particle*), 84
FickianSingleParticle (*class in pybamm.particle*), 83
find_parameter() (*pybamm.ParameterValues static method*), 95
FiniteVolume (*class in pybamm*), 109
FirstOrder (*class in pybamm.oxygen_diffusion*), 80
FirstOrderKinetics (*class in pybamm.interface*), 74
FOQS (*class in pybamm.lead_acid*), 48
format() (*pybamm.BinaryOperator method*), 25
format() (*pybamm.Timer method*), 130
ForwardTafel (*class in pybamm.interface*), 73
Full (*class in pybamm.convection.through_cell*), 56
Full (*class in pybamm.convection.transverse*), 58
Full (*class in pybamm.electrode.ohm*), 61
Full (*class in pybamm.electrolyte_conductivity*), 63
Full (*class in pybamm.electrolyte_diffusion*), 69
Full (*class in pybamm.lead_acid*), 49
Full (*class in pybamm.oxygen_diffusion*), 81
Full (*class in pybamm.porosity*), 88
FullAlgebraic (*class in pybamm.electrolyte_conductivity.surface_potential_form*), 66
FullBroadcast (*class in pybamm*), 33
FullBroadcastToEdges (*class in pybamm*), 34
FullDifferential (*class in pybamm.electrolyte_conductivity.surface_potential_form*), 65
Function (*class in pybamm*), 35
FunctionControl (*class in pybamm.external_circuit*), 70
FunctionParameter (*class in pybamm*), 19

G

Geometry (*class in pybamm*), 97
get() (*pybamm.ParameterValues method*), 95
get_children_auxiliary_domains() (*pybamm.Symbol method*), 18
get_children_domains() (*pybamm.BinaryOperator method*), 25

get_children_domains() (*pybamm.Function method*), 35
get_children_domains() (*pybamm.FunctionParameter method*), 20
get_coupled_variables() (*pybamm.BaseSubModel method*), 50
get_coupled_variables() (*pybamm.convection.through_cell.Explicit method*), 56
get_coupled_variables() (*pybamm.convection.through_cell.Full method*), 56
get_coupled_variables() (*pybamm.convection.through_cell.NoConvection method*), 55
get_coupled_variables() (*pybamm.convection.transverse.Uniform method*), 58
get_coupled_variables() (*pybamm.current_collector.Uniform method*), 53
get_coupled_variables() (*pybamm.electrode.ohm.Composite method*), 60
get_coupled_variables() (*pybamm.electrode.ohm.Full method*), 61
get_coupled_variables() (*pybamm.electrode.ohm.LeadingOrder method*), 60
get_coupled_variables() (*pybamm.electrode.ohm.SurfaceForm method*), 62
get_coupled_variables() (*pybamm.electrolyte_conductivity.Composite method*), 63
get_coupled_variables() (*pybamm.electrolyte_conductivity.Full method*), 64
get_coupled_variables() (*pybamm.electrolyte_conductivity.LeadingOrder method*), 62
get_coupled_variables() (*pybamm.electrolyte_diffusion.Composite method*), 68
get_coupled_variables() (*pybamm.electrolyte_diffusion.Full method*), 69
get_coupled_variables() (*pybamm.electrolyte_diffusion.LeadingOrder method*), 67
get_coupled_variables() (*pybamm.interface.BaseKinetics method*), 72
get_coupled_variables() (*pybamm.interface.DiffusionLimited method*), 75
get_coupled_variables() (*pybamm.interface.FirstOrderKinetics method*), 74
get_coupled_variables() (*pybamm.interface.inverse_kinetics.InverseButlerVolmer method*), 73
get_coupled_variables() (*pybamm.interface.InverseFirstOrderKinetics method*), 74
get_coupled_variables() (*pybamm.oxygen_diffusion.Composite method*), 80
get_coupled_variables() (*pybamm.oxygen_diffusion.FirstOrder method*), 80
get_coupled_variables() (*pybamm.oxygen_diffusion.Full method*), 81
get_coupled_variables() (*pybamm.oxygen_diffusion.LeadingOrder method*), 82
get_coupled_variables() (*pybamm.particle.FickianManyParticles method*), 84
get_coupled_variables() (*pybamm.particle.FickianSingleParticle method*), 83
get_coupled_variables() (*pybamm.porosity.Full method*), 88
get_coupled_variables() (*pybamm.porosity.LeadingOrder method*), 87
get_coupled_variables() (*pybamm.sei.ConstantSEI method*), 75
get_coupled_variables() (*pybamm.sei.ElectronMigrationLimited method*), 76
get_coupled_variables() (*pybamm.sei.InterstitialDiffusionLimited method*), 77
get_coupled_variables() (*pybamm.sei.NoSEI method*), 77
get_coupled_variables() (*pybamm.sei.ReactionLimited method*), 78
get_coupled_variables() (*pybamm.sei.SolventDiffusionLimited method*), 79
get_coupled_variables() (*pybamm.thermal.isothermal.Isothermal method*), 89
get_coupled_variables() (*pybamm.thermal.lumped.Lumped method*), 90
get_coupled_variables() (*pybamm.thermal.pouch_cell.CurrentCollector1D*

method), 91
get_coupled_variables() (*pybamm.thermal.pouch_cell.CurrentCollector2D method*), 92
get_coupled_variables() (*pybamm.thermal.x_full.OneDimensionalX method*), 90
get_coupled_variables() (*pybamm.tortuosity.Bruggeman method*), 94
get_external_variables() (*pybamm.BaseSubModel method*), 50
get_external_variables() (*pybamm.electrolyte_conductivity.Full method*), 64
get_fundamental_variables() (*pybamm.BaseSubModel method*), 50
get_fundamental_variables() (*pybamm.convection.through_cell.Full method*), 56
get_fundamental_variables() (*pybamm.convection.through_cell.NoConvection method*), 55
get_fundamental_variables() (*pybamm.convection.transverse.Full method*), 58
get_fundamental_variables() (*pybamm.convection.transverse.NoConvection method*), 57
get_fundamental_variables() (*pybamm.convection.transverse.Uniform method*), 58
get_fundamental_variables() (*pybamm.current_collector.BaseCompositePotentialPair method*), 51
get_fundamental_variables() (*pybamm.current_collector.BasePotentialPair method*), 53
get_fundamental_variables() (*pybamm.current_collector.BaseQuiteConductivePotentialPair method*), 54
get_fundamental_variables() (*pybamm.electrode.ohm.Full method*), 61
get_fundamental_variables() (*pybamm.electrolyte_conductivity.Full method*), 64
get_fundamental_variables() (*pybamm.electrolyte_diffusion.Composite method*), 69
get_fundamental_variables() (*pybamm.electrolyte_diffusion.ConstantConcentration method*), 67
get_fundamental_variables() (*pybamm.electrolyte_diffusion.Full method*), 69
get_fundamental_variables() (*pybamm.electrolyte_diffusion.LeadingOrder method*), 68
get_fundamental_variables() (*pybamm.external_circuit.CurrentControl method*), 70
get_fundamental_variables() (*pybamm.external_circuit.FunctionControl method*), 70
get_fundamental_variables() (*pybamm.interface.BaseKinetics method*), 72
get_fundamental_variables() (*pybamm.oxygen_diffusion.Full method*), 81
get_fundamental_variables() (*pybamm.oxygen_diffusion.LeadingOrder method*), 82
get_fundamental_variables() (*pybamm.oxygen_diffusion.NoOxygen method*), 83
get_fundamental_variables() (*pybamm.particle.FastManyParticles method*), 86
get_fundamental_variables() (*pybamm.particle.FastSingleParticle method*), 85
get_fundamental_variables() (*pybamm.particle.FickianManyParticles method*), 84
get_fundamental_variables() (*pybamm.particle.FickianSingleParticle method*), 84
get_fundamental_variables() (*pybamm.porosity.Constant method*), 87
get_fundamental_variables() (*pybamm.porosity.Full method*), 88
get_fundamental_variables() (*pybamm.porosity.LeadingOrder method*), 87
get_fundamental_variables() (*pybamm.sei.ConstantSEI method*), 76
get_fundamental_variables() (*pybamm.sei.ElectronMigrationLimited method*), 76
get_fundamental_variables() (*pybamm.sei.InterstitialDiffusionLimited method*), 77
get_fundamental_variables() (*pybamm.sei.NoSEI method*), 78
get_fundamental_variables() (*pybamm.sei.ReactionLimited method*), 78
get_fundamental_variables() (*pybamm.sei.SolventDiffusionLimited method*), 79
get_fundamental_variables() (*pybamm.thermal.isothermal.Isothermal method*),

89
get_fundamental_variables() (pybamm.thermal.lumped.Lumped method), 90
get_fundamental_variables() (pybamm.thermal.pouch_cell.CurrentCollector1D method), 92
get_fundamental_variables() (pybamm.thermal.pouch_cell.CurrentCollector2D method), 93
get_fundamental_variables() (pybamm.thermal.x_full.OneDimensionalX method), 91
get_infinite_nested_dict() (in module pybamm), 129
get_spatial_scale() (pybamm.ProcessedVariable method), 123
get_spatial_var() (pybamm.QuickPlot method), 128
get_termination_reason() (pybamm.BaseSolver method), 118
get_variable() (pybamm.VariableDot method), 21
get_variable_array() (pybamm.Simulation method), 126
grad() (in module pybamm), 31
grad_squared() (in module pybamm), 31
Gradient (class in pybamm), 28
gradient() (pybamm.FiniteVolume method), 110
gradient() (pybamm.ScikitFiniteElement method), 115
gradient() (pybamm.SpatialMethod method), 106
gradient_matrix() (pybamm.FiniteVolume method), 111
gradient_matrix() (pybamm.ScikitFiniteElement method), 116
Gradient_Squared (class in pybamm), 28
gradient_squared() (pybamm.ScikitFiniteElement method), 116
gradient_squared() (pybamm.SpatialMethod method), 107

H

has_symbol_of_classes() (pybamm.Symbol method), 18
Heaviside (class in pybamm), 26

I

indefinite_integral() (pybamm.FiniteVolume method), 111
indefinite_integral() (pybamm.ScikitFiniteElement method), 116
indefinite_integral() (pybamm.SpatialMethod method), 107
indefinite_integral() (pybamm.ZeroDimensionalSpatialMethod method), 117
indefinite_integral_matrix_edges() (pybamm.FiniteVolume method), 111
indefinite_integral_matrix_nodes() (pybamm.FiniteVolume method), 112
IndefiniteIntegral (class in pybamm), 29
IndependentVariable (class in pybamm), 21
Index (class in pybamm), 27
info() (pybamm.BaseModel method), 42
info() (pybamm.electrolyte_conductivity.Full method), 64
initial_conditions (pybamm.BaseModel attribute), 40
initial_conditions (pybamm.BaseSubModel attribute), 50
initialise_0D() (pybamm.ProcessedSymbolicVariable method), 124
initialise_1D() (pybamm.ProcessedSymbolicVariable method), 124
initialise_2D() (pybamm.ProcessedVariable method), 123
Inner (class in pybamm), 25
input_parameters (pybamm.BaseModel attribute), 42
input_parameters (pybamm.electrolyte_conductivity.Full attribute), 64
InputParameter (class in pybamm), 36
inputs (pybamm._BaseSolution attribute), 122
Integral (class in pybamm), 28
integral() (pybamm.FiniteVolume method), 112
integral() (pybamm.ScikitFiniteElement method), 116
integral() (pybamm.SpatialMethod method), 107
integral() (pybamm.ZeroDimensionalSpatialMethod method), 117
internal_neumann_condition() (pybamm.FiniteVolume method), 112
internal_neumann_condition() (pybamm.SpatialMethod method), 107
Interpolant (class in pybamm), 36
InterstitialDiffusionLimited (class in pybamm.sei), 77
InverseButlerVolmer (class in pybamm.interface.inverse_kinetics), 73
InverseFirstOrderKinetics (class in pybamm.interface), 74
is_constant() (pybamm.Symbol method), 18
Isothermal (class in pybamm.thermal.isothermal), 89
items() (pybamm.ParameterValues method), 95

J

jac() (*pybamm.Jacobian method*), 38
jac() (*pybamm.Symbol method*), 18
Jacobian (*class in pybamm*), 38
jacobian (*pybamm.BaseModel attribute*), 41
jacobian_algebraic (*pybamm.BaseModel attribute*), 41
jacobian_rhs (*pybamm.BaseModel attribute*), 41

K

keys() (*pybamm.ParameterValues method*), 95

L

Laplacian (*class in pybamm*), 28
laplacian() (*in module pybamm*), 31
laplacian() (*pybamm.FiniteVolume method*), 112
laplacian() (*pybamm.ScikitFiniteElement method*), 116
laplacian() (*pybamm.SpatialMethod method*), 108
LeadingOrder (*class in pybamm.electrode.ohm*), 60
LeadingOrder (*class in pybamm.electrolyte_conductivity*), 62
LeadingOrder (*class in pybamm.electrolyte_diffusion*), 67
LeadingOrder (*class in pybamm.oxygen_diffusion*), 82
LeadingOrder (*class in pybamm.porosity*), 87
LeadingOrderAlgebraic (*class in pybamm.electrolyte_conductivity.surface_potential_form*), 66
LeadingOrderDifferential (*class in pybamm.electrolyte_conductivity.surface_potential_form*), 66
linspace() (*in module pybamm*), 23
load_function() (*in module pybamm*), 130
Log (*class in pybamm*), 35
log() (*in module pybamm*), 36
LOQS (*class in pybamm.lead_acid*), 47
Lumped (*class in pybamm.thermal.lumped*), 89

M

Mass (*class in pybamm*), 28
mass_matrix (*pybamm.BaseModel attribute*), 41
mass_matrix() (*pybamm.ScikitFiniteElement method*), 116
mass_matrix() (*pybamm.SpatialMethod method*), 108
mass_matrix() (*pybamm.ZeroDimensionalSpatialMethod method*), 117
mass_matrix_inv (*pybamm.BaseModel attribute*), 41
Matrix (*class in pybamm*), 23
MatrixMultiplication (*class in pybamm*), 25
max() (*in module pybamm*), 36
Maximum (*class in pybamm*), 26
maximum() (*in module pybamm*), 26
Mesh (*class in pybamm*), 98
MeshGenerator (*class in pybamm*), 98
meshgrid() (*in module pybamm*), 23
min() (*in module pybamm*), 36
Minimum (*class in pybamm*), 26
minimum() (*in module pybamm*), 26
model (*pybamm._BaseSolution attribute*), 122
Multiplication (*class in pybamm*), 25

N

name (*pybamm.BaseModel attribute*), 40
name (*pybamm.Event attribute*), 44
name (*pybamm.Symbol attribute*), 18
ndim (*pybamm.Array attribute*), 23
Negate (*class in pybamm*), 27
new_copy() (*pybamm.Array method*), 23
new_copy() (*pybamm.BaseModel method*), 42
new_copy() (*pybamm.BinaryOperator method*), 25
new_copy() (*pybamm.Concatenation method*), 32
new_copy() (*pybamm.electrolyte_conductivity.Full method*), 64
new_copy() (*pybamm.Function method*), 35
new_copy() (*pybamm.FunctionParameter method*), 20
new_copy() (*pybamm.InputParameter method*), 36
new_copy() (*pybamm.Parameter method*), 19
new_copy() (*pybamm.Scalar method*), 22
new_copy() (*pybamm.SpatialVariable method*), 22
new_copy() (*pybamm.Symbol method*), 18
new_copy() (*pybamm.Time method*), 22
new_copy() (*pybamm.UnaryOperator method*), 27
NoConvection (*class in pybamm.convection.through_cell*), 55
NoConvection (*class in pybamm.convection.transverse*), 57
node_to_edge() (*pybamm.FiniteVolume method*), 112
NoOxygen (*class in pybamm.oxygen_diffusion*), 83
NoReaction (*class in pybamm.interface*), 72
NoSEI (*class in pybamm.sei*), 77
NotEqualHeaviside (*class in pybamm*), 26
NumpyConcatenation (*class in pybamm*), 32

O

on_boundary() (*pybamm.ScikitSubMesh2D method*), 101
OneDimensionalX (*class in pybamm.thermal.x_full*), 90
ones_like() (*in module pybamm*), 34
options (*pybamm.BaseBatteryModel attribute*), 43
options (*pybamm.BaseModel attribute*), 40

orphans (*pybamm.Symbol attribute*), 18

P

param (*pybamm.BaseSubModel attribute*), 49
Parameter (*class in pybamm*), 19
parameters (*pybamm.BaseModel attribute*), 42
parameters (*pybamm.electrolyte_conductivity.Full attribute*), 64
parameters (*pybamm.Geometry attribute*), 98
ParameterValues (*class in pybamm*), 94
plot() (*in module pybamm*), 129
plot() (*pybamm.QuickPlot method*), 128
plot() (*pybamm.Simulation method*), 126
plot2D() (*in module pybamm*), 129
post_process() (*pybamm.current_collector.AlternativeEffectiveResistance2D method*), 52
post_process() (*pybamm.current_collector.EffectiveResistance method*), 52
PotentialPair1plus1D (*class in pybamm.current_collector*), 54
PotentialPair2plus1D (*class in pybamm.current_collector*), 54
Power (*class in pybamm*), 25
PowerFunctionControl (*class in pybamm.external_circuit*), 71
pre_order() (*pybamm.Symbol method*), 18
preprocess_external_variables() (*pybamm.FiniteVolume method*), 112
PrimaryBroadcast (*class in pybamm*), 33
PrimaryBroadcastToEdges (*class in pybamm*), 34
print() (*pybamm.Citations method*), 131
print_citations() (*in module pybamm*), 131
print_evaluated_parameters() (*pybamm.ParameterValues method*), 95
print_parameters() (*pybamm.ParameterValues method*), 95
process_binary_operators() (*pybamm.FiniteVolume method*), 113
process_binary_operators() (*pybamm.SpatialMethod method*), 108
process_boundary_conditions() (*pybamm.Discretisation method*), 103
process_boundary_conditions() (*pybamm.ParameterValues method*), 96
process_dict() (*pybamm.Discretisation method*), 104
process_geometry() (*pybamm.ParameterValues method*), 96
process_initial_conditions() (*pybamm.Discretisation method*), 104
process_model() (*pybamm.Discretisation method*), 104
process_model() (*pybamm.ParameterValues method*), 96
process_parameters_and_discretise() (*pybamm.BaseBatteryModel method*), 44
process_rhs_and_algebraic() (*pybamm.Discretisation method*), 104
process_symbol() (*pybamm.Discretisation method*), 104
process_symbol() (*pybamm.ParameterValues method*), 96
ProcessedSymbolicVariable (*class in pybamm*), 123
ProcessedVariable (*class in pybamm*), 123
pybamm (*module*), 15
pybamm.parameters.electrical_parameters (*module*), 97
pybamm.parameters.geometric_parameters (*module*), 97
pybamm.parameters.parameter_sets (*module*), 97
pybamm.parameters.standard_parameters_lead_acid (*module*), 97
pybamm.parameters.standard_parameters_lithium_ion (*module*), 97
pybamm.parameters.thermal_parameters (*module*), 97

Q

QuickPlot (*class in pybamm*), 127
QuiteConductivePotentialPair1plus1D (*class in pybamm.current_collector*), 54
QuiteConductivePotentialPair2plus1D (*class in pybamm.current_collector*), 54

R

r_average() (*in module pybamm*), 31
ReactionLimited (*class in pybamm.sei*), 78
read_citations() (*pybamm.Citations method*), 131
read_operating_conditions() (*pybamm.Experiment method*), 125
read_parameters_csv() (*pybamm.ParameterValues method*), 96
read_string() (*pybamm.Experiment method*), 125
register() (*pybamm.Citations method*), 131
relabel_tree() (*pybamm.Symbol method*), 18
remove_parameter() (*in module pybamm.parameters_cli*), 131
render() (*pybamm.Symbol method*), 18
reset() (*pybamm.Simulation method*), 126
reset() (*pybamm.Timer method*), 130
reset_axis() (*pybamm.QuickPlot method*), 128
rhs (*pybamm.BaseModel attribute*), 40
rhs (*pybamm.BaseSubModel attribute*), 49
rmse() (*in module pybamm*), 130

root_dir() (*in module pybamm*), 130

S

save() (*pybamm._BaseSolution method*), 122
save() (*pybamm.Simulation method*), 126
save_data() (*pybamm._BaseSolution method*), 122
Scalar (*class in pybamm*), 22
ScikitChebyshev2DSubMesh (*class in pybamm*), 101
ScikitExponential2DSubMesh (*class in pybamm*), 101
ScikitFiniteElement (*class in pybamm*), 114
ScikitsDaeSolver (*class in pybamm*), 120
ScikitsOdeSolver (*class in pybamm*), 120
ScikitSubMesh2D (*class in pybamm*), 101
ScikitUniform2DSubMesh (*class in pybamm*), 101
ScipySolver (*class in pybamm*), 119
search() (*pybamm.ParameterValues method*), 96
secondary_domain (*pybamm.Symbol attribute*), 19
SecondaryBroadcast (*class in pybamm*), 34
SecondaryBroadcastToEdges (*class in pybamm*), 34
sensitivity() (*pybamm.ProcessedSymbolicVariable method*), 124
set_algebraic() (*pybamm.BaseSubModel method*), 50
set_algebraic() (*pybamm.convection.through_cell.Full method*), 56
set_algebraic() (*pybamm.convection.transverse.Full method*), 58
set_algebraic() (*pybamm.current_collector.BasePotentialPair method*), 53
set_algebraic() (*pybamm.current_collector.BaseQuiteConductivePotentialPair method*), 54
set_algebraic() (*pybamm.electrode.ohm.Full method*), 61
set_algebraic() (*pybamm.electrolyte_conductivity.Full method*), 65
set_algebraic() (*pybamm.electrolyte_conductivity.surface_potential_form.FullAlgebraic method*), 66
set_algebraic() (*pybamm.electrolyte_conductivity.surface_potential_form.LeadingOrderAlgebraic method*), 66
set_algebraic() (*pybamm.external_circuit.FunctionControl method*), 71
set_algebraic() (*pybamm.interface.BaseKinetics method*), 72
set_boundary_conditions() (*pybamm.BaseSubModel method*), 51
set_boundary_conditions() (*pybamm.convection.through_cell.Full method*), 57
set_boundary_conditions() (*pybamm.convection.transverse.Full method*), 58
set_boundary_conditions() (*pybamm.current_collector.PotentialPair1plus1D method*), 54
set_boundary_conditions() (*pybamm.current_collector.PotentialPair2plus1D method*), 54
set_boundary_conditions() (*pybamm.electrode.ohm.BaseModel method*), 59
set_boundary_conditions() (*pybamm.electrode.ohm.Composite method*), 60
set_boundary_conditions() (*pybamm.electrode.ohm.Full method*), 61
set_boundary_conditions() (*pybamm.electrode.ohm.LeadingOrder method*), 60
set_boundary_conditions() (*pybamm.electrolyte_conductivity.BaseElectrolyteConductivity method*), 62
set_boundary_conditions() (*pybamm.electrolyte_conductivity.Full method*), 65
set_boundary_conditions() (*pybamm.electrolyte_diffusion.Composite method*), 69
set_boundary_conditions() (*pybamm.electrolyte_diffusion.Full method*), 70
set_boundary_conditions() (*pybamm.oxygen_diffusion.Full method*), 81
set_boundary_conditions() (*pybamm.particle.FickianManyParticles method*), 85
set_boundary_conditions() (*pybamm.particle.FickianSingleParticle method*), 84
set_boundary_conditions() (*pybamm.thermal.pouch_cell.CurrentCollector1D method*), 92
set_boundary_conditions() (*pybamm.thermal.pouch_cell.CurrentCollector2D method*), 93
set_boundary_conditions() (*py-*

bamm.thermal.x_full.OneDimensionalX method), 91
set_defaults() (*pybamm.Simulation method*), 126
set_events() (*pybamm.BaseSubModel method*), 51
set_events() (*pybamm.electrolyte_conductivity.Full method*), 65
set_events() (*pybamm.electrolyte_diffusion.BaseElectrolyteDiffusion method*), 67
set_events() (*pybamm.particle.BaseParticle method*), 83
set_events() (*pybamm.porosity.BaseModel method*), 86
set_expected_size() (*pybamm.InputParameter method*), 36
set_external_circuit_submodel() (*pybamm.BaseBatteryModel method*), 44
set_external_circuit_submodel() (*pybamm.lead_acid.LOQS method*), 47
set_external_variables() (*pybamm.Discretisation method*), 105
set_full_convection_submodel() (*pybamm.lead_acid.BaseHigherOrderModel method*), 48
set_full_interface_submodel() (*pybamm.lead_acid.BaseHigherOrderModel method*), 48
set_full_porosity_submodel() (*pybamm.lead_acid.BaseHigherOrderModel method*), 48
set_full_porosity_submodel() (*pybamm.lead_acid.Composite method*), 48
set_full_porosity_submodel() (*pybamm.lead_acid.FOQS method*), 48
set_id() (*pybamm.Array method*), 23
set_id() (*pybamm.BoundaryIntegral method*), 30
set_id() (*pybamm.BoundaryOperator method*), 30
set_id() (*pybamm.DefiniteIntegralVector method*), 29
set_id() (*pybamm.DeltaFunction method*), 30
set_id() (*pybamm.FunctionParameter method*), 20
set_id() (*pybamm.Index method*), 27
set_id() (*pybamm.Integral method*), 29
set_id() (*pybamm.Interpolant method*), 37
set_id() (*pybamm.Scalar method*), 22
set_id() (*pybamm.Symbol method*), 19
set_initial_conditions() (*pybamm.BaseSubModel method*), 51
set_initial_conditions() (*pybamm.convection.through_cell.Full method*), 57
set_initial_conditions() (*pybamm.convection.transverse.Full method*), 59
set_initial_conditions() (*pybamm.current_collector.BasePotentialPair method*), 53
set_initial_conditions() (*pybamm.current_collector.BaseQuiteConductivePotentialPair method*), 54
set_initial_conditions() (*pybamm.electrode.ohm.Full method*), 61
set_initial_conditions() (*pybamm.electrolyte_conductivity.Full method*), 65
set_initial_conditions() (*pybamm.electrolyte_diffusion.Composite method*), 69
set_initial_conditions() (*pybamm.electrolyte_diffusion.Full method*), 70
set_initial_conditions() (*pybamm.electrolyte_diffusion.LeadingOrder method*), 68
set_initial_conditions() (*pybamm.external_circuit.FunctionControl method*), 71
set_initial_conditions() (*pybamm.interface.BaseKinetics method*), 72
set_initial_conditions() (*pybamm.oxygen_diffusion.Full method*), 81
set_initial_conditions() (*pybamm.oxygen_diffusion.LeadingOrder method*), 82
set_initial_conditions() (*pybamm.particle.FastManyParticles method*), 86
set_initial_conditions() (*pybamm.particle.FastSingleParticle method*), 85
set_initial_conditions() (*pybamm.particle.FickianManyParticles method*), 85
set_initial_conditions() (*pybamm.particle.FickianSingleParticle method*), 84
set_initial_conditions() (*pybamm.porosity.Full method*), 88
set_initial_conditions() (*pybamm.porosity.LeadingOrder method*), 87
set_initial_conditions() (*pybamm.sei.ElectronMigrationLimited method*), 76
set_initial_conditions() (*pybamm.sei.InterstitialDiffusionLimited method*), 77
set_initial_conditions() (*pybamm.sei.ReactionLimited method*), 78
set_initial_conditions() (*pybamm.sei.SolventDiffusionLimited method*),

79
set_initial_conditions() (pybamm.thermal.lumped.Lumped method), 90
set_initial_conditions() (pybamm.thermal.pouch_cell.CurrentCollector1D method), 92
set_initial_conditions() (pybamm.thermal.pouch_cell.CurrentCollector2D method), 93
set_initial_conditions() (pybamm.thermal.x_full.OneDimensionalX method), 91
set_internal_boundary_conditions() (pybamm.Discretisation method), 105
set_parameters() (pybamm.Simulation method), 126
set_rhs() (pybamm.BaseSubModel method), 51
set_rhs() (pybamm.electrolyte_conductivity.Full method), 65
set_rhs() (pybamm.electrolyte_conductivity.surface_potential_form.FullDifferential method), 65
set_rhs() (pybamm.electrolyte_conductivity.surface_potential_form.LeadingOrderDifferential method), 66
set_rhs() (pybamm.electrolyte_diffusion.Composite method), 69
set_rhs() (pybamm.electrolyte_diffusion.Full method), 70
set_rhs() (pybamm.electrolyte_diffusion.LeadingOrder method), 68
set_rhs() (pybamm.oxygen_diffusion.Composite method), 80
set_rhs() (pybamm.oxygen_diffusion.Full method), 81
set_rhs() (pybamm.oxygen_diffusion.LeadingOrder method), 82
set_rhs() (pybamm.particle.FastManyParticles method), 86
set_rhs() (pybamm.particle.FastSingleParticle method), 85
set_rhs() (pybamm.particle.FickianManyParticles method), 85
set_rhs() (pybamm.particle.FickianSingleParticle method), 84
set_rhs() (pybamm.porosity.Full method), 88
set_rhs() (pybamm.porosity.LeadingOrder method), 88
set_rhs() (pybamm.sei.ElectronMigrationLimited method), 76
set_rhs() (pybamm.sei.InterstitialDiffusionLimited method), 77
set_rhs() (pybamm.sei.ReactionLimited method), 78
set_rhs() (pybamm.sei.SolventDiffusionLimited method), 79
set_rhs() (pybamm.thermal.lumped.Lumped method), 90
set_rhs() (pybamm.thermal.pouch_cell.CurrentCollector1D method), 92
set_rhs() (pybamm.thermal.pouch_cell.CurrentCollector2D method), 93
set_rhs() (pybamm.thermal.x_full.OneDimensionalX method), 91
set_soc_variables() (pybamm.BaseBatteryModel method), 44
set_soc_variables() (pybamm.lead_acid.BaseModel method), 47
set_up() (pybamm.BaseSolver method), 118
set_up_experiment() (pybamm.Simulation method), 126
set_variable_slices() (pybamm.Discretisation method), 105
shape (pybamm.Array attribute), 23
shape (pybamm.Symbol attribute), 19
shape_for_testing (pybamm.Symbol attribute), 19
simplify() (pybamm.FiniteVolume method), 113
Sign (class in pybamm), 27
simplify() (pybamm.LeadingOrderDifferential method), 32
Simplification (class in pybamm), 37
simplify() (pybamm.Simplification method), 37
simplify() (pybamm.Symbol method), 19
simplify_addition_subtraction() (in module pybamm), 37
simplify_if_constant() (in module pybamm), 37
simplify_multiplication_division() (in module pybamm), 38
Simulation (class in pybamm), 125
Sin (class in pybamm), 36
sin() (in module pybamm), 36
Sinh (class in pybamm), 36
sinh() (in module pybamm), 36
size (pybamm.ExternalVariable attribute), 21
size (pybamm.Symbol attribute), 19
size_for_testing (pybamm.Symbol attribute), 19
slider_update() (pybamm.QuickPlot method), 128
Solution (class in pybamm), 123
solve() (pybamm.BaseSolver method), 118
solve() (pybamm.Simulation method), 126
SolventDiffusionLimited (class in pybamm.sei), 79
source() (in module pybamm), 26
SparseStack (class in pybamm), 33
spatial_variable() (pybamm.FiniteVolume method), 113
spatial_variable() (pybamm.ScikitFiniteElement method), 117
spatial_variable() (pybamm.SpatialMethod method), 108

SpatialMethod (*class in pybamm*), 105
SpatialOperator (*class in pybamm*), 28
SpatialVariable (*class in pybamm*), 22
SpecificFunction (*class in pybamm*), 35
specs() (*pybamm.Simulation method*), 126
SPM (*class in pybamm.lithium_ion*), 45
SPMe (*class in pybamm.lithium_ion*), 46
StateVector (*class in pybamm*), 23
StateVectorDot (*class in pybamm*), 24
step() (*pybamm.BaseSolver method*), 119
step() (*pybamm.Simulation method*), 127
stiffness_matrix() (*pybamm.ScikitFiniteElement method*), 117
sub_solutions (*pybamm.Solution attribute*), 123
SubMesh (*class in pybamm*), 98
SubMesh0D (*class in pybamm*), 99
SubMesh1D (*class in pybamm*), 99
Subtraction (*class in pybamm*), 25
surf() (*in module pybamm*), 31
SurfaceForm (*class in pybamm.electrode.ohm*), 61
Symbol (*class in pybamm*), 15
SymbolUnpacker (*class in pybamm*), 39

T

t (*in module pybamm*), 22
t (*pybamm._BaseSolution attribute*), 122
t_event (*pybamm._BaseSolution attribute*), 122
termination (*pybamm._BaseSolution attribute*), 122
test_shape() (*pybamm.Symbol method*), 19
Time (*class in pybamm*), 22
time() (*pybamm.Timer method*), 130
Timer (*class in pybamm*), 130
timescale (*pybamm.BaseModel attribute*), 42
timescale (*pybamm.electrolyte_conductivity.Full attribute*), 65
to_casadi() (*pybamm.Symbol method*), 19

U

UnaryOperator (*class in pybamm*), 27
Uniform (*class in pybamm.convection.transverse*), 58
Uniform (*class in pybamm.current_collector*), 53
Uniform1DSubMesh (*class in pybamm*), 99
unpack_list_of_symbols() (*pybamm.SymbolUnpacker method*), 39
unpack_symbol() (*pybamm.SymbolUnpacker method*), 39
update() (*pybamm._BaseSolution method*), 122
update() (*pybamm.BaseModel method*), 42
update() (*pybamm.electrolyte_conductivity.Full method*), 65
update() (*pybamm.ParameterValues method*), 96
update_from_chemistry() (*pybamm.ParameterValues method*), 97
use_jacobian (*pybamm.BaseModel attribute*), 41
use_simplify (*pybamm.BaseModel attribute*), 41
UserSupplied1DSubMesh (*class in pybamm*), 100
UserSupplied2DSubMesh (*class in pybamm*), 102

V

value (*pybamm.Scalar attribute*), 22
value() (*pybamm.ProcessedSymbolicVariable method*), 124
value_and_sensitivity() (*pybamm.ProcessedSymbolicVariable method*), 124
values() (*pybamm.ParameterValues method*), 97
Variable (*class in pybamm*), 20
VariableDot (*class in pybamm*), 20
variables (*pybamm.BaseModel attribute*), 40
variables (*pybamm.BaseSubModel attribute*), 50
Vector (*class in pybamm*), 23
visualise() (*pybamm.Symbol method*), 19
VoltageFunctionControl (*class in pybamm.external_circuit*), 71

X

x_average() (*in module pybamm*), 31

Y

y (*pybamm._BaseSolution attribute*), 122
y_event (*pybamm._BaseSolution attribute*), 123
yz_average() (*in module pybamm*), 32

Z

z_average() (*in module pybamm*), 32
ZeroDimensionalSpatialMethod (*class in pybamm*), 117